Doc. No.: 00-0058/N1281
Date: 6 November 2000
Author: Performance WG

Performance TR - Working Paper

Contents:

1 Introduction

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.

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. It has been a guiding principle from the earliest days of C++ that "you don't pay for what you don't use". Language features that are never used in a program should not have a cost in extra code size, memory size, or run time. If there are places where C++ cannot guarantee zero overheads for unused features, this paper will attempt to document them. It will also discuss ways in which compiler writers, library vendors, and programmers can minimize or eliminate performance penalties, and will discuss the trade offs among different methods of implementation.

Programming for resource-constrained environments is another focus of this paper. Typically, it is very small or very large programs that run into resource limits of some kind. Very large programs, such as database servers, may run into limits of disk space or virtual memory. At the other extreme, an embedded application may be constrained to run in the ROM and RAM space provided by a single chip, perhaps a total of 64K of memory, or even smaller.

Apart from the issues of resource limits, some programs must interface with system hardware on a very low level. Historically the interfaces to hardware have been implemented as proprietary extensions to the compiler (often as macros). This led to the situation that code has not been portable, even for programs written for a given environment, because each compiler for that environment has implemented different sets of extensions.

2 Overheads – C++ Cost of Using Features

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?

- 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 from RTTI
- Overheads from Templates
- Overheads from 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:

  In addition to the syntax of casts in C, for example:

  ```
  int i = (int)3.14159;
  ```

  Standard C++ adds four additional forms of casting, using syntax that looks like function-templates1 for example:

  ```
  int i = static_cast<int> ( 3.14159 );
  ```

  The four syntactic forms are:

  - `const_cast<Type>(expression)`
  - `static_cast<Type>(expression)`
  - `reinterpret_cast<Type>(expression)`
  - `dynamic_cast<Type>(expression)`

  These perform the same functions as the C-style cast, which is still recognized, but distinguish between the different purposes for which a

1 Indeed, prototype implementations of the so-called new-style casts were often implemented as function-templates.

cast is used. The syntax of new casts is easier to identify in source code, and thus may contribute to programs that are more correct. If the compiler does not provide new-style casts natively, it is possible to implement them using template functions.

The first three forms of new-style cast have no size or speed penalty; versus the equivalent C-style cast notation. dynamic_cast<T> may incur some overhead at run time, if the required conversion involves using RTTI mechanisms (e.g. cross-casting).

It should also be pointed out that as in C, a cast may create a temporary object of the desired type, so casting can have run time implications.

- Compile-time evaluation, like locales in libraries.

2.1 Overheads from Inheritance

2.1.1 Overhead examples

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

2.1.2 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.

Question 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: virtual-functions on left branch of tree; virtual-functions on right branch
  - Static function calls are slightly faster than ordinary member functions (less than 25%)

- No significant difference in runtime speed between ordinary function calls, virtual function calls, and virtual function calls among different branches of MI.
- Function calls are cheaper than they used to be (compared to inline)
- Virtual function 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).*
- Cross-casts are more expensive. A cross-cast 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 function 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.

2.1.3 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 functions 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]

2.1.4 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.)

2.1.5 Virtual Inheritance Overheads

[??? No notes]

2.1.6 Class Hierarchies Overheads

[??? No notes]

2.1.7 Unnecessary costs for empty base Overheads

[??? No notes]

2.2 Overheads from Exception Handling

2.2.1 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.

2.2.1.1 The "dynamic" approach.

- try

  Save the 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.

2.2.1.2 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.

2.2.1.3 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.

2.2.1.4 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 bookkeeping (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.

2.2.1.5 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.

2.2.1.6 Missing stuff

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

- Advice to implementers, specifically references to literature on EH (e.g. ‘C’ Language Translation).
- Potential implementation pitfalls.
- A comparison of the costs of other strategies.

2.2.2 Myths and Reality of Exception Handling Overhead

2.2.2.1 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 themselves invoke 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.

2.2.2.2 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*

2.2.2.3 Run-time Overhead

Two main strategies: setjmp model and table model..

2.2.2.3.1 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

2.2.2.3.2 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

2.2.3 Predictability of Exception Handling Overhead

2.2.3.1 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.

2.2.3.2 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();
    // }
}
```

2.2.3.3 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?

2.2.4 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). For 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).

2.3 Overheads from Templates

2.3.1 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 edition *[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.

2.3.2 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 CopyConstructable; 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.

Templates can provide powerful facilities for evaluation at compile-time. Doing more of the work at compile time means less work at run time.

Hints can be exchanged between the compiler and the library, to select a more efficient specialization, or to select linkage with a reduced-footprint version of the library. In C, it's possible to optimise printf this way:

When you see printf, if __crt_float is defined, then invoke printf_float, else invoke printf_int. Defining a float f; has the side effect of defining __crt_float.

3 Performance – Techniques & Strategies

Description of current approaches

- Code generation control, including memory placement, initialization characteristics, et al.
- #pragma, 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

3.1 Programmer Directed Optimisations

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 optimisation.

- 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 initialise it. This prevents initialising the variable twice.
- In constructors, prefer initialisation of data members to assignment. These are the steps taken to construct a variable of class type: first, any initialisations specified in the member initialisation list are performed. Next, other members of class type (but not primitive types) are initialised by their default constructor, if one is available. Only then is the body of the constructor executed.
- Passing arguments to a function by value [e.g. `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 [e.g. `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 [e.g. `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 initialising 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 really
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, and 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.

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** can 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 optimising 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.

3.2 ROM-ability

ROM-ability is important for embedded programs whose code and data must be stored in ROM. Objects without const-qualifier can be modified. Both ROM and RAM area must be allocated for such objects, if they have constant initialisation. The definition of ROM-able object here is an object that needs only ROM area.

The embedded programs whose memories are very tight require the compilers to identify strictly ROM-able objects and allocate ROM area only for them.

3.2.1 ROM-able objects

The following objects should be ROM-able:

3.2.1.1 <User-defined objects>

The const-qualified objects that are initialised with constant expressions. Examples:

- The aggregate (IS 8.5.1) object with static storage duration (3.7.1) whose initialisers are all constants

```
static const int tab[] = {1,2,3};
```

- Scalar type objects with external linkage

  Although scalar type objects with internal linkage are ROM-able, if they are not used for initialization or assignment of pointer/reference variables, only values are used at the time of compilation, i.e., object data areas are not allocated.

  Therefore, it is not expected to allocate data area for such objects even in ROM area *[Note: Objects which are const-qualified and are not explicitly declared to be extern have internal linkage(7.1.5.1)]*.

```
extern const int a = 1; // extern linkage
const int b = 1;        // internal linkage
const int *c = &b;      // variable b should be allocated
const int tbsize = 256; // it is expected that tbsize is not
                        // allocated at run-time
char ctb[tbsize];
```

- String literals

  String literals are const-qualified array of char (IS 2.13.4), and so they are ROM-able. But when it is used as the initializer of a character array, and the variable to be initialized is not a const-qualified array of char, it is not ROM-able.

```
const char *str1 = "abc"; // ROM-able
char str2[] = "abc";      // not ROM-able
```

3.2.1.2 <Compiler-generated objects>

- Virtual function tables

  If the virtual function of a class becomes static after linkage, it is expected that the table is ROM-able.

- Jump tables for switch statements

  If a jump table is generated to implement switch statement, the table is expected to be ROM-able.

- Type identification tables

  When a table is generated to identify RTTI types, the table is expected to be ROM-able.

- Exception tables

  When exception handling is implemented by a static table, it is expected that the table is ROM-able.

- Reference to constants

  If a constant expression is specified as the initialiser for a const-qualified reference, a temporary object is generated (8.5.3).

This temporary object is expected to be allocated to ROM.

```
const double & a = 2.0;       // interpreted as follows
static const double tb = 2.0; // tb can be in ROM
const double & b = tb;
```

- Initialisers for aggregate objects with automatic storage duration

  If all initialisers for an aggregate object that has automatic storage duration are constant expressions, a temporary object that has the value of the constant expressions and a code that copies the value of the temporary object to the aggregate object may be generated.

  This temporary object is expected to be allocated to ROM.

```
void test() {
    struct A {int a,b,c;};
    A a = {1,2,3};                // may be interpreted as:
    static const A tb = {1,2,3};  // tb can be in ROM
    A b = tb;
}
```

- constants generated during code generation

  Some constants such as integer constants, floating point constants and address constants may not be the part of instruction code but the data. These data are stored in memory and loaded when they are used.

  They are expected to be allocated to ROM.

```
void test() {
    double a;
    a += 1.0;                     // may be interpreted as:
    static const double t = 1.0;  // t can be in ROM
    const double *tp = &t;
    a += *tp;
}
```

3.2.2 Constructors and ROM-able objects

Even though constant expressions are specified as the initialisers for all the members of const qualified class object that has user-declared constructors (12.1), the initialisation will be done dynamically in general. But in this case, because the initialisation can be done statically by analysing the constructors, the optimisation so as to allocate the class object to ROM is expected to be implemented.

```
class A { public:
    int a;
    A(int v) : a(v) { }
};
const A tab[2] = {1,2};
```

3.2.3 Guide for Users

Even though constant expressions are specified as the initialisers for all the members of const qualified class object that has constructors, the initialisation will be done dynamically in general.

Note that such class object may not be expected to ROM.

```
class A { public:
    int a;
    A(int v) : a(v) { }
};
const A tab[2] = {1,2};    // A(int) may be used for initialisation
```

A class object which has private member, destructor, base-class or virtual function should not be initialised statically (8.5.1). Also, a class object that has non-POD type members will be initialised dynamically (IS 12.6.1).

```
class complex {
    // ...
public:
    complex(double);
    // ...
};
class X { public:
    int i;
    complex c;
};
const X x = {99,77.7};   // complex(double) is applied for X.c
```

4 Embedded Systems – Special Needs

4.1 BASIC I/O HARDWARE ADDRESSING

4.1.1 Scope

As the C language has matured over the years various extensions for doing basic I/O hardware register addressing have been added to address limitations and weaknesses in the language, and today almost all C compilers for freestanding environments and embedded systems support direct access to I/O hardware registers from the C source level. However, these extensions have not been consistent across dialects. As a growing number of C++ compiler vendors are now entering the same market, the same I/O driver portability problems become apparent for C++.

This Technical Report is a step towards codifying common existing practice in the market and providing a single uniform syntax for basic I/O hardware register addressing.

4.1.2 Rationale

Ideally, it should be possible to compile C or C++ source code, which operates directly on I/O hardware, registers with different compiler implementations for different platforms and get the same logical behaviour at runtime. As a simple portability goal the driver source code for a given I/O hardware should be portable to all processor architectures where hardware itself can be connected.

The problem areas are the same for C and C++, and the standardization method proposed is applicable for both languages.

4.1.3 Standardization objectives

4.1.3.1 Basic requirements

A standardization method for basic I/O hardware addressing must be able to fulfil three requirements at the same time:

- The standardized syntax must not prevent compilers from producing machine code with absolutely no overhead compared to the code produced by the existing non-standardized solutions. This speed requirement is essential in order to get widespread acceptance from the market
- The I/O driver source code modules should be completely portable to any processor system (from 8-bit systems and up) without needing any modification to the driver source code itself. I.e. the syntax should promote *I/O driver source code portability* across different execution environments
- The syntax should provide an *encapsulation* of the underlying access mechanisms to allow different access methods, different processor architectures, and different bus systems to be used with the same I/O driver source code

I.e. the standardization method should separate the characteristics of the I/O register itself from the characteristics of the underlying execution environment (processor architecture, bus system, addresses, alignment, endian, etc.)

4.1.3.2 New perception of I/O registers simplifies the syntax standardization.

There have been several different attempts to create an international standard for the general syntax for basic I/O operations over the years, but these all failed to meet the special requirements of the embedded market and the market for freestanding environments.

The major reason for this is twofold: 1) that I/O registers have usually been treated as another type of "memory" and 2) that I/O register access has been thought of as something related to processor busses and address ranges.

The I/O standardization method used here overcomes these limitations by treating I/O registers as individual objects with individual properties which are fixed and independent of both the compiler implementation and the surrounding processor system.

It is worth noting that although the overall aim in standardizing basic I/O hardware addressing is to promote portability of library source code, the major challenge is to provide a standardized solution which does not reduce execution performance, especially with respect to speed and code size overheads.

It is important to keep in mind that standardized I/O access does *not* mean standardized hardware. The goal is to standardize the *syntax* for I/O operations, not their platform functionality.

4.1.3.3 Typical I/O register characteristics

An I/O register has a fixed size and endian, which are independent of how standard C types are implemented by different compiler vendors and independent of the access methods supported by different processor architectures and bus systems.

Most important is the fact that I/O registers do not usually behave like memory cells. I/O registers have special individual characteristics:

- write-only (Unidirectional)
- read-only (Unidirectional)
- read-once (New data at each read)
- write-once (Each write triggers a new event)
- read-write (Bi-directional, but read != write)
- read-modify-write (Memory like)

Individual bits in an I/O register may have individual characteristics. Only true read-modify-write registers behave like memory cells. The above list also shows that the default should be that I/O registers be treated as *volatile* data types.

4.1.3.4 Access method encapsulation

Since processor architectures and hardware platforms ARE different, standardization must also provide a method of separating the description of the hardware differences and addressing methods from the source-code. The standardization method should *encapsulate* descriptions of hardware differences, e.g. in a separate header file.

The best way to encapsulate differences in allowed I/O access methods, and at the same time to create a uniform syntax for I/O access, is by using a few standardized I/O *functions (*or *class member functions).* This corresponds to the way encapsulation is done in the spirit of C/C++. (The functions may be implemented as in-line functions, macros, or templates for speed optimisation.)

4.1.3.5 Reduced basic operation set

Normally, arithmetic operations on I/O registers cannot be performed or have no logical meaning. Often read-modify-write operations on I/O registers are prohibited by the actual hardware. Operators like: +=, -=, *=, /=, >>=, <<=, ++, --, etc. are only meaningful where the I/O register and the bus architecture both allow read-modify-write operations. These natural access limitations make it obvious that a standard only has to define functions for the most basic operations on I/O registers.

A standardization method must define basic *read* and *write* operations as a minimum. In addition, the *iohw* header includes functions for the most common I/O register operations: set, clear, and invert for one or more register bits.

The programmer can build all other arithmetic and logical operations on top of these few basic I/O access operations.

4.1.3.6 Handling of intrinsic types

With many existing processor architectures, I/O register access often requires the use of special machine instructions to operate on special I/O address ranges.

This means that an extension of the type system is needed so that I/O registers can be accessed from the C/C++ source level. If a *function-syntax* is used for standardized I/O access, all use of processor and platform specific I/O access types (implementation specific types) will be limited to the implementation of these basic I/O functions and to the definition of the *access type* for a register object.

In this way, the language can define a basic I/O hardware addressing syntax that is portable to any processor system, without extending the type system defined by the C/C++ standard.

It is worth noting that although a function syntax makes basic I/O hardware addressing functions look like traditional library functions (API functions), the main underlying intention is to create a portable way of extending the type system with compiler (processor and platform) specific *access types*.

4.2 Basic concepts

4.2.1 Simple conceptual model for I/O registers

The I/O syntax standardization method creates a conceptually simple model for I/O registers:

Symbolic name for I/O port = I/O register object definition.

Example:

```
#include <iohw>
unsigned char mybuf[10];
//..
iowr8(MYPORT1, 0x8);                 // write single register
for (int i = 0; i < 10; i++)
    mybuf[i] = iordbuf8(MYPORT2, i); // read register array
```

The programmer only sees the characteristics of the I/O register itself. The underlying platform, bus architecture, and compiler implementation do not matter during driver programming. The underlying system hardware may later be changed without modifications to the I/O driver source code being necessary.

4.2.2 The access_type parameter

A new *access_type* type is used by the standardized I/O functions.

Example:

```
uint_8t iord8(access_type);        // Read from I/O register
void iowr8(access_type, uint_8t);  // Write to I/O register
```

The access_type *parameter represents or references a complete description of how the I/O hardware register should be addressed in the given hardware platform. It is an abstract type with a well-defined behaviour.*

The implementation of *access_types* will be processor and platform specific. The definition of an I/O register object may or may not require a memory instantiation, depending on how a compiler vendor has chosen to implement *access_types*. For maximum performance, this could be a simple definition based on compiler specific address range and type qualifiers, in which case no instantiation of an *access_type* object would be needed in data memory.

Footnote: This use of an abstract type is similar to the philosophy behind the well-known FILE type. Some general properties for FILE and streams are defined in the standard, but the standard deliberately avoids telling how the underlying file system should be implemented or initialised.

4.2.3 Exact sized data types

The data parameter and return parameters used in the I/O functions are integer data types with an exact (minimum) bit precision.

I/O registers have a fixed size independent of how a compiler implements the standard integer types. Data values for use with I/O registers should therefore always have an exact (minimum) size independent of the compiler implementation.

Another reason for the exact-sized types is to allow the programmer to decide what precision is needed by the application. A programmer can then do code optimisation without the risk of running into the portability problems which exist with the old *int* and *long* types.

For instance, with smaller processor architectures it is often very performance expensive, with respect to execution speed and code size, if a program uses integer data types with a precision greater than needed by the application. Fixed sized data types are therefore a performance issue and not just related to I/O.

The exact-sized data types are currently defined in the header file <**stdint.h**> (ISO/IEC 99 Programming Language C). It is suggested that these types be adopted by C++.

4.2.4 I/O initialisation

With respect to the standardization process, it is important to make a clear distinction between I/O hardware (chip) related initialisation and platform related initialisation. Typically, three types of initialisation are related to I/O:

- I/O hardware (chip) initialisation.
- I/O selector initialisation.
- I/O access initialisation.

Here only I/O access initialisation (3) is relevant for basic I/O hardware addressing.

I/O hardware initialisation is a natural part of a hardware driver and should always be considered as a part of the I/O driver application itself. This initialisation is done using the standard functions for basic I/O hardware addressing. I/O hardware initialisation is therefore not a topic for this standardization process.

I/O selector initialisation is used when, for instance, the same I/O driver code needs to service multiple I/O hardware chips of the same type.

One solution is to define multiple *access_type* objects, one for each of the hardware chips, and then have the *access_type* passed to the driver functions from a calling function.

I.e. *Instead of the usual (platform dependent) I/O selector initialisation, it now becomes a matter of selecting between standardized* access_type *objects*.

Note, this means that it is important that a standardization method does not prevent a compiler implementation from generating efficient code for *access_type* parameter passing. (This is an area, which typically creates performance problems with implementations for C99 compilers). Apart from this performance issue, I/O selector initialization is not a problem with respect to basic I/O hardware addressing.

I/O access initialization concerns the initialization and definition of *access_type* objects.

This process is implementation defined. It depends both on the platform and processor architecture and on which underlying access methods are supported by an *iohw* implementation.

With most freestanding environments and embedded systems the platform hardware is well defined, so all *access_types* for I/O registers used by the program can be completely defined at compile time. For such platforms, standardized I/O access initialization is not a standardization issue.

With larger processor systems I/O hardware is often allocated dynamically at runtime. Here the *access_type* information can only be partly defined at compile time. Some platform software dependent part of it must be initialized at runtime.

When designing the *access_type* object a compiler implementer should therefore make a clear distinction between *static information* and *dynamic information* – i.e. what can be defined and initialized at compile time and what must be initialized at runtime.

Depending on the implementation method and depending on whether the *access_type* objects need to contain dynamic information, the *access_type* object may or may not require an instantiation in data memory. If more of the information is static, a better execution performance can usually be achieved.

4.3 The <iohw> header

4.3.1 Overview

The header file **<iohw>** defines a number of functions which:

- Support the most common fixed register sizes.
  - 8-bit, 16-bit, 32-bit, 64-bit or 1-bit (logical)
- Support the most basic I/O register operations.
  - Read, Write,
  - Bit-set (Or) in register, bit-clear (And) in register, bit-invert (Xor) in register.
  - Single register objects, register array objects.
- Define new abstract types for I/O register referencing: *access_type*(s)
- Provide a uniform encapsulation method for hardware and platform differences.
- Provide a uniform header file name. <iohw>

4.3.2 Single register access

The I/O access functions defined here are for operations on a single register object. The functions are defined for 8-, 16-, 32-, 64- and 1-bit register sizes:

```
/* Read operations */
uint_8t  iord8(access_type_8);
uint_16t iord16(access_type_16);
uint_32t iord32(access_type_32);
uint_64t iord64(access_type_64);
bool     iord1(access_type_1);

/* Write operations: */
void iowr8(access_type_8, uint_8t);
void iowr16(access_type_16, uint_16t);
void iowr32(access_type_32, uint_32t);
void iowr64(access_type_64, uint_64t);
void iowr1(access_type_1, bool);

/* AND operations (Clear group of bits) */
void ioand8(access_type_8, uint_8t);
void ioand16(access_type_16, uint_16t);
void ioand32(access_type_32, uint_32t);
void ioand64(access_type_64, uint_64t);
void ioand1(access_type_1, bool);

/* OR operations (Set group of bits) */
void ioor8(access_type_8, uint_8t);
void ioor16(access_type_16, uint_16t);
void ioor32(access_type_32, uint_32t);
void ioor64(access_type_64, uint_64t);
void ioor1(access_type_1, bool);

/* XOR operations (Invert group of bits) */
void ioxor8(access_type_8, uint_8t);
void ioxor16(access_type_16, uint_16t);
void ioxor32(access_type_32, uint_32t);
void ioxor64(access_type_64, uint_64t);
void ioxor1(access_type_1, bool);
```

The one-bit functions (ioxxx1) access a single bit in a register. The access_type_1 parameter defines both how to access the I/O register and the position of the bit in the register.

4.3.3 Register array access

This refers to I/O functions for operations on I/O register array objects. It could be I/O circuitry with internal buffers or multiple registers, e.g. a peripheral chip with a linear hardware buffer.

The *index* parameter is offset in the buffer (or register array) starting from the I/O location specified by *access_type*, where element 0 is the first element located at the address defined by *access_type*, and element n+1 is located at a higher physical address than element n.

```
/* Read operations on hardware buffers */
uint_8t iordbuf8(access_type_8, unsigned int index);
uint_16t iordbuf16(access_type_16, unsigned int index);
uint_32t iordbuf32(access_type_32, unsigned int index);
uint_64t iordbuf64(access_type_64, unsigned int index);

/* Write operations on hardware buffers */
void iowrbuf8(access_type_8, unsigned int index, uint_8t dat);
void iowrbuf16(access_type_16, unsigned int index, uint_16t dat);
void iowrbuf32(access_type_32, unsigned int index, uint_32t dat);
void iowrbuf64(access_type_64, unsigned int index, uint_64t dat);

/* AND operations on hardware buffers (Clear group of bits)*/
void ioandbuf8(access_type_8, unsigned int index, uint_8t dat);
void ioandbuf16(access_type_16, unsigned int index, uint_16t dat);
void ioandbuf32(access_type_32, unsigned int index, uint_32t dat);
void ioandbuf64(access_type_64, unsigned int index, uint_64t dat);

/* OR operations on hardware buffers (Set group of bits) */
void ioorbuf8(access_type_8, unsigned int index, uint_8t dat);
void ioorbuf16(access_type_16, unsigned int index, uint_16t dat);
void ioorbuf32(access_type_32, unsigned int index, uint_32t dat);
void ioorbuf64(access_type_64, unsigned int index, uint_64t dat);

/* XOR operations on hardware buffers (Invert group of bits) */
void ioxorbuf8(access_type_8, unsigned int index, uint_8t dat);
void ioxorbuf16(access_type_16, unsigned int index, uint_16t dat);
void ioxorbuf32(access_type_32, unsigned int index, uint_32t dat);
void ioxorbuf64(access_type_64, unsigned int index, uint_64t dat);
```

In addition to these I/O function definitions, *iohw.h* must also contain definitions for *access_types* and must define the fixed sized types (perhaps by including <**stdint.h**>)

4.3.4 Common function syntax for C and C++

It would be beneficial, especially in the market for freestanding environments and embedded systems, if source code for I/O hardware drivers could be written so that the driver code can be compiled with both C and C++ compilers. In this market C compilers are still dominant.

With a C-like syntax for basic I/O hardware access, users could benefit from a broader range of source library products from third party vendors. A common syntax would also ensure a smoother transition in this market from C to C++.

Therefore, it is recommended that the same C function syntax for basic I/O hardware addressing be used with both C and C++.

Note that it is only the standardized function interfaces that need to be C-like. *iohw* and *access_type* implementations for C and C++ may be very different in both performance and the number of access methods supported. An implementation for C++ can still take advantage of templates, classes and other advanced C++ features.

Appendix A: Implementing the iohw header

(A guide for implementers)

A.1 Purpose

The *iohw* header defines standardized function syntax for basic I/O hardware addressing. This header should normally be created by the compiler vendor.

The idea behind the standardized syntax is that the source code looks the same independent of where the I/O register is located in the hardware and independent of the underlying method used to address the I/O hardware register.

While this standardized function syntax for basic I/O hardware addressing provides a simple, easy-to-use method for a programmer to write portable and hardware-platform-independent I/O driver code, nevertheless the *iohw* header implementation itself may require careful consideration to achieve an efficient implementation.

This chapter gives some guidelines for implementers on how to implement the *iohw* header in a relatively straightforward manner given a specific processor and bus architecture.

Recommended steps

Briefly, the recommended steps for implementing the *iohw* header are:

- Get an overview of all the possible and relevant ways the I/O register hardware is typically connected with the given bus hardware architectures, and get an overview of the basic software methods typically used to address such I/O hardware registers.
- Define a number of I/O functions, macros and *access_types* which support the relevant I/O access methods for the given compiler market.
- Provide a way to pick the right I/O function at compile time and generate the right machine code based on the *access_type* type or the *access_type* value.

A.1.1 Compiler considerations

In practice, an implementation will often require that very different machine code is generated for different I/O access cases. Furthermore, with some processor architectures, I/O hardware access will require the generation of special machine instructions not used otherwise when generating code for the traditional C, C++ memory model.

Selection between different code generation possibilities must be determined solely by the *access_type* declaration for each I/O register. Whenever possible this access method selection should be implemented, so it is done entirely at compile time in order to avoid any runtime or machine code overhead.

For a compiler vendor, of course, selection between code generation possibilities can always be implemented by supporting different intrinsic *access_type* types and keywords designed specially for the given processor architecture and additional to the traditional types and keywords defined by the language.

However, with a conforming C++ compiler, an efficient and all-round implementation of the *iohw* header can usually be made using template functionality. A template solution allows the number of compiler specific intrinsic I/O types or intrinsic I/O functions to be minimized or even removed completely, depending on the processor architecture.

For compilers not supporting templates (such as C compilers) other implementation methods must be used. In any case, at least the most basic *iohw* functionality can be implemented efficiently using a mixture of macros, *in-line* functions and intrinsic types or functions. Full feature *iohw* implementations will usually require direct compiler support (or extensions to the language).

The considerations described in the following are generally applicable to both C and C++ compilers. However, in this document it is primarily C++ template-based solutions that are used in the implementation examples.

A.2 Overview of I/O hardware connection options

The various ways an I/O register can be connected to processor hardware are primarily determined by combinations of the following three hardware characteristics:

- The bit width of the logical I/O register
- The bit width of the data-bus of the I/O chip
- The bit width of the processor-bus

A.2.1 Multi-addressing and I/O register endian

If the width of the logical I/O register is greater than the width of the I/O chip data bus, an I/O access operation will require multiple consecutive addressing operations.

The I/O register endian information describes whether the MSB or the LSB byte of the *logical I/O register* is located at the *lowest* processor bus address.

(Note that the I/O register endian has nothing to do with the endian of the underlying processor hardware architecture).

Table: Logical I/O register / I/O chip addressing overview

Logical I/O register widths	I/O chip bus widths							
	8-bit chip bus		*16-bit chip bus*		*32-bit chip bus*		*64-bit chip bus*	
	LSB-MSB	MSB-LSB	LSB-MSB	MSB-LSB	LSB-MSB	MSB-LSB	LSB-MSB	MSB-LSB
8-bit register	direct		n.a.		n.a.		n.a.	
16-bit register	r8{0-1}	r8{1-0}	Direct		n.a.		n.a.	
32-bit register	r8{0-3}	r8{3-0}	r16{0-1}	r16{1-0}	Direct		n.a.	
64-bit register	r8{0-7}	r8{7-0}	r16{0,3}	r16{3,0}	R32{0,1}	r32{1,0}	Direct	

(For byte-aligned address ranges)

A.2.2 Address Interleave

If the size of the I/O chip data bus is less than the size of the processor data bus, buffer register addressing will require the use of *address interleave*.

Example:

If the processor architecture has a byte-aligned addressing range and a 32-bit processor data bus, and an 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.

Table: Interleave overview: (bus to bus interleave relations)

I/O chip bus widths	Processor bus widths			
	8-bit bus	*16-bit bus*	*32-bit bus*	*64-bit bus*
8-bit chip bus	Interleave 1	interleave 2	Interleave 4	interleave 8
16-bit chip bus	n.a.	interleave 2	Interleave 4	interleave 8
32-bit chip bus	n.a.	n.a.	Interleave 4	interleave 8
64-bit chip bus	n.a.	n.a.	n.a.	interleave 8

(For byte-aligned address ranges)

A.2.3 I/O connection overview:

The two tables above can be combined and will then show all relevant cases for how I/O hardware registers can be connected to a given processor hardware bus.

Table: Interleave between adjacent I/O registers in buffer (all cases).

I/O Register width	*Chip bus*			*Processor data bus width*			
	Bus width	*LSB MSB*	*No. Opr.*	*width=8*	*width=16*	*width=32*	*width=64*
				size 1	*size 2*	*size 4*	*size 8*
8-bit	*8-bit*	*n.a.*	*1*	**1**	**2**	**4**	**8**
16-bit	*8-bit*	*LSB*	*2*	**2**	**4**	**8**	**16**
		MSB	*2*	**2**	**4**	**8**	**16**
	16-bit	*n.a.*	*1*	n.a.	**2**	**4**	**8**
32-bit	*8-bit*	*LSB*	*4*	**4**	**8**	**16**	**32**
		MSB	*4*	**4**	**8**	**16**	**32**
	16-bit	*LSB*	*2*	n.a.	**4**	**8**	**16**
		MSB	*2*	n.a.	**4**	**8**	**16**
	32-bit	*n.a.*	*1*	n.a.	n.a.	**4**	**8**
64-bit	*8-bit*	*MSB*	*8*	**8**	**16**	**32**	**64**
		LSB	*8*	**8**	**16**	**32**	**64**
	16-bit	*LSB*	*4*	n.a.	**8**	**16**	**32**
		MSB	*4*	n.a.	**8**	**16**	**32**
	32-bit	*LSB*	*2*	n.a.	n.a.	**8**	**16**
		MSB	*2*	n.a.	n.a.	**8**	**16**
	64-bit	*n.a.*	*1*	n.a.	n.a.	n.a.	**8**

(For byte-aligned address ranges)

A.2.4 Generic buffer index

The interleave distance between two logically adjacent registers in an I/O register array can be calculated from:

- The size of the logical I/O register in bytes
- The processor data bus width in bytes
- The chip data bus width in bytes

Conversion from I/O register index to address offset can be calculated using the following generic formula:

```
Address_offset = index *
                  sizeof( logical_IO_register ) *
                   sizeof( processor_data_bus ) /
                    sizeof( chip_data_bus )
```

where a byte-aligned address range is assumed, the widths are a whole number of bytes, the width of the l*ogical_IO_register* is greater than or equal to the width of the *chip_data_bus*, and the width of the *chip_data_bus* is less than or equal to the *processor_data_bus*.

A.3 Exact sized data types

I/O registers have exact bit widths, which are independent of the compiler integer implementations and the integer types supported by the processor hardware.

An *iohw* implementation should therefore define exact width types for at least 8, 16, 32 and 64-bit registers. The definition should map the exact width integer types to an unsigned integer type of a matching width given by the compiler implementation (and optional compilation modes).

The exact width types can be defined directly in the *iohw* header. Alternatively, the *iohw* can include the C99 standard header *stdint.h.*

Example:

```
// Definition of exact sized data types for the compiler.
// (In C99 these types are defined by the header stdint.h)
#define uint8_t  unsigned char
#define uint16_t unsigned short
#define uint32_t unsigned long
#define uint64_t unsigned long long
```

A.3.1 Other register sizes

The most common widths of I/O chip busses and I/O registers are 2^N. However, if other bus and I/O register widths are native for a given processor architecture (for instance 24 bits) a vendor is free to extend an *iohw* implementation and incorporate functions for any odd-sized register widths along the line: iord24(..), iowr24(..), etc.[2]

2 As such processor-specific I/O registers are hardly ever used in other platforms, cross-platform portability will seldom be an issue for driver code which uses such iord24(..), iowr24(..) functions.

A.4 Implementation of the access type *parameter:*

The number of I/O access functions, which must be implemented in order to cover all connection cases, depends on how the access type parameter is constructed. As shown in the following table an implementation can, with advantage, have at least the processor bus width as a separate parameter:

Table: Number of access functions for each access method.

Data bus widths supported by processor	*8-bit*	*8–16-bit*	*8–32-bit*	*8–64-bit*
Total number of access cases	7	19	34	50
Number of access functions to implement when using a processor bus size parameter	7	12	15	16

The possible I/O chip connections can be described with a single parameter, which combines information about the I/O chip data bus width and the I/O register endian. The possible I/O register to bus connections can therefore be completely specified using only two parameters:

- A bus parameter, which specifies access relations between the I/O chip data bus and the processor data bus
- A multi-addressing and endian parameter, which specifies access relations between the logical I/O register and the I/O chip data bus

Example 1:

Definition of general I/O register connection types:

```
typedef enum {bw8 = 1, bw16 = 2, bw32 = 4, bw64 = 8} bus_t;
typedef enum {chip8, chip8l, chip8h, chip16, chip16l, chip16h,
    chip32, chip32l, chip32h, chip64} chip_t;
```

Example 2:

Possible I/O register connections with the processor H8/300H (supporting only an 8-bit and a 16-bit processor data bus)

```
typedef enum bus_t {
    bw8 = 1, bw16 = 2
} bus_t;
typedef enum chip_t {
    chip8, chip8l, chip8h, chip16, chip16l, chip16h
} chip_t;
```

Example 3:

Template implementation of access_type for direct addressing of memory-mapped I/O registers:

```
typedef uint32_t address_t; // Address range type.

// Define an access_type template for direct addressing, combining
// IO register width, I/O address, processor bus width, chip bus width
// and endian

template<class T, address_t address, bus_t buswidth,
         chip_t chiptype>
    class IO_MM { };

// Use of access_type in I/O register declarations made by user
// (normally placed in a separate platform dependent header file)

// An 8-bit register in an 8-bit chip using 8-bit processor bus mode
typedef IO_MM<uint8_t,  41000, bw8, chip8>  PORT1;
// a 32-bit register in an 8-bit chip using 16-bit processor bus mode
typedef IO_MM<uint32_t, 41800, bw16, chip8l> PORT2;
```

A.4.1 *Access_types* for different processor busses

If the processor architecture has multiple different addressing ranges (i.e. it requires different sets of machine instructions for the different busses), each addressing range should have its own set of *access_type* specifications.

Example:

The 80x86 processor architecture has two addressing ranges: the normal memory bus with memory-mapped I/O and a separate bus for I/O only. Implementations for the 80x86 processor family will therefore require at least two sets of access_type specifications.

```
typedef uint32_t address_t; // Memory-mapped address range
typedef uint16_t io_addr_t; // IO-mapped address range

template <class T, address_t address, bus_t buswidth,
          chip_t chiptype>
    class IO_MM { };
template <class T, io_addr_t address, bus_t buswidth,
          chip_t chiptype>
    class IO_IOM { };
```

A.4.2 *Access_types* for different I/O addressing methods

An implementer should consider the following typical addressing methods:

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

- Variable base address + constant offset. I.e. the *access_type* must contain an address pair (address of base register + offset address).

  The user-defined base address is normally initialized at runtime (by some platform-dependent part of the program). This also enables a set of I/O driver functions to be used with multiple instances of the same I/O hardware

- *Indexed bus addressing*
- Also called orthogonal or pseudo-bus addressing. It is a common way to connect a large number of I/O registers to a bus, while still only occupying a few addresses in the processor address space
- This is how it works: First the index address (or pseudo-address) of the I/O register is written to an address bus register located at a given processor address. Then the data read/write operation on the pseudo-bus is done via the following processor address. I.e. the *access_type* must contain an address pair (the processor address of indexed bus, and the pseudo-bus address (or index) of the I/O register itself).

  This access method also makes it particularly easy for a user to connect common I/O chips, which have a multiplexed address/data bus, to a processor platform with non-multiplexed busses using a minimum amount of glue logic. The driver source code for such an I/O chip is then automatically made portable to both types of bus architecture

- *Access via user-defined access driver functions*
- These are typically used with larger platforms and with small single chip processors (e.g. to emulate an external bus). In this case the *access_type* must contain pointers or references to access functions,

  The access driver solution makes it possible to connect a given I/O driver source library to any kind of platform hardware and platform software using the appropriate platform-specific interface functions

In general, an implementation should always support the simplest addressing case; whether it is the constant address or base address method that is used will depend on the processor architecture. Apart from this, an implementer is free to add any additional cases required to satisfy a given market.

Because of the different number of parameters required in an *access_type* specification and because of the different parameter ranges used, it is often convenient to define a number of different *access_type* formats for the different access methods.

For the same reasons it is often convenient to implement the iord1, iowr1, ioor1, ioand1, and ioxor1 functions so that they use their own *access_type* format, simply because of the extra parameters needed to specify the bit position in the register.

Example:

```
// Define types used in access_type declarations
typedef uint32_t address_t;      // Memory mapped address range
typedef uint8_t  sub_address_t;  // Sub address on indexed bus
typedef uint16_t io_addr_t;      // User I/O driver address
typedef uint8_t  bit_pos_t;      // Bit position in register

// Define access_type template for direct addressing
template <class T, address_t address, bus_t buswidth,
          chip_t chiptype>
    class IO_MM { };

// Define access_type template for addressing via base register
template <class T, address_t * base, address_t offset,
          bus_t buswidth, chip_t chiptype>
    class IO_MM_BASE { };

// Define access_type template for indexed bus addressing
template <class T, address_t address, sub_address_t idx,
          bus_t buswidth, chip_t chiptype>
    class IO_MM_IDX { };

// Define access_type for user-supplied access driver functions
template<class T, io_addr_t address,
         T iord ( io_addr_t address),
                  void iowr( io_addr_t address, T val)>
    class IO_MM_DRV { };

// Define access_type for direct addressing of bit in register
template<class T, address_t address, bit_pos_t bitpos,
         bus_t buswidth, chip_t chiptype>
    class IO_MM_BIT { };
```

A.4.3 Detection of read / write violations in I/O registers

The *access_type* specification can be extended with an extra parameter, which makes it possible to detect illegal use of an I/O register at compile time.

The minimal parameter set for a read / write limitation specification would be:

- Defined as Read-only register
- Defined as Write-only register
- Defined as Read-modify-Write register (behaves like a RAM cell)

Table: Allowed operations on different I/O register types:

	iowrxx	**iordxx**	**Ioorxx**	**ioandxx**	**ioxorxx**
Read-Write	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*
Write-only	*Yes*	*No*	*No*	*No*	*No*
Read-only	*No*	*Yes*	*No*	*No*	*No*

The not-allowed cases should generate some kind of error message at compile time. With a template implementation of *iohw*, the compiler will usually at least complain that no matching template function can be found for the not-allowed cases.

Example:

```
// Define type to validate I/O register access
enum rw_t          // Access mode type
{
    read,          // Read only access
    write,         // Write only access
    read_write     // Read, Write or Read-Modify-Write access
  };

// Define access_type template for direct addressing
template <class T, address_t address, bus_t buswidth,
          chip_t chiptype, rw_t access>
    class IO_MM { };

// User declaration of I/O registers in platform (normally placed in a
// separate platform dependent header file)
typedef IO_MM <uint8_t, 10800, bw8, chip8, write>      WR_PORT;
typedef IO_MM <uint8_t, 20800, bw8, chip8, read>       RD_PORT;
typedef IO_MM <uint8_t, 30800, bw8, chip8, read_write> RDWR_PORT;

// User code
uint8_t myval;
myval = iord8(RD_PORT);       // ok
myval += iord8(RDWR_PORT);    // ok
iowr8(WR_PORT,myval);         // ok
iowr8(RDWR_PORT,0x45);        // ok

myval = iord8(WR_PORT);       // Illegal, generate compile time error
iowr8(RD_PORT,0x55);          // Illegal, generate compile time error
```

A.5 Other implementation considerations

A.5.1 Atomic operation

It is an *iohw* implementation requirement that in each I/O function a given (partial) I/O register is addressed exactly once during a read or a write operation and exactly twice during a read-modify-write operation.

It is an *iohw* implementation recommendation that each I/O function be implemented so that the I/O access operation becomes *atomic* whenever possible.

However, atomic operation is not guaranteed to be portable across platforms for read-modify-write operations (*ioorxx, ioandxx*, *ioxorxx*) or for multi-addressing cases.

The reason for this is simply that many processor architectures do not have the instruction set features required for assuring atomic operation.

A.5.2 Read-modify-write operations in multi-addressing cases.

Read-modify-write operations should, in general, do a complete read of the I/O register, followed by the operation, followed by a complete write to the I/O register.

It is therefore recommended that an implementation of multi-addressing cases *should not* use read-modify-write machine instructions during *partial* register addressing operations.

The rationale for this restriction is to use the lowest common denominator of multi-addressing hardware implementations in order to support as wide a range of I/O hardware register implementation as possible.

For instance, more advanced multi-addressing I/O register implementations often take a snap-shot of the whole logical I/O register when the first partial register is being read, so that data will be stable and consistent during the whole read operation. Similarly, write registers are often made double-buffered so that a consistent data set is presented to the internal logic at the time when the access operation is completed by the last partial write.

Such hardware implementations often require that each access operation is completed before the next access operation is initiated.

A.6 Typical implementation optimization possibilities

Pre-calculation of constant expressions

All constant expressions should be solved at compile time. Using inline functions, both interleave factors and constant buffer indexes should therefore be folded into the address value(s) used in the machine code.

Therefore, the following two I/O write statements should result in exactly the same machine code:

```
iowr8(PORT1,0x33);
iowrbuf8(PORT1, 0, 0x33);
```

An implementation can take advantage of this, because the number of I/O functions that have to be implemented can be reduced with no efficiency penalty using simple macro definitions like:

```
#define iowr8(access_type,val)  iowrbuf8(access_type,0,(val))
```

Multi-addressing and endian

Typical candidates for platform dependent optimization are I/O functions for the multi-addressing cases (logical I/O register width > I/O chip bus width) where the width of the chip data bus matches the width of the processor data bus. In these cases, multi-byte access can often use data types directly supported by the processor for either the LSB or MSB endian functions. The other endian functions can often be implemented efficiently using one load or store operation plus one or more register swap operations.

Appendix B: Bibliography

Alexander, Rene, and Graham Bensley
C++ Footprint and Performance Optimization
Sams Publishing, 2000

More general than the Bulka-Mayhew book, and omits any mention of the containers and algorithms in the C++ Standard Library.

Bulka, Dov, and David Mayhew
Efficient C++: Performance Programming Techniques
Addison-Wesley, 2000

Contains many specific low-level techniques for improving time performance, with measurements to illustrate their effectiveness.

"If used properly, C++ can yield software systems exhibiting not just acceptable performance, but superior software performance."

Glass, Robert L
Software Runaways: Lessons Learned from Massive Software Project Failures
Prentice Hall PTR, 1998.

Written from a management perspective rather than a technical one, this book makes the point that a major reason why some software projects have been classified as massive failures is for failing to meet their requirements for performance.

"Of all the technology problems noted earlier, the most dominant one in our own findings in this book is that performance is a frequent cause of failure. A fairly large number of our runaway projects were real-time in nature, and it was not uncommon to find that the project could not achieve the response times and/or functional performance times demanded by the original requirements."

Hewlett-Packard Corp.
CXperf User's Guide

`http://docs.hp.com/hpux/onlinedocs/B6323-96001/B6323-96001.html`

This guide describes the CXperf Performance Analyzer, an interactive runtime performance analysis tool for programs compiled with HP ANSI C (c89), ANSI C++ (aCC), Fortran 90 (f90), and HP Parallel 32-bit Fortran 77 (f77) compilers. This guide helps you prepare your programs for profiling, run the programs, and analyze the resulting performance data.

IBM
AIX Versions 3.2 and 4 Performance Tuning Guide, 5th Edition (April 1996)

`http://www.rs6000.ibm.com/doc_link/en_US\`
`/a_doc_lib/aixbman/prftungd/toc.htm`

An extensive discussion of performance issues in many areas, such as CPU use, disk I/O, and memory management, and even the performance effects of shared libraries. It discusses AIX tools available to measure performance, and the compiler options, which can be used to optimize an application for space or time. The chapter "Design and Implementation of Efficient Programs"

`http://www.rs6000.ibm.com/doc_link/en_US\`
`/a_doc_lib/aixbman/prftungd/desnimpl.htm`

includes low-level recommendations such as these:

"Whenever possible, use int instead of char or short. In most cases, char and short data items take more instructions to manipulate. The extra instructions cost time, and, except in large arrays, any space that is saved by using the smaller data types is more than offset by the increased size of the executable. If you have to use a char, make it unsigned, if possible. A signed char takes another two instructions more than an unsigned char each time the variable is loaded into a register."

Lajoie, Joseé
"Exception Handling: Behind the Scenes."
(Included in **C++ Gems**, edited by Stanley B. Lippman)
SIGS Reference Library, 1996.

A brief overview of the C++ language features, which support exception handling, and of the underlying mechanisms necessary to support these features.

Lippman, Stan
Inside the C++ Object Model

Explains typical implementations and overheads of various C++ language features, such as multiple inheritance and virtual functions. A good in-depth look at the internals of typical implementations.

Mitchell, Mark
Type-Based Alias Analysis
Dr. Dobbs' Journal, October 2000.

Some techniques for writing source code that is easier for a compiler to optimize.

"Although C++ is often criticized as being too slow for high-performance applications, ... C++ can actually enable compilers to create code that is even faster than the C equivalent."

Stroustrup, Bjarne
The C++ Programming Language, 3rd Edition
Addison-Wesley, 1998

The definitive language manual from the language's author. This edition reflects the latest committee draft of the ISO C++ language standard.