Analyzing Assembler To Eliminate Dead Functions: An Industrial Experience

Ian J. Davis, Michael W. Godfrey, Richard C. Holt
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
{ijdavis, migod, holt}@uwaterloo.ca

Serge Mankovskii and Nick Minchenko
CA Labs, CA Technologies
{Serge.Mankovskii, Nick.Minchenko}@ca.com

Abstract — Industrial software systems often contain fragments of code that are *vestigial*; that is, they were created long ago for a specific purpose but are no longer useful within the current design of the system. In this work, we describe how we have adapted some research tools to remove such code; we use a hybrid static analysis approach of both source code and assembler to construct a model of the system, and then use graph querying to detect possible dead functions. Suspected dead functions are then commented out of the source. The system is then rebuilt and run against existing test suites to verify that the removals do not affect the semantics of the system. Finally, we discuss the results of performing this technique on a large and long-lived industrial software system as well as a large open source system.

Keywords— Reverse engineering, static analysis, maintenance, program and system comprehension

I. Introduction

Successful software systems typically evolve over time. As consequence large long-lived systems tend to become poorly understood, as the design erodes and knowledge about the system is lost. Any tools that can assist in rapidly comprehending aspects of these systems may prove to be very valuable.

Absent some understanding of the context in which logic is supposed to be invoked, it is often hard to comprehend the purpose of code being examined. It is very frustrating to spend considerable time attempting to discover this context before reaching the conclusion that source code is never executed. Since it is never executed it is never tested, irrespective of the number of tests associated with it. And since it is never actually tested, there is no guarantee that this code actually works. An engineer thus risks cloning [1] bad code within a stable system they trust as a foundation for what they are about to undertake.

At the request of industry partners at CA Technologies, we were invited to explore how, given a large C++ system, one might discover functions that are not reachable at execution time, so that this redundant logic might be identified, reviewed, and potentially removed [2][3]. Such a tool would permit improvements to be made in existing legacy code at minimal cost, and would be an asset in evaluating the amount of genuinely useful logic in a large unfamiliar source code base. It also allows an engineering organization to be more efficient in their day to day jobs.

In this paper we concentrate on the problem of how to identify functions that cannot logically be invoked directly or indirectly from the function mainline, from static initialization of variables, from dynamic local variable construction, or though other language mechanisms such as polymorphism and the throwing of objects. A supplementary problem is how to then automate the removal of such functions from the source code, when source code can be used in many different configurations, and potentially even across different projects.

Armed with ground truth about dependency information, dead code analysis is straightforward; thus, the central issue here is how best to approximate ground truth from arbitrary source code. Because it is challenging to correctly parse and interpret the nuances of modern source code languages, we tailor the build process in order to discover precisely how compilers build the system from the totality of the source code, and then, using this knowledge, we repeat the compilation of all source, modifying the compilation parameters so that the corresponding assembler is obtained. Fact extraction is then performed on this collective assembler, with the results then presented as if they had been discovered within the original source code.

Our approach has the advantage that the input operated upon by our fact extractor is simple to parse and is known to be a genuine representation of the source code as used in the actual build process, accurately reflecting the results of files included, macros employed, data types used, etc. This input can readily be derived from source code written in various languages without any need to develop, support, and maintain different language parsers for each such language. It also cleanly permits extraction to be performed on source code that contains embedded assembler, or on assembler (potentially generated by a tool as part of the build process) for which no corresponding high level source code is available.

The rest of the paper is structured as follows: Section 2 provides a general overview of fact extraction. Sections 3 to 6 address specific C++ issues of relevance to dead code elimination. Section 7 presents an industry case study. Finally, in Section 8 we summarize our contributions.

II. FACT EXTRACTION

In our approach the task of converting C and C++ code [4] to assembler is achieved by "wrapping" the compiler invocations inside scripts that are transparently invoked as consequence of being discovered earlier on the program directory search path. These scripts capture the directory each compilation occurs within, the compiler invoked and the parameters passed to this compiler in a log. Once captured this log is then presented as input to the fact extraction tool [5] which repeats sequentially each of the compilation steps performed, altering parameters appropriately, and after each compilation reads as input data the assembler produced by the compiler from the corresponding source. Fact extraction recovers all of the information in the assembler which is relevant to dead code analysis [6][7], consolidating this information into a graph consisting of typed nodes, typed directed edges, and attributes name/value pairs associated with both.

The sheer quantity of facts thus recovered can be intimidating. These facts are recovered from post-processed code, through a mechanism where all of the internals of such things as macro expansion, templates, use of the C++ Standard Template Libraries (STL), initialization of variables buried within such libraries, and the internal structures needed to support language concepts such as objects, classes, polymorphism, etc. are all exposed. This however is one of the advantages of performing fact extraction on the assembler; the hidden complexity buried in the source code is exposed for all to see.

There remain of course facts present in the source not present in the assembler. For example, "friend" relationships might be present in the source code, but these cannot be discovered in the assembler, if trust is entirely enforced by the compiler itself. This has some unexpected ramifications for dead code removal. One might by analyzing assembler conclude that entire classes could be removed from source code, only to discover having removed these classes that the code no longer compiled because of friend references to these now no longer declared classes. But such caveats aside, it is reasonable to argue that if no explicit usage is made of something in the assembler it isn't required in order to build the system produced from this assembler.

III. POLYMORPHISM

Virtual functions may be invoked directly by name or indirectly via pointers in the compiler generated v-table associated with a class. When invoked indirectly, we must be careful to discover the assembler instructions that allow us to infer that a known function with a given signature is being called. This pattern typically involves:

1. *Load a pointer associated with a class into a register.*
2. *Move the class v-table pointer to a register.*
3. *Adjust this v-table pointer to a function pointer offset.*
4. *Move the virtual function pointer at offset to a register.*
5. *Call the function indirectly via the register.*

In practice compilers achieve the above operations using a variety of different registers, and may interleave other code with the above sequence of operations, provided that the interleaved code does not alter the value in the currently relevant register. The actual assembler instructions vary depending on whether a 32 or 64 bit pointer architecture is being used, as do the registers used to store pointers.

Discovering that a pointer in a register addresses a class can be achieved by the following process: the DWARF symbolic debugging information [8] is obtained from the assembler, by translating the assembler describing this structure into its internal quasi-hierarchical data representation. Whenever the assembler loads a pointer into a register, either as an atomic variable, or as a member variable of a structure, the type of this pointer is determined by examining this symbolic information, just as a debugger would do. Pointer dereferences must also be handled since these change the type of item contained in a register. Collectively, this allows us to statically determine when a register contains a pointer to a class.

Since we are performing static analysis situations can arise where we are unable to infer the type a pointer. We cannot be certain of type information immediately following any assembler label, since this might be branched to. In theory we might also be misled if the original pointer used to discover the polymorphic function invoked was not itself declared to be a pointer to a pointer to a vtable, but such code would be somewhat irregular.

IV. INHERITANCE

When fact extraction identifies polymorphic function invocation, it merely indicates that a named virtual function in a named class is being invoked indirectly. Dead code analysis augments this information to create additional call edges between the invoking agent, and the matching polymorphic function in every subclass of the named class. This is achieved by exploiting other information provided by fact extraction.

For each virtual function, having at least one virtual call to it, the class it belongs to is recovered from the mangled function's signature [9]. Then for each reference to the function's address it is determined if this reference occurs within a structure identifiable as a v-table. Such g++ compiler generated structures are assigned a mangled name that begins with the prefix *"_ZTV*". This is followed by the class name that the v-table belongs which permits us to recover the unique v-table addressing this function which is associated with the function's class present in this function's signature.

In addition to containing pointers to functions , each v-table also contains a pointer to a type information structure associated with the class. This structure can be identified because its mangled name has the prefix *"_ZTI*". The type information structures for classes that directly inherit from one or more multiply inherited classes, include within them pointers to super-type information structures, and this information is present in the facts extracted.

This permits enough navigation within the facts extracted to locate the v-tables that may potentially be accessed when polymorphic functions are invoked. It is then a simple matter to match the signatures of functions addressed by these v-tables, in order to deduce the set of functions potentially visited by a polymorphic call.

V. INCLUDE FILES

While C/C++ function bodies typically occur in "implementation" source code files they may also appear in header (i.e., `.h`) files. When such functions are declared to be static, or in lined within the body of classes this may result in multiple distinct functions in the assembler, some of which are invoked, and some of which are potentially not invoked. Similar problems may arise when C++ templates are used, either within the standard template library, or in other contexts.

To avoid this problem, all functions with the same signatures that are declared in the same file, and occur at the same line number within that file, are consolidated into a single node within the fact extraction graph, by migrating all incoming and outgoing edges from all duplicate functions to this arbitrarily chosen single node, followed by removal of the duplicated functions. This ensures that if a function in a header file is used in any source code, it is not subsequently presumed to be dead in other source code in which it is not called. In addition we are careful to avoid altering system header files.

VI. DEAD CODE REMOVAL

Because dead code should, in our opinion, remain accessible to programmers, the safest mechanism for automating removal of dead code [10] in a manner which can be readily undone is to surround this dead code with an **#ifdef** that causes it to be ignored by the compiler, but not by the human reader.

Because the same source code can be used in many different build configurations care is taken in how such **#ifdef's** are inserted. As earlier noted, fact extraction is performed with respect to a specific build, and all of the build parameters associated with the compilation of source can thus be included within the facts operated on by dead code analysis.

So suppose that a source file main.cpp containing unreachable functions is compiled using the arguments:

```
g++ -Dlinux –Uwin32 –Dversion=2 main.cpp
```

We will add to the start of main.cpp the following preamble:

```
#if !defined(_DEAD_) &&
    defined(linux) && !defined(win32) &&
    defined(version) && (version == 2)
#define _DEAD_20110312203041
#endif /* _DEAD_ */
```

and bracket unreachable functions such as foo() thus:

```
#ifndef _DEAD_20110312203041 /* Mar 12 15:30:41 2011 */
int foo() {}
#endif /* _DEAD_20110312203041 Mar 12 15:30:41 2011 */
```

This mechanism permits all dead code to be made visible to the compiler by simply defining _DEAD_; it ensures that to the extent possible[1] source code is only hidden from the compiler when using the same configuration information that fact extraction was earlier performed on; it permits dead code analysis to be performed on source code multiple times, under the same or different configuration options; and clearly documents when and why dead code elimination has occurred.

[1] cpp expressions cannot test if a preprocess variable matches a string

In cases where the same source code is compiled into different projects, it is only necessary to ensure that each project is configured with at least one distinctly defined preprocessor variable in order to continue to accommodate distinct usage of source code.

The remaining challenge is to determine where to physically insert the **#ifdef** and **#endif** statement that are to bracket each discovered dead function. The solution to this challenge is to again exploit the compiler. We know how the source was compiled, and are thus in a position to recompile it. If we recompile this source code by using the '-E' option, gcc and g++ will emit to the standard output the post-processed source that the compiler sees. This post-processed source includes the provenance of which file and which line number in original source generated the resulting output, primarily so that error messages can refer back to the original source, and so that this same information can be inserted into the assembler. By examining the post-processed source rather than the source to be updated we avoid encountering any comments, macros, #ifdefs, or #include statements that would otherwise make parsing the source code problematic for us.

Having discovered where functions start and end in the post processed source, we can then infer the file, line and character position where we wish to insert our #ifdef statements in the original source. Character positions must be adjusted to account for comments present in the original source which are removed by the preprocessor. Expansion of tabs by the preprocessor that would also complicate character positioning mapping between pre and post processed source, is avoided by using the *-ftabstop=1* option.

We may be still unable to correctly remove dead functions in all source code. Dead default class constructors and destructors may not even being declared in the source, and even if they are we may not wish to remove them, because subsequent changes to source code that invoke such removed functions will produce no warning that the function the programmer intends to invoke has been removed. This is just as much a concern when removing virtual functions that override other virtual functions.

Further problems can arise when removing dead virtual functions. If they inherit from pure virtual functions, their removal transforms the class containing them into an abstract class that cannot be instantiated. And if virtual functions that directly inherit from such removed virtual functions fail to explicitly declare that they are also virtual then, following removal of functions they inherit from, they cease to be virtual.

In some cases it is not sufficient to remove function definitions that are not invoked. These functions may also be declared to be members of a class. While class functions are permitted to be declared, but never defined, providing that they are never invoked, this is *not* the case with virtual class functions, since the linker wishes to place the address of such functions in one or more v-tables, even if they are never actually invoked. We currently have no mechanism for automating removal of virtual function declarations, when these declarations do not also contain an in lined definition of the functions body. Automated removal of virtual functions will produce linkage errors, which must currently be addressed manually in order to remove the corresponding virtual function declaration.

Macro expansion is also problematic. If one macro expands into the definition of two functions, one of which is dead, we will be advised that the dead function occurs not within a macro, but at the line where the macro is employed within the code. We do not wish to comment out the macro usage in the original source, since it defines a function that is still invoked, but we have no mechanism to re-engineer the internals of macro definitions. Such actions will continue to require human intervention. Automated dead code removal is an end user convenience, but not a panacea.

To the extent that programming is an art, and not a science, there are also deep philosophical questions as to when it is and is not appropriate to remove unreachable functions from source code. Instances of a class not constructed today might easily be employed in the source code of tomorrow, and an unreachable polymorphic function that generates an error message should it be invoked is an integral part of a defensive programmer's arsenal. Destructors which are never invoked should probably remain active in the code, because it is unreasonable to cause a future delete to fail, by undermining the conceptual notion that what a constructor creates, a destructor destroys. And any unreachable function not removed from the source code, requires that all of the code reachable from it, even if from nowhere else must also remain in the source. So the decision to preserve some unreachable functions, may conflict with the desire to remove others.

VII. PROOF OF CONCEPT

CA provided us with a large corpus of source code, associated with one of their product lines. We identified the largest executable within this product family, and attempted to perform dead code analysis on the 27,460 lines of C++ source code contained in the 50 files compiled when building this executable. Because our fact extraction tool operates only on Linux platforms, we conducted our analysis on this platform.

We discovered 27 dead functions within the code provided us. These functions consisted of 270 lines of code distributed across 13 files. A further 16 functions in 4 header files were discovered to not be invoked, but were ignored because these files lay outside the scope of the source code being studied. A total of 9 constructors and 39 destructors were never invoked. The removal of the 270 lines of code was fully automated, and the code subsequently rebuilt without any manual intervention.

As a second proof of concept we downloaded the latest development version (1.7.6.rc2.dirty) of GIT [11], and performed dead code elimination on this source. This software comprised 112,000 lines of C source in 217 files. We discovered 43 dead functions, and automatically removed 699 lines of source from 16 source files. Again after automatic removal of the dead code the source continued to compile without errors.

VIII. CONCLUSIONS

We have at the request of industry partners explored the issue of dead code analysis, and developed a publically available tool [12] that permits functions that are never invoked to be discovered, and optionally removed from the body of software to be built. We have built our approach around existing and improved research tools, using a variety of static analysis techniques, operating on a graph based model of the code. We have explored a variety of issues in creating such a tool, some peculiar to C++, and shown how many of them can be overcome by relatively straightforward techniques.

Removal of source code identified by our fact extraction tools as never invoked, followed by attempts to build of the modified system, was a very good way of testing that relationships not present in the facts extracted by us, also were not present in the source. Of course within industry extensive testing would occur before concluding that dead code could indeed be safely removed from source code.

A future challenge is to extend our tool to permit it to be used with libraries and interfaces having multiple entry points.

In summary this research has proven challenging, but also interesting in exploring an industrial problem using existing research tools.

ACKNOWLEDGMENT

This research is supported by grants from CA Technologies Canada Inc., and NSERC.

REFERENCES

[1] I. J.Davis, M. W. Godfrey. "From Whence It Came: Detecting Source Code Clones by Analyzing Assembler". 17th Working Conf. on Reverse Engineering (WCRE), October 2010.

[2] J. Knoop, O. Rüthing and B. Steffen, "Partial dead code elimination," Proc. ACM SIGPLAN PLDI, pp.147–158, June 1994.

[3] P. Briggs, R. Shillingburg and T. Simpsonl, "Dead code elimination," www.cs.princeton.edu/~ras/**dead**.ps, October 1993.

[4] GCC, the GNU Compiler Collection. http://gcc.gnu.org

[5] SWAG: Software Architecture Group. ASX C/C++ Assembler Fact Extractor. www.swag.uwaterloo.ca/asx

[6] Y. Chen, E. R. Gansner and E. Koutsofios K. Elissa, "A C++ data model supporting reachability analysis and dead code detection," IEEE Transactions on Software Engineering, Vol 24 No. 9, September 1998.

[7] Y. A. Liu and S. D. Stoller, "Eliminating dead code on recursive data," Science of Computer Programming – Special Issue on static analysis. Volume 47 Issue 2-3 May 2003.

[8] DWARF 3.0 Debugging Standard.. http://dwarfstd.org.

[9] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. Itanium C++ ABI. 2001. http://www.codesourcery.com/public/cxx-abi

[10] R. Bodik and R. Gupta, "Partial dead code elimination using slicing transformations" Proc. ACM SIGPLAN PLDI, pp. 159–170. June 1997.

[11] GIT – Fast Version Control System. http://git-scm.com/download

[12] SWAG: Software Architecture Group. XCISE: ASX C/C++ Dead Function Eliminator. www.swag.uwaterloo.ca/xcise