

z/TPF C/C++ Performance Hints

Jim Tison and Bob Kopac, IBM ® TPF Development Lab, Poughkeepsie, NY 12601 USA

Abstract

For those accustomed to programming with the z/OS ® C/C++ compiler for the TPF 4.1 system, the application landscape might seem identical in the z/TPF system. It is not. The performance guidelines you might have used in the past changed a little. This article explains these changes as applicable to **gcc** and **g++**.

Background

In the z/TPF system, everything is implemented as a *shared object library*; even traditional TPF assembler segments are members of a z/TPF-unique shared object type called a *BAL shared object* (BSO). There are no longer distinctions between DLL, DLM, or LLM library types. A shared object library must have at least one member and can consist of any number of members as needed to suit your design and implementation requirements.

Any shared object can have only one entry point that is callable by an ENTxC service; but it may have multiple external entry points that are callable by external references. These types of entry points are resolved by the linkage editor and by the z/TPF system on a dynamic basis, invoking the system's enter/back service.

Calls between members of the same shared object library are resolved without enter/back intervention. This linkage method is much faster than enter/back because the system does not have to participate.

There is no longer an entity called "writeable static storage", as there was in TPF 4.1. In the z/TPF system, writable data pages (note the terminology change from "writeable static") do not cause copy activity *unless you write to them*. The pages are mapped to the current ECB virtual memory (EVM) as read-only shared pages and handled with *copy-on-write* processing in your EVM. The first time any copy-on-write page is written to (even for just one byte), the *entire* page is copied to a private page that is mapped to your EVM; the code uses this private page instead of the read-only copy. This operation is triggered by a hardware exception called a *page fault*, which also has performance costs associated with it.

The compiler family and the object module formats have changed. The z/OS C/C++ compiler is no longer used and is replaced by **gcc** and **g++**. The PM1-PM3 load module and OBJ/XOBJ relocatable object formats were replaced by a single format called ELF, which is very similar in its relocatable (unlinked) and absolute (linked) forms and more readily understood than its predecessors. Every detail of the ELF format is well-documented.

The following recommendations take all of these factors into consideration.

Considerations for C language only:

1. ***Disable the "common block" unless you understand its implications and costs.***

This Fortran-like extension of the C language is unique to **gcc**, and is enabled by default. Use the `-fno-common` command line option to disable it. While inspired by Fortran, the term "common block" as applied to **gcc** is a misnomer; *individual data*

objects are marked common in the relocatable object, not grouped into isolated blocks. Common data objects may reside in any ELF section.

Use of this feature causes all data objects declared in file scope to be marked common in the relocatable object. A later link edit operation against two or more different relocatable objects maps *identically named data objects marked common into the same storage address, **irrespective of their possibly different types***. This behavior can cause some seemingly unreproducible and asynchronous run-time errors when it is used naïvely.

You can use the common block safely only if you are: (a) aware that it is in effect, and (b) understand its effects and scope. **g++** does not use the common block, so if you choose to use it, understand that it is restricted to C only. Otherwise, use the `-fno-common` option.

Using the common block means that you are going to write to a copy-on-write memory page. Avoid this practice for performance reasons.

2. Use static inline functions instead of preprocessor macros whenever possible.

The compiler generates inline code at the call point for any function that is defined as static inline. This action omits the linkage overhead and external reference generation. Inlined functions are just as fast as preprocessor macros and much easier to maintain, because you can represent complex logic without having to worry about preprocessor rules. **gcc** will not generate any inline code at `-O0`; you must optimize at level `-O1` or higher.

Considerations common to C and C++:

1. Avoid altering constants.

Not only will altering constants risk unnecessary copy-on-write activity, but certain optimization techniques *merge constants*. Altering an in-memory value that you *think* is constant and isolated might have adverse consequences.

2. It is possible to debug optimized code.

There is no longer any reason to develop and test at no optimization (`-O0`). You can safely debug at all levels of optimization. Understand that certain optimizations might reorder instructions, remove variables from the debugger view, eliminate common expressions, eliminate useless instructions, or eliminate intermediate data objects; any of which can be confusing as you debug. Depending on your tolerance for these kinds of changes to your original code, you might wish to lower the optimization level during unit test. You are free to make the choice that suits you.

3. The compiler is better than you are at optimizing code.

Do not try to optimize code by moving statements around, sequencing automatic storage, or engaging in similar techniques -- the compiler will do this anyway as long as you choose to optimize at levels 1 through 3. Instead, cooperate with the optimizer by inlining aggressively.

4. Consider using the maximum level of optimization (-O3) instead of lower levels.

In **gcc** and **g++**, there are over 100 possible optimization techniques that you can select individually, or you can enable over 80 of the most useful of these by simply passing the level 3 optimization option (-O3) on the command line. Most significantly, the compilers attempt to generate inline functions that meet certain size tests without you having to do anything in code.

5. Avoid writing inline assembler.

gcc and **g++** both have the `asm()` construct, which permits you to write inline assembler instructions within its bounds.

The drawback to using inline assembler is that it impedes optimization. Avoid using it except when absolutely necessary.

6. Use internal linkage instead of enter/back or external linkage whenever possible.

A function call between members of the same shared object library occurs without interference from the operating system, which results in the shortest path length during a control transfer.

External function calls (calls on external labels between members of different shared object libraries) call enter/back indirectly.

Enter/back linkage involves system intervention whether it is used directly or indirectly.

Use either enter/back or external linkage if you must; but consider your alternatives.

7. Pass values as function arguments instead of referring to file-scoped, static, or extern storage whenever possible.

The goal is to keep as many data objects on the stack or in the heap as possible. As previously explained, using data objects outside of these storage areas risks unnecessary copy-on-write activity.

8. Avoid writing functions that have more than 5 arguments.

The *zSeries ELF Application Binary Interface Supplement* document states that up to five integer arguments are passed from the calling function to the called function in general registers 2-6. If there are six or more arguments, additional arguments are saved in storage, which impedes high performance.

9. When writing assembly language subroutines for C/C++ programs, use PRLGC instead of TMSPC as the prolog macro.

The `TMSPC` macro has a TPF 4.1 parameter passing interface, which means the macro allocates storage for and stores the register-resident arguments into a simulated Type-1 parameter list. This allows you to migrate existing assembler programs of this type more

easily. However, this parameter-handling scheme always causes extra path length. The PRLGC macro respects the ELF ABI use of R2-R6, which avoids excess instructions.

Considerations for C++ only:

1. **Use C++ only when your needs require its features.**

Compilation in C++ generates larger code and has certain runtime overhead that makes C++ marginally more expensive than C in terms of performance.

Code in C++ where appropriate; otherwise, code in C.

2. **A `struct` and a `class` are the same things in C++.**

Even a `struct` has default constructors (instantiation and copy), a default assignment operator, and a default destructor. Like a `class`, a `struct` might have member functions or virtual functions (or both). Declaration of a `const` data member has initialization constraints that do not exist in C.

3. **C++ support is at the 1998 ISO standard level. Treat namespaces as mandatory.**

The `std::` namespace exists in all the standard C++ headers and **must** be referenced with a `using` statement if you plan to use its identifiers without redundant scope qualification. The objective is to keep the global namespace uncluttered. In the spirit of this goal, partition your user-defined classes and data objects into uniquely-identified namespaces, too.

4. **Prefer the C language `stdio.h` family to the C++ stream classes derived from `ios`.**

These classes (`fstream`, `istream`, `ostream`, `iostream`, and relatives) multiply inherit in both lateral and vertical directions; they call all required constructors and destructors at the appropriate times. Their C language counterparts, such as the `printf` function, can usually perform the same job at a much lower cost.

5. **Beware of temporary objects.**

The C++ compiler generates *temporary objects* when it needs to preserve intermediate results (or *rvalues*), most commonly in the resolution of an arithmetic-style expression that consists of more than one operation. Certain types of conversion operations and initializations also can cause the compiler to generate temporaries.

This is not serious when the operation involves plain old data types (such as `int`, `double`, `char`, and similar); but can impact performance when user-defined data types are the operands.

A temporary object impacts performance because the constructor and destructor of the object must be run during the evaluation of the expression. In addition, the compiler might generate code to run a copy constructor if the particular operation requires it.

Just to make all this a little worse, the compiler can allocate temporary objects from the

default source (usually the heap) used by the *allocator* method of the class, which is more overhead than you might notice as you write that innocent-looking expression.

Sometimes there is no alternative to creating an rvalue. However, if you are designing a class with overloaded operators and conversion functions, you often can handle the operation and assignment at the same time, which can prevent the compiler from generating temporaries.

6. *Hide inline class methods.*

g++ treats function members of a class (methods) as inline when their definition appears in the context of their enclosing class declaration and optimization levels permit inlining.

C++ language rules require that the compiler generates external labels for such methods, even when they are unnecessary. The use of templates amplifies this behavior.

Use the `-fvisibility-inlines-hidden` command line option to prevent the compiler from generating these externals. This reduces module size and saves useless relocations from happening when the module is loaded.

7. *Improve your knowledge of C++ at every opportunity.*

C++ is not a simple programming language that is useful for every circumstance. It has many features that have subtle but iron-clad meanings. Troubles arise when programmers attempt to violate those meanings.

There are many choices available to you to solve all kinds of problems; knowing which to choose can mean the difference between a poorly performing program and one that works efficiently.

We strongly suggest that you read journals, books, and other publications designed to extend your understanding of the C++ programming language.

Trademarks

IBM, the IBM logo, and `ibm.com` are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at <http://www.ibm.com/legal/copytrade.shtml>.

Other company, product, or service names may be trademarks or service marks of others.

© **Copyright IBM Corporation 2010. All rights reserved.**

U.S. Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp. IBM Web site pages might contain other proprietary notices and copyright information that should be observed.