# **Safe ICF**: Pointer Safe and Unwinding aware Identical Code Folding in the Gold Linker

Sriraman Tallam
*Google Inc.*
`tmsriram@google.com`

Cary Coutant
*Google Inc.*
`ccoutant@google.com`

Ian Lance Taylor
*Google Inc.*
`iant@google.com`

Xinliang (David) Li
*Google Inc.*
`davidxl@google.com`

Chris Demetriou
*Google Inc.*
`cgd@google.com`

## Abstract

We have found that large C++ applications and shared libraries tend to have many functions whose code is identical with another function. As much as 10% of the code could theoretically be eliminated by merging such identical functions into a single copy. This optimization, Identical Code Folding (ICF), has been implemented in the gold [4] linker. At link time, ICF detects functions with identical object code and merges them into a single copy. ICF can be unsafe, however, as it can change the run-time behaviour of code that relies on each function having a unique address. To address this, ICF can be used in a safe mode where it identifies and folds functions whose addresses are guaranteed not to have been used in comparison operations.

Further, profiling and debugging binaries with merged functions can be confusing, as the PC values of merged functions cannot be always disambiguated to point to the correct function. To address this, we propose a new call table format for the DWARF debugging information to allow tools like the debugger and profiler to disambiguate PC values of merged functions correctly by examining the call chain.

Detailed experiments on the x86 platform show that ICF can reduce the text size of a selection of Google binaries, whose average text size is 64 MB, by about 6%. Also, the code size savings of ICF with the safe option is almost as good as the code savings obtained without the safe option. Further, experiments also show that the run-time performance of the optimized binaries on the x86 platform does not change.

## 1 Motivation

We have found that identical code is particularly common in C++ programs with heavy use of templates. Figure 1 shows an example. Here, the template class *Foo* is instantiated with different types and each of them gets a separate copy of the *getElement* function. However, since the size of the different pointer types is the same, the bodies of the different *getElement* functions are identical and can be merged into one instance. Using ICF on this example results in these functions being merged into one function and the output of *nm* shows the various *getElement* functions mapped to the same address.

The rest of the paper is organized as follows. Section 2 discusses the Identical Code Folding algorithm. Section 3 talks about the extensions to ICF to make it safe for pointer comparisons. Section 4 is about debugging binaries with folded functions and Section 5 contains the experimental data showing the effectiveness of ICF in reducing the code size of binaries. Section 6 talks about merging identical data members and Section 7 concludes the paper.

## 2 The Identical Code Folding Algorithm

In this section, we discuss how ICF detects identical functions. In order for the linker to perform ICF, the compiler must place each function in a separate section, which can be done in the *GCC* compiler with the flag *-ffunction-sections*. Now, let us define identical functions. From the point of view of the linker, a function has a *text* section and *relocations* that are applied to the text section. Two function sections are identical if and

Figure 1: Motivating Example.

```
template <typename T>
class Foo
{
   ...
   T element;
 public:
   ...
   T getElement ()
   {
     return element;
   }
   ...
};

int main ()
{
  ...
  Foo<int *> p;
  Foo<float *> q;
  Foo<void *> r;
  ...
}

Output of nm :

400432 W Foo<float*>::getElement()
400432 W Foo<int*>::getElement()
400432 W Foo<void*>::getElement()
```

only if their *text* is bit-identical and their *relocations* point to sections that are identical. That is, either the relocations point to the same section or they point to different function sections that are determined to be identical. Figure 2 shows an example where functions *foo* and *bar* are identical because their text is bit-identical and their relocations, to *zip* and *zap* respectively, are identical.

In order to detect such identical functions we do the following. We first split the contents of each function section into two parts, *constant* and *variable*. The constant part refers to the contents that will not change throughout our analysis. These are the *text* content and the *relocations* that do not point to function sections that are our folding candidates. The variable part refers to the relo-

Figure 2: Examples of identical functions - foo and bar are identical because zip and zap are identical.

```
int foo ()
{
  return zip ();
}

int bar ()
{
  return zap ();
}

int zip ()
{
  return 0;
}

int zap ()
{
  return 0;
}

What the linker sees :

Disassembly of section .text._Z3foov:

0000000000000000 <_Z3foov>:
   55                push   %rbp
   48 89 e5          mov    %rsp,%rbp
   e8 00 00 00 00    callq  9
     R_X86_64_PC32 relocation to zip
   c9                leaveq
   c3                retq

Disassembly of section .text._Z3barv:

0000000000000000 <_Z3barv>:
   55                push   %rbp
   48 89 e5          mov    %rsp,%rbp
   e8 00 00 00 00    callq  9
     R_X86_64_PC32 relocation to zap
   c9                leaveq
   c3                retq
```

cations that point to function sections that will be considered for folding. Such relocations will be referred to as variable relocations. Our analysis will form groups of function sections such that all function sections in a group are identical to each other. When we compute the contents of a function, we take the constant part as is and substitute every variable relocation with its group identifier which denotes the group of the function section pointed to by the relocation. Then, we checksum the contents and divide the functions into different groups based on the checksum. Now, we repeat the same steps using the new group identifiers and continue until convergence is obtained. Notice that when we repeat these steps only the variable relocations have to be recomputed. This procedure continues until the group identifier of every function section does not change from the previous iteration. Figure 3 summarizes the steps.

Figure 3: The ICF Algorithm overview.

1. Process each function section. Separate contents into constant and variable parts.
2. Pre-process and find functions that are folding candidates.
3. Initialize the group identifiers of all functions that are candidates for folding.
4. For a function that is a folding candidate, replace the variable relocations of the function with the corresponding group identifiers.
5. Compute the function checksum.
6. Determine the new group id of the function (Look up a hash table mapping function checksum to group id).
7. Repeat steps 4 to 6 for every function that is a folding candidate until convergence.
8. Keep just one copy of function from each group and discard the rest.
9. Map symbols corresponding to duplicate functions to the appropriate kept function symbol.

After the functions have been split into groups, we only retain one candidate in each group, called the *kept* function, for the final binary and discard all the other copies. We then map the symbols corresponding to the duplicate functions to have the same value as the symbol of the kept function.

## 2.1 Initialization step

Let us look in more detail at the third step of the algorithm in Figure 3 where we initialize the group identifiers of all the candidate functions. We have two initialization choices :

1. Pessimistic - Each function is in a unique group (no duplicates).
2. Optimistic - All functions are in the same group (all functions are identical to each other).

If we conservatively initialize each function to be in a separate group then after each iteration the decisions made regarding functions that are identical are guaranteed to be correct. This has the advantage that the algorithm does not necessarily have to be run until convergence is obtained, although some opportunities may be lost if it is stopped early. However, identical functions with recursive calls or mutually recursive calls will not be detected because of the initialization. Figure 4 shows an example. The functions *funcA* and *funcB* are identical but will not be detected if we start by assuming that all functions are unique. On the other hand, if we aggressively initialize all functions to be identical, we will capture the recursive and mutually recursive cases but the algorithm has to be run to convergence as correctness is not guaranteed if we arbitrarily stop it.

For our implementation, we have used the pessimistic approach, that is, we initialize by placing each function in a separate group. We handle recursive calls by detecting and replacing it with a special symbol. Also, we did not find any mutually recursive calls in the benchmarks we used for our experiments. Further, we found that in all of our benchmarks, this approach leads to convergence in 3 iterations whereas with the optimistic strategy, we needed 5 to 6 iterations. We set the default number of iterations in our implementation to two as the third iteration merely checks for convergence.

Figure 4: Mutually recursive identical functions - funcA and funcB are identical.

```
int funcA (int a)
{
  if (a == 1)
    return 1;
  return 1 + funcB(a - 1);
}

int funcB (int a)
{
  if (a == 1)
    return 1;
  return 1 + funcA(a - 1);
}
```

Figure 5: ICF unsafe in the presence of function pointer comparisons.

```
int foo ()
{
  return 0;
}

int bar ()
{
  return 0;
}

int main ()
{
  assert (foo != bar);
}
```

### 2.2 Pre-processing for performance

We do some pre-processing to reduce the number of function sections to be analyzed. Before we begin the main algorithm, we find and eliminate functions which have unique static content from being considered for folding. Also, any group of sections found identical in the pre-processing step that does not have any variable relocations is finalized and not considered for further analysis.

### 2.3 Merge sections

Merge sections hold read-only constants and the gold linker treats each merge section as a list of constants, and merges them all into a list of unique constants. However, the ICF analysis happens before the linker has merged identical sections. Hence, we inline the referenced constant at places where it is referenced in the function contents while computing the function checksum so that this opportunity is not missed.

### 2.4 Choice of checksumming method

Our implementation of ICF uses the crc32 algorithm [5] to compute the function checksum. Using crc32 gives rise to a number of hash collisions and so we use a multi-map hash table to map the function checksum to group id. Further, before inserting a function into a hash group, we do a bit-wise comparison of the function contents with that of the kept function in the group. We explored the alternative of using a more robust checksumming method like md5sum [3]. However, computing md5sums was found to be much slower than computing crc32 checksums.

## 3 Safe Identical Code Folding

ICF in general can be unsafe if the binary uses the function pointers of merged functions in comparison operations. Figure 5 shows an example to illustrate this. Function *foo* and *bar* are identical and folded and the assertion in function *main* no longer holds. We found a real example in one of our benchmarks which was doing function pointer comparisons and its execution crashed, fortunately, when it was linked using ICF. This helped us in notice the problem and be able to develop a solution.

In order to guarantee run-time safety, we have implemented a safety option to detect functions whose pointers are accessed and prevent them from being folded. This is done as follows. Usually, the relocation type for a function call differs from that of a function address access. Our method inspects the relocation type to identify the functions whose addresses are accessed and marks such functions as not foldable. Although not all function address accesses are used in comparisons, we conservatively assume the worst in order to ensure correctness. Depending on the build combinations, the target architecture and to some extent on the compiler used to generate the code, the relocation types may vary and we account for all cases. In the cases where are not sure, we conservatively assume that the function pointer is accessed. As an example, the following are the different cases when linking for 32-bit *x86*.

1. For executables and shared libraries built with non-pic and non-pie objects, a function call relocation type is always *R_386_PC32* and a function address access relocation is always of type *R_386_32*.
2. For executables and shared libraries built with pie objects, a function call relocation type is always *R_386_PC32* and a function address access relocation is always of type *R_386_GOTOFF*.
3. For executables and shared libraries built with pic objects, a function call relocation type is always *R_386_PLT32* and a function address access relocation is always of type *R_386_GOT32*.

### 3.1 Always fold constructors and destructors

In C++, accessing function pointers of constructors and destructors is forbidden. Hence, Safe ICF will always consider any function that is a constructor or a destructor as a folding candidate.

### 3.2 Vtable accesses

Function pointers of virtual functions are accessed for vtable purposes and we ignore such accesses as they are not used in a comparison operation that will affect the run-time behaviour. Relocations corresponding to function pointer accesses for the vtable occur in specially named sections, for example, sections with prefix *.rodata._ZTV*. These are detected by our implementation and ignored.

## 4 Unwinding across merged functions

For binaries optimized with ICF, a PC value can no longer be unambiguously mapped to a particular function. This can be expected to cause mysterious behavior when profiling and debugging an application that has been built with this optimization. We address this problem by adding a new table to the DWARF debugging information to allow the debugger and other tools to disambiguate such PC values by examining the call chain.

### 4.1 Overview

When the PC is inside a merged function, we attempt to disambiguate the function by examining the function's return pointer (i.e., the point of call that invoked the merged function). There are four cases.

1. *Direct call*: The point of call is a direct call to the merged function. In this case, we can look up the address of the call site in a direct-call table that provides, for each direct call site in the program, a pointer to the debug information entry for the called function.
2. *Virtual call*: The point of call is a C++ virtual function call. In this case, we can look up the address of the call site in a virtual-call table that provides, for each virtual call site in the program, the index of the vtable slot used for the call. Given the vtable address, which can be obtained from the this pointer in the callee, the debug information entry for the corresponding class can be identified, and the function being called can be determined by matching the slot index to the virtual function members of that class. If the callee's this pointer cannot be determined (e.g., due to optimization of the routine), the vtable slot index might still be used to eliminate some candidate functions.
3. *Other indirect call*: The point of call is an indirect call, but not a virtual function call. This case is not relevant with Safe ICF as this function is not folded since its address has been accessed.
4. *Tail call*: The point of call may have led to an intermediate procedure that made a tail call to reach the current callee. In this case we will not be able to disambiguate the PC.

In order to perform the disambiguation we construct two additional debug information tables in the final executable: a direct-call table and a virtual-call table. Each of these will be generated by the compiler and placed in a new debug section in the relocatable object files. The linker will combine these sections as it normally does to produce a single combined section for each table. For functions that will not be merged together, entries in the direct call table are not needed, and would represent a significant waste of space. At compile time, unfortunately, all functions are candidates for merging and the

Table 1: Code size reduction with ICF.

| Benchmark | Text size of binaries in MB | | | | | |
|---|---|---|---|---|---|---|
| | Improvement over Baseline | | | Improvement over Linker GC | | |
| | Orig. | With ICF | With SafeICF | GC | With ICF | With SafeICF |
| Index search | 96.06 | 89.73 (6.59%) | 89.77 (6.55%) | 68.35 | 66.25 (3.07%) | 66.34 (2.95%) |
| Database | 61.91 | 57.59 (6.97%) | 57.72 (6.77%) | 47.10 | 45.39 (3.61%) | 45.38 (3.64%) |
| Ads | 121.02 | 111.99 (7.46%) | 112.14 (7.34%) | 85.44 | 82.53 (3.40%) | 82.46 (3.49%) |
| Image Processing - I | 77.84 | 73.96 (4.99%) | 74.09 (4.82%) | 55.67 | 54.33 (2.41%) | 54.38 (2.31%) |
| Image Processing - II | 43.72 | 41.14 (5.89%) | 41.19 (5.78%) | 26.93 | 26.16 (2.87%) | 26.18 (2.81%) |
| File system | 21.11 | 19.79 (6.23%) | 19.83 (6.04%) | 15.47 | 14.99 (3.12%) | 14.99 (3.11%) |
| Web search | 106.61 | 99.79 (6.39%) | 99.85 (6.34%) | 80.86 | 78.31 (3.15%) | 78.46 (2.96%) |
| NLP | 145.41 | 137.01 (5.77%) | 137.34 (5.55%) | 115.31 | 112.13 (2.76%) | 112.35 (2.57%) |
| Text detection | 25.10 | 23.67 (5.68%) | 23.71 (5.55%) | 16.74 | 16.29 (2.72%) | 16.31 (2.58%) |
| Serialization | 60.13 | 55.46 (7.76%) | 55.49 (7.72%) | 42.15 | 40.36 (4.24%) | 40.36 (4.25%) |
| Clustering | 42.49 | 40.39 (4.95%) | 40.44 (4.83%) | 30.12 | 29.45 (2.22%) | 29.50 (2.06%) |
| Map reduce | 93.59 | 87.47 (6.54%) | 87.53 (6.47%) | 71.30 | 69.12 (3.06%) | 69.25 (2.87%) |
| Geo. Mean | 64.10 | 60.08 (6.27%) | 60.15 (6.16%) | 46.02 | 44.61 (3.06%) | 44.65 (2.98%) |

duplicate functions have not yet been identified, so the compiler must generate call table entries for all direct and virtual calls. In order to reduce the total size of the direct call table, therefore, the linker can discard any call table entry that refers to a function that has not been merged with another.

For more details, please refer to the description in the dwarf wiki [2].

## 5 Experimental Evaluation

We have implemented ICF in the gold linker. It is available with linker option *–icf=all*. Safe ICF can be turned on with option *–icf=safe*. We conducted experiments to measure the code size reductions obtained with ICF and Safe ICF. We have used a set of benchmarks representative of Google work loads. In particular, we measured the effectiveness of ICF in reducing the text size of the binaries. Table 1 shows the results of our experiments. The mean text size of binaries we considered is about 64 MB. We first measured the effectiveness of ICF and Safe ICF in reducing the text size of the original binary. We then measured the effectiveness of ICF and Safe ICF when applied along with linker garbage collection (*GC*), which shrinks the binaries by removing functions that are not referenced.

Our experiments show that ICF can reduce the code size of the original binaries from 4.95% to 7.76%. Also, ICF can further reduce the code size of binaries from 2.22% to 4.24% over linker garbage collection. Further, our experiments show that Safe ICF is almost as effective as ICF, about 97%, in reducing the code size of binaries.

Finally, our experiments showed that ICF has no measurable impact on the run-time performance of these binaries on the x86 platform, both 32 and 64-bit.

## 6 Detecting identical read-only data sections

We could get even more code size savings by extending this work to also detect and fold identical read-only data sections. Gold already supports merging of read-only string constants and folding other sections is in progress.

## 7 Related work

Microsoft's Visual Studio product [1] offers a compiler option, /Gy, that directs the compiler to place the object code for each function in a separate COMDAT section. This can be used with a linker option, /OPT:ICF ("Identical COMDAT Folding"), that directs the linker to detect duplicate instances of identical COMDAT sections and remove the redundant copies. All the original function symbols are then set to the address of the one remaining copy. Microsoft advises users to not use this feature if the execution of the program depends on the addresses of the functions and data members being different. Microsoft also advises its users not to use this option when profiling or debugging an application, as there is no support for disambiguating a PC value that may belong to any of several different functions that were folded into one copy.

## 8 Conclusion

We have described the implementation of Identical Code Folding in the gold linker and have shown that it can be very effective in reducing the code size of binaries. Also, our implementation can guarantee execution correctness in the presence of function pointer comparisons and also allow unwinding across merged functions for tools like debuggers and profilers.

## 9 Acknowledgements.

We would like to thank the compiler team at Google for their contributions towards the successful completion of this work.

## References

[1] Microsoft visual studio 2010, 2010. `http://msdn.microsoft.com/en-us/library/bxwfs976.aspx`.

[2] Cary Coutant. Dwarf extensions for unwinding across merged functions, 2009. `http://wiki.dwarfstd.org/index.php?title=ICF`.

[3] R. Rivest. The md5 message-digest algorithm, 1992. `http://tools.ietf.org/html/rfc1321`.

[4] Ian Lance Taylor. A New ELF Linker. In *Proceedings of the GCC Developers Summit*, pages 129–136, Ottawa, Canada, June 2008.

[5] Ross N. Williams. A painless guide to crc error detection algorithms, 1993. `http://www.ross.net/crc/download/crc_v3.txt`.