TRICK - A framework for Tracking and ReusIng Compiler's Knowledge

Sandya S.M, Shruti Doval, Hariharan S, Mahesha N, Dibyendu Das

Abstract

Compilers, during compilation, analyze the application being compiled and build up extensive knowledge about the program. This knowledge is essential for the compiler to produce correct and optimized object code. Though some part of this knowledge is retained in the generated object files as symbol table information to be used by the linker and/or debugger, most of it is discarded after the compilation is over. In this paper, we introduce the TRICK framework, which is an attempt to retain and reuse this internal information generated by the compiler as part of its program analysis, for building new tools, enhancing existing tools or reuse by the compiler for continuous program optimization or incremental compilation. We present examples of how development and maintenance of various program analysis tools can be simplified by using the TRICK framework. TRICK framework can be part of both static and dynamic compilation system, though our current usage model is in the context of a static compilation system.

1 Introduction

"We shall not fail or falter; we shall not weaken or tire...Give us the tools and we will finish the job". Sir Winston Churchill.

Just as in other trades, tools form a critical part in software development lifecycle. Program analysis tools play a key role in development, deployment and maintenance of software applications. There are a whole range of analysis tools available today.

These analysis tools can broadly be categorized as

1. Static source code analysis tools which require only the source code. These include cross-referencing tools like Cscope [8], program discovery tools, e.g. tool to construct program dependency graph or a class diagram generator and program verification tools like lint

2. Runtime analysis tools which work on a combination of source code and runtime information that is obtained by running an instrumented executable. These include code coverage tools which measure how well the test cases exercise the application code, test prioritization tools which determine which test cases are important to run for covering the parts of the code that have changed, performance analyzers, runtime verification tools like heap analysis tools, thread analysis tools and synchronization analysis tools like locklint

3. Advanced compiler features like Incremental compilation, refactoring and continuous optimization.

Much of the useful information needed by various analysis tools is available to the compiler during compilation. Some examples of such information are

1. Variable definition and usage information needed by various discovery tools e.g. cscope
2. Class hierarchy information needed by reverse engineering tools
3. Dominator information needed by the code coverage tools to decide the points of instrumentation
4. Machine resource utilization information required by the performance analysis tools

Even though the above information is available to the compiler, especially when compiling at higher levels of optimization, there is no mechanism available to retain this information once the compilation process is over. Such a mechanism would have simplified the task of tool

development. It is possible to achieve the ease of tools development by compiler reuse in two ways. One possibility is to expose the compiler interfaces and the other is to expose the compiler source code itself, so that various tool developers can modify the compiler front-end/ optimizer/ backend and remorph it as an analysis tool. GCC has been very successful example of the latter approach and hence is used widely as a tool developer's underlying framework. Some of the complex tools such as racer X, mpatrol are based on GCC. However this option is not viable for commercial compilers, making it necessary to develop alternate mechanisms for facilitating the development of a healthy ecosystem of tools around them.

In this paper we describe the TRICK framework which retains the information extracted from the source code during various phases of compilation and explain how it can be reused by program analysis tools. The goal of TRICK (Tracking and ReusIng Compilers Knowledge) framework is to make the compilers treasure trove of information even after the compilation process is over so that other tools can utilize that information. The compiler itself can be considered as just another tool that can reuse the information generated by itself as part of a continuous program optimization cycle. We propose capturing all the knowledge that a compiler gains about a program during compilation in a database and making it available to various tools through standard API's or custom queries.

Today many commercial compilers offer value added features like static memory leak detection, race detection, dead-lock detection, potential buffer overflow violations, to name a few. For example, the recent release of HP's C/C++ compiler offers a 'vaccine' feature for static memory leak detection. Though much of this information can be obtained as part of normal compilation and build process, the customization, tracking and maintaining of this information is not a job the compiler is well suited to do. The compiler is the best place to do the analyses required for these features/tools but the application of the results of the analysis is best left outside the compiler to the individual tools. For instance, a call graph is built by the compiler as part of its whole program compilation, but producing an impact analysis report from this call graph is not the compilers work, and it is better done by another tool. Hence it is profitable to retain the information generated by the compiler as part of its complex analyses and allow the reuse of this information by the various tools so that they do not have to redo these time consuming analyses.

Another motivating reason for retaining the results of compiler analyses is the increasing importance of continuous program optimization in commercial compilers. (SPEC 2006 disallows profile based optimizations, but allows dynamic runtime optimizations). If the compiler's static analysis information is retained, it would be of use during the dynamic optimization. From here onwards, when we refer to the tools which can make use of the compiler generated information, we implicitly refer to the reuse of information by the compiler also.

This paper does not propose a new optimization or performance improvement technique. We believe that compiler writers have the power to influence the role compilers will play in SDLC over the years, and this role needs to be broadened, from being a mere source code cruncher to an omnipotent authority of the knowledge about the application source, which can be queried. We are still trying to find answers to questions such as

- What are the compact representations of such compiler generated information so that it can be retained, reused and updated incrementally in an efficient manner?
- What are the various views for presenting the information to the consumers?
- How extensible can such a framework be made?
- How to make the collection of information from the compiler and storing into the Database transparent to the normal build process and do so without slowing down the build?

The subsequent parts of the paper are organized as follows:
Section 2 explains the design of the TRICK framework. In section 3 we do a study of how various program analysis tools can use the TRICK framework and give initial prototype results for TRICK. Section 4 talks about related works which aim at easing the task of tool development. Section 5 concludes the paper giving pointers for the future work.

2 TRICK Design:

TRICK consists of 3 major components as shown in figure 1:

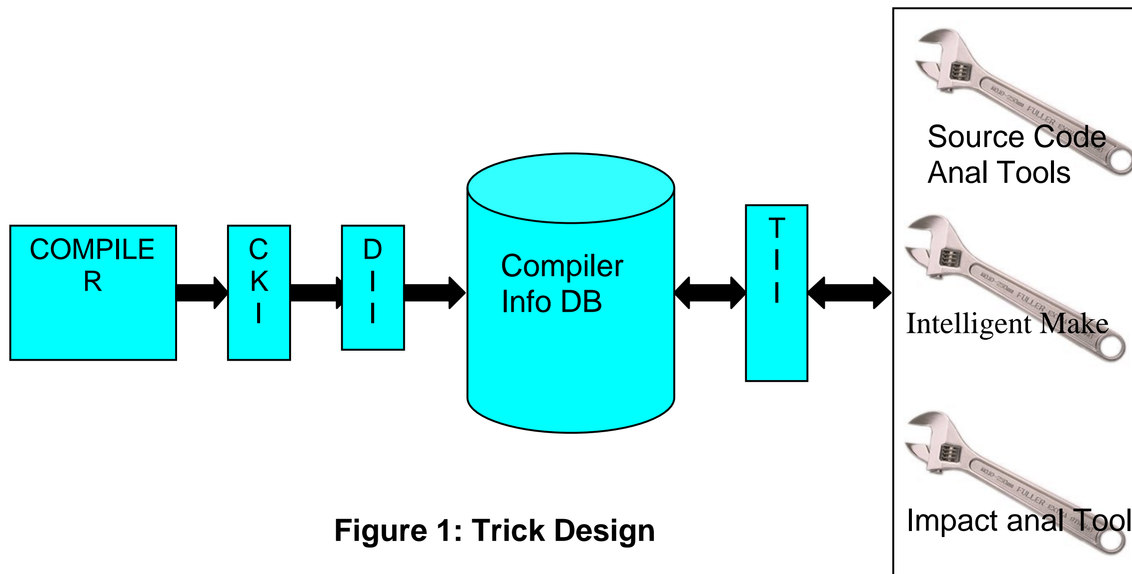

Figure 1: Trick Design

2.1 Compiler Information Interfaces (CII)

These interfaces define how the information flows from the compiler to the database. This is implemented as two layers, namely Compiler Knowledge Interfaces, CKI, and Database Information Interfaces, DII. Compiler knowledge interfaces capture the information generated by the compiler as part of the compilation process. They are defined by TRICK and will need to be implemented by compiler vendors. Implementing the CKI will require the compiler to capture the information generated as part of the compilation process and represent it in a predefined format. A compiler vendor can choose to implement some or all the interfaces defined by TRICK.

Database Information Interfaces process the information outputted by the compiler through the CKI into a compact form to be inserted into the database. The advantage of this two layered approach is that it makes the information outputted by the compiler independent of the database. CKI and DII define an interface for information transfer between the compiler and the CIDB, allowing the compiler to work in conjunction with any 3rd party implementation of the CIDB and DII. Also a compiler vendor can extend the existing set of CKIs by adding a new one and providing a corresponding DII implementation.

2.2 Compiler Information DataBase (CIDB)

CIDB houses different types of information which can be classified according to the phase of compilation in which they are gathered as shown in figure 2.

High level information This includes symbol table information, source dependency information, source position information etc... This is the information that is typically gathered by the compiler

front-ends as part of their syntax and semantic analysis phases and is usually required by various source code analysis tools.

Intermediate level information This is the information that is generated when the compiler operates on the high level intermediate representation of the source code. This information is obtained as part of the analysis phases of the high level optimizer; it typically includes control flow information, static call graph, dominator information, alias information, type information, array access information, local points to information etc. In the case when compiler also does inter-procedural analysis the inter-procedural analysis information such as inter-procedural points to information and side effects analysis results are also retained. In addition to storing the results of various analyses, we can also store the IR representation of the application source.

Low level information This is the architecture dependent information that is generated during the low level optimization phase of the compiler. This includes information such as the types of machine resources required by the application, its data access pattern, register usage etc.

The initial design of the database was a rudimentary design based on the existing pre-compiled headers (PCH) database infrastructure provided by the compiler, the main motivation being the possibility of leveraging existing PCH DB implementation available in the HP aCC compiler for the TRICK prototype. Though it was good for a quick prototype, the PCH DB implementation is insufficient to represent complex information such as "points to analysis information".

We would track the changes to the application sources for incrementally updating the CIDB and for verifying if the information in the database is in sync with the sources. The "Program Database" computes digital signatures to detect changes in source code.

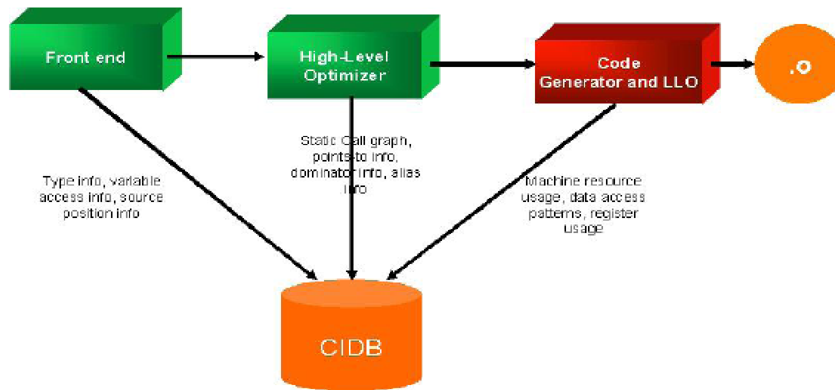

Figure 2. Information passed on to CIDB from various compiler components

In future we plan to remodel our database design along the lines of the Source Code Algebra studied in [12]. Source code algebra [12] provides an algebraic framework that can represent both structural and flow information in a single data model. We plan to extend the SCA work to support other compiler analysis information as required by TRICK.

2.3 Tools Information Interfaces (TII)

These interfaces define the flow of information between the database and the tools. They are used by the tool developer for retrieving the information from the database. These interfaces can provide the information in multiple formats such as text, graphs, sets, etc. TRICK will also

provide multiple visitor patterns which can be used by tool smiths to traverse the information and process it in a manner required by the tool. It is also possible for the tool developer to develop his own TII using existing TIIs.

Some possible examples of Tools Information Interfaces are enumerated as follows:

Get caller list: Given a function name, it returns the list of functions which call that function. This is extracted from the High Level Information generated by the compiler and stored in CIDB.

Get dominator tree: Returns dominator tree information for a function. This information is built from the intermediate level information generated by the compiler's high level optimizer..

Get application call graph: Static call graph rooted at a particular function is returned. This information is built from the intermediate level information generated by the compiler as part of its inter-procedural analysis phase.

Get indirect call targets: For each indirect call site (specified in source position information) return the potential list of callees associated with that call site.

Get mach res usage: Machine resource usage of the application instructions is returned. This information is built from the low level information generated by the compiler as part of its low level optimization and stored in CIDB.

3 Usage of TRICK framework:

We now give some brief examples of how TRICK framework can be used for simplifying tool development and describe couple of tools which have been prototyped using TRICK. TRICK is currently based on the hp-ux compiler suite, and prototyping on gcc is under way.

3.1 Indirect Call Tracker

The **I**ndirect **C**all **T**racker (**ICT/ict**) is a compiler-based tool that informs the user of the possible targets of an indirect function call statically (i.e. without any knowledge of application's runtime behavior). ICT is useful for impact analysis for cases when the call graph becomes inscrutable due to the presence of indirect function calls. By exposing the set of possible targets, at the points where indirect calls are made, the accuracy of impact analysis can be improved. We have prototyped a indirect call tracker tool which obtains the results of indirect call analysis done by the compiler, for improving the call graph and for de-virtualization, from TRICK using "GetIndirectCallTargets" interface.. We retain this information and feed it to an impact analysis tool which needs this information to improve its accuracy of analysis. This has been employed as part of a major ISV's build environment system and has been used in analyzing software having millions of lines of code.

3.2 Imake

In a program like Make [14], a compilation unit is recompiled if either the compilation unit has changed or if the context on which the compilation unit depends on changes. But recompilations due to a change in a particular context may be redundant since that context change may not be directly relevant to the compilation unit. The reason for these redundant recompilations is the fact that build systems like 'make' maintain dependencies at a high translation unit level.

Smart recompilation systems minimize the redundant recompilations, by analyzing the source code to build the knowledge of the dependencies at granularities finer than the translation unit [17]. Smart recompilation tools do extensive source code analysis to build up knowledge about dependencies. This information is generated by the compiler as part of its routine compilation

process, and hence is available to the tools by querying the CIDB. We are prototyping a minimal smart compilation where a tool which diffs the sources to determine which definitions have changed and then queries the CIDB to determine the translation units that directly or indirectly use these definitions. It uses the information obtained from CIDB to modify the translation unit level dependency information in the makefiles which are in turn used by the make programs.

3.3 Refactoring tools

Refactoring [11] is a series of behavior preserving transformations aimed at code reuse, improvement etc. We describe a few functionalities of refactoring tool which can benefit from using the TRICK framework:

1. Determining Impact of modifying/deleting a function entity: For doing this the refactoring tool needs to know the dependencies on a function. This information is generated by the compiler front end and forms a part of the high level information that is stored in CIDB.
2. Removing dead code: The tool can retrieve this information from CIDB at function, source file or whole application level and refactor the source code accordingly.
3. Extracting a function for reuse: This will require the tool to extract the function definition along with the minimal set direct or indirect dependencies. The tool can query the CIDB for function dependency list using Get dependency list tool interface.

In the absence of TRICK, the refactoring tool developer would need to develop a parser and AST builder and then perform analysis on the AST created by them. By using TRICK framework, the tool developer can utilize the compiler generated information obtained using well known and tested TIIs and hence cut down the complexity and time needed for developing, testing and maintaining a refactoring tool. TRICK not only simplifies the task of tool development but also improves its performance by providing ready analysis results.

3.4 Code Coverage Tools

Code coverage tools are generally built using static source code instrumentation. Most existing source code coverage tools insert instrumentation at the beginning of each basic block, while some of the more sophisticated ones try to reduce the number of instrumentation points thus reducing the instrumentation overhead. Hence many sophisticated tools use the dominator tree information to reduce the number of instrumentation points [3]. Building the dominator tree is a cumbersome and complex job. But, under the TRICK framework, the code coverage tool can retrieve the dominator tree information from CIDB using the Get dominator tree TII and then just use this information to select the instrumentation sites.

3.5 Performance analysis tools

Static low level performance analysis tools are used to analyze the application and provide suggestions to the developer to improve the performance. These static analysis tools identify the portions of the code which can possibly suffer from machine resource crunches. In order to perform such analysis, these tools need to examine the application binary, build a resource usage model based on the underlying architecture, and report the usage of resources by different source constructs to the developer. This information is generated by the compiler low level optimizer (LLO) while doing instruction scheduling and is present as a part of CIDB’s low level information. Hence static low level performance analysis tools built using the TRICK framework can use the Get mach res usage TII, for retrieving the instructions’ resource usage information from the CIDB. They can then use this information for reporting machine resource usage.

4.0 Related Work:

I. There have been instances of object file itself being used for retaining some of the information post compilation. One example of this is annotations passed by the compiler to other tools that use these annotations to perform some actions on the binary [7]. Another example is the smart recompilation tool [17] that uses reference sets to decide what components needs to be compiled in the next compilation cycle. To pass this information from one compilation cycle to the next one they extended the symbol table to store this information and then extract it into a file which is used in the next compilation. Retaining the compile information in the object files makes reuse of the information very limited because it binds the information to the file format and compiler vendor. It also requires the client to read the object file and extract the information from there. The effort required for this might overweigh the benefits in case of simple tools like impact analysis tools. Also there is a limit on the amount of information that can practically be stored in the object file. TRICK Framework overcomes all these limitations by storing the information in a database and providing API's for retrieving the information.

II. There have been many research projects around compilation and binary instrumentation frameworks which have been used to develop many program analysis tools. (SUIF, ATOM, DYNINST). They have been used to allow developers to add new analysis phases easily to an existing framework, either for optimization or instrumentation. However none of them to our knowledge, allow the reuse of information generated by compiler itself. TRICK differs from them in the sense that the Program Database of TRICK framework contains the complete knowledge of application and tries to facilitate reuse of this information by other tools without the need to recompute this information.

III. In their paper, [4] Devanbu et.al. have done a detailed study of source code analysis tools and brought out the fact that most static source code analysis tools differ only in the way in which they traverse the abstract syntax tree. They reuse an existing front end and define a scripting language for specifying the kind of analysis that needs to be done on the abstract syntax tree. We believe that these tools can reuse not only the front end as done in GENOA framework[4], but also the results of various analysis that are done by the compiler especially when compiling at higher levels of optimization and this is what the TRICK framework provides. In addition to providing the reuse of compiler front end, TRICK framework also provides the reuse of compiler analysis thus providing a double benefit of easing the task of tool development and at the same time making resultant tools more efficient.

5.0 Conclusions

In summary, we aim to provide a mechanism whereby the information generated by the compiler during compilation can be tracked, retained and reused by the development environment tools. We provide standard querying interfaces to the CIDB so that they can be used programmatically by any tool developer. We plan to facilitate the tool developer to extend the query interfaces as per his requirement, by providing interfaces to the metadata information of the database. In future it is also possible to provide a mechanism for tools to insert results of their analysis into the database so that it can be re-used by compiler or other tools. We believe that TRICK will help in freeing up the tool developer's from the task of doing standard compiler analyzes and will enable them to concentrate on the processing and presentation of the gathered information in a suitable

manner to the end user. It will also enable programmers to easily develop customized tools which best address their needs since TRICK simplifies tool development and move it from a realm of expert tool developers to that of general programmers. Since the same framework can be used for any programming language the tools are delinked from the idiosyncrasies of various programming languages. Effectively we see the possibility of TRICK evolving from its current form of being a mechanism for retaining compilers knowledge to a framework for sharing information between various development and deployment tools. But there are open questions on how best to provide open interfaces to access the retained information and what kind of information representations a framework like TRICK should support. We are currently working on these problems.

References

[1] Method and system for optimizing compilation time of a program by selectively reusing object code. In *US Patent Office Application Number 10/017,572, 2001.*

[2] H. Agrawal. Efficient coverage testing using global dominator graphs. In Workshop on Program Analysis for Software *Tools and Engineering, 1999.*

[3] H. W. Cain, B. P. Miller, and B. J. Wylie. A callgraph-based search strategy for automated performance diagnosis. In *International Euro-Par Conference on Parallel Processing,* 2000.

[4] P. T. Devanbu. Genoa: a customizable language- and frontend independent code analyzer. In *International conference on Software engineering, 1992.*

[5] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *ACM symposium on Operating systems principles, 2003.*

[6] J. Goguen, J. Thatcher, and E.Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Current Trends in Programming *Methodology, Vol 4, chapter 5, Prentice Hall, 1978.*

[7] R. Hundt. Hp caliper. In Workshop on Industrial Experiences *with Systems Software, 2000.*

[8] S. J. The cscope program. In *Berkley UNIX Release 3.2*, 1981.

[9] S. McFarling. Reality-based optimization. In *International symposium on Code generation and optimization, 2003.*

[10] S. McFarling and J. Hennesey. Reducing the cost of branches. In *International symposium on Computer architecture*, 1986.

[11] W. Opdyke. Refactoring object-oriented frameworks. In *Ph.D Thesis, University of Illinois at Urbana-Champaign,* 1999.

[12] S. Paul and A. Prakash. Supporting queries on source code: A formal framework. In International Journal of Software *Engineering and Knowledge Engineering, 1994.*

[13] K. Pettis and R. Hansen. Profile guided code positioning. In *International conference on Programming Language Design and Implementation, 1990.*

[14] F. S. Make - a program for maintaining computer programs. *In Software Practice and Experience, 9(3):255–265, 1979.*

[15] W. J. Schmidt, R. R. Roediger, C. S. Mestad, B. Mendelson, and I. Shavit-Lottem. Profile-directed restructuring of operating *system code. In IBM Systems Journal - 37-2, 1998.*

[16] R. W. Schwanke and G. E. Kaiser. Smarter recompilation. *In ACM Transactions on Programming Languages and Systems (TOPLAS), 1988.*

[17] W. F. Tichy. Smart recompilation. In *ACM Transactions on Programming Languages and Systems (TOPLAS), 1986.*

[18] A. W. A. Zhong Shao. Smartest recompilation. In *Symposium on Principles of programming languages, 1993.*