Whole program compilation for embedded software: the ADSL experiment

A. Johan Cockx, IMEC, Belgium

ABSTRACT

The increasing complexity and decreasing time-to-market of embedded software forces designers to write more modular and reusable code, using for example object-oriented techniques and languages such as C++. The resulting memory and runtime overhead cannot be removed by traditional optimizing compilers; a global, whole program analysis is required. To evaluate the potential of whole program optimization techniques, we have manually optimized the embedded software of a commercial ADSL modem. Using only techniques that can be automated, a memory footprint reduction of nearly 60% has been achieved. We conclude that a consistent and aggressive use of whole system optimization techniques is feasible and worthwhile, and that the implementation of such techniques in a compiler for embedded software will allow software designers to write more modular and reusable code without suffering the associated implementation overhead.

Keywords

Whole program compilation, embedded software, interprocedural optimization, C++

1. INTRODUCTION

Due to increasing complexity and decreasing time-to-market, embedded software designers are feeling an increasing pressure to write more modular and reusable code. The use of OO (Object-Oriented) techniques and languages can improve modularity and reusability, but the acceptance of OO in the embedded world has been very slow. We believe that embedded software designers are reluctant to embrace these techniques because they are afraid of the additional overhead in an embedded environment with limited memory and computational power. A compiler that automatically removes such overhead would therefore be very useful.

OO techniques encourage the use of many small objects that are relatively stand-alone and thus easy to understand and reuse. This is a clear advantage at the source level, but a disadvantage for efficient implementation. It leads to a large number of small (member) functions, and thus not only causes a function call overhead, but also seriously complicates other optimizations such as dataflow analysis, constant propagation and dead code elimination. The use of polymorphism with dynamic binding (virtual functions in C++, non-final functions in Java) enables even further splitting of objects and thus better modularity and reusability, at the cost of an increased function call overhead and optimizations problems.

Traditional optimizing C++ or Java compilers cannot remove this overhead, because these compilers examine the code file per file (more precisely: compilation unit per compilation unit), and never see the whole program at once. Since the calling context of functions is usually not known, optimization is done function per (small) function, and the context information necessary for aggressive optimizations is not available. The linker, on the other hand, does see the whole program at once, but the object code no longer contains sufficient information for most optimizations.

Interprocedural optimization is a well established discipline that attempts to reduce or eliminate the performance penalty of using procedure (or function) calls to structure software. Most research on interprocedural optimization targets high performance scientific computing, often on multiprocessor hardware, and typically using FORTRAN as a programming language. Care must be taken when applying the results of this research to embedded software; some techniques obtain improved performance at the cost of seriously increased code size, which is often inappropriate for embedded software. Also, techniques developed for FORTRAN cannot always be applied for C++[13].

Interprocedural optimization for object-oriented languages and the associated problem of polymorphic calls was pioneered in 1991 in the SELF system[6] and since then, techniques have been developed that are adapted for use in most object-oriented languages including Cecil[7], Eiffel[1, 2, 3, 4], C++[5], and Java[8, 7]. A recently started project on interprocedural optimization for object-oriented languages is the OSUIF project at UCSB[9].

Research projects that specifically target interprocedural optimization for embedded software are rare and limited in scope. The MOVE project[11] will try to parallelize FOR-

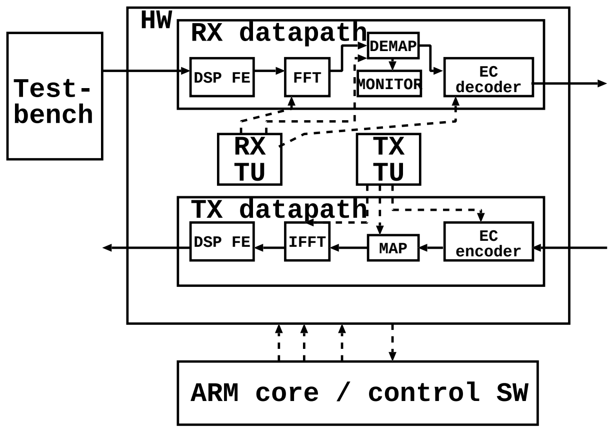

Figure 1: *Architecture of the ADSL modem. The embedded software on the ARM processor controls the hardware datapath through two timing units (TU).*

loops for mapping on multiple embedded processors. Sjdin[12] reports a technique using whole program analysis to optimize the allocation of global data in on-chip RAM.

The terms "interprocedural" and "whole program" seem to be closely related. I prefer to use the term "whole program" because it does not suggest that optimization is restricted to a reduction of procedure call overhead. Some of the techniques that we have used and will present further are not related to procedure calls but to data allocation.

To evaluate the potential for whole program optimization on industrial quality software, we have taken the source code for the embedded software controlling an ADSL modem (described in section 2) and manually optimized it by rewriting the source code, disregarding all modularity and reusability issues. Our optimization target was memory footprint, not performance. The techniques used can in principle be automated and are described in section 3. We were able to reduce the memory footprint of this software by nearly 60%; detailed results are given in section 4 and conclusions in section 5.

2. THE ADSL DRIVER APPLICATION

To verify the feasibility and effectivity of whole program compilation for industrial embedded software, we have manually optimized the software for a commercial ADSL modem (figure 1). The software runs on an ARM processor and controls the hardware datapath that processes the high speed data passing through the modem. Since the software does not process high speed data, the execution time is far from critical and we decided to optimize memory footprint (code + data).

The software is written in a mix of C and C++, with a little bit of assembly code for startup and interrupt handling. It is designed using the OCTOPUS[14] methodology, compiled with the C and C++ compilers provided by ARM (version 2.50) and runs on top of the Virtuoso[10] RTOS. The source code of the complete program was available, except for the RTOS, which represents less than 2% of the total footprint. The source code is about 130.000 lines long and is distributed over 272 C/C++ files and the same number of header files. The C++ code uses classes, multiple inheritance, virtual functions and dynamic allocation. The code is multithreaded and uses Virtuoso queues for interthread communication. Communication with hardware is done through interrupts and memory mapped I/O using a proprietary protocol.

The software can be configured for different configurations of the ADSL modem (central office or customer premises usage, different versions of the ADSL standard, etc) using preprocessor macros. We have optimized for one specific configuration. The code that is specific for other configurations is of course removed by the preprocessor and does not contribute to the footprint of the compiled software, but we have found many instances of code shared by two or more configurations that was more complex and had a larger footprint then necessary for our specific configuration.

3. OPTIMIZATION TECHNIQUES

The ADSL software is optimized by transforming the source code; this implies that only techniques of which the result can be expressed in C++ have been used. For example, register allocation based on interprocedural dataflow analysis cannot be expressed in C++ code and has therefore not been used.

Since all optimizations can be done by rewriting the C++ code, they can also be done manually. Manual application of these optimizations is however not desirable because it breaks the modularity and reusability of the code. Our goal is to allow software developers to write modular and reusable code without worrying about implementation efficiency.

Most of the techniques used have only a small effect by themselves; it is the combined effect of applying all these techniques that produces a significant amount of optimization. Very often, application of one technique (e.g. function inlining) exposes opportunities for another technique (e.g. constant propagation). As a result, it is impossible to measure the impact of each technique separately.

Only static (compile-time) techniques have been used; we do not rely on dynamic techniques such as run-time optimization (e.g. caching for dynamic dispatch of polymorphic calls) or profiling.

The optimization techniques used can be classified in three groups: traditional interprocedural optimization, OO-specific optimization, and data allocation optimization.

3.1 Traditional interprocedural optimization

The following interprocedural optimization techniques have been applied.

- agressive constant propagation and dead code elimination.
- Inlining of functions with a single call site.
- Inlining of very simple functions, e.g. returning a data member. Note that functions must be very small to

save memory footprint when inlined; when optimizing for speed, they can be larger.

- Removal of unused variables, including unused data members of structs and classes. Note that traditional interprocedural optimization does not change data structures such as structs and classes; we have been a bit more aggressive here.
- Replacement of initialized variables that are never written by a constant (which can then be propagated).
- Removal of variables that are never read.

The ADSL software contains a number of variables (actually, large arrays) that are never read by the software itself, but instead are intended to be read by the debugger. These variables are used as an alternative to `printf`, which is often unavailable in embedded software. Compilation of these variables and the code writing to them did not depend on a preprocessor macro. It is not clear whether this was intentional (i.e. debugging functionality was intended to work even in production code) or just not done because it is a lot of work and makes the code less readable. We have therefor separately recorded the optimizations resulting from removing debugging variables and code, and present figures with and without debugging code in the results section.

We have carefully avoided optimizing variables marked `volatile`, as these may be written or read by interrupt handlers or used for memory-mapped I/O.

3.2 OO-specific optimization

From the literature on interprocedural optimization for object-oriented languages, we selected some techniques that can potentially reduce footprint as well as execution time. Many other techniques are not appropriate for us because they only reduce execution time at the expense of memory footprint.

- Removal of `virtual` keyword from functions that are never called polymorphically (i.e. via a pointer to a baseclass).
- Replacement of polymorphic calls by a runtime type test followed by a statically bound call. To make this possible, an `enum` data member identifying the type has to be added to the classes involved. We found that this technique only reduces memory footprint for very small functions, or when inlining and other optimizations are enabled by this transformation.

3.3 Data allocation optimization

Whether data is allocated statically, dynamically (`new/delete`) or automatically (on the stack) can have an important impact on memory footprint. We have applied four techniques to reduce the memory footprint of the ADSL software by changing the way data is allocated. To our knowledge, these techniques have not previously been published in the context of compiler optimization, so we give a more detailed explanation than for the previous types of optimizations.

1. *Data inlining.* If some data `B` is always dynamically allocated when an object of class `A` is created, and deleted when the object is destroyed, then `B` can be implemented as a data member of `A`. For example:

   ```
   class A {
     B* b;
     A() { b = new B(0); }
     A(int i) { b = new B(i); }
     ~A() { delete b; }
   };
   ```

   can be replaced by

   ```
   class A {
     B b;
     A(): b(0) { }
     A(int i): b(i) { }
     ~A() { }
   };
   ```

   This situation typically occurs when, for reasons of reusability, class `A` is written such that it can work with any derived class of `B`. The transformation is only possible when, in a given configuration of the software, only one concrete class `B` is used. The transformation eliminates dynamic allocation and a pointer indirection when accessing `B`, so it optimizes both execution time and memory footprint.

2. *New on entry, delete on exit.* If some data `B` is dynamically allocated on entry to a function `f` and deleted on exit, then `B` can be implemented as an automatic variable in `f`. For example:

   ```
   void f() {
     float* myarray = new float[100];
     process(myarray);
     delete myarray;
   }
   ```

   can be replaced by

   ```
   void f() {
     float myarray[100];
     process(myarray);
   }
   ```

   In the ADSL code, this situation commonly occurs for arrays. For reasons of reusability, the array size is an argument of the function in which it occurs. Since C++ does not provide automatic (on-stack) allocation of variable sized arrays, these arrays were allocated dynamically. In our specific configuration of the ADSL software, the array size is the same and constant for all calls, so automatic allocation can be used.

   The transformation eliminates dynamic allocation and a pointer indirection when accessing `B`, and places the data on the stack where it may automatically reuse memory already reserved for other branches of the call tree.

3. *New on startup, no delete.* If some data `B` is allocated at the beginning of the main function and never deleted, it can be implemented as a static variable. The transformation eliminates dynamic allocation and a pointer indirection when accessing `B`.

4. *Static reinitialized on entry.* If a static variable B is reinitialized on every entry to a function f, it can be implemented as an automatic variable in f. This allows branches of the call tree on which f does not occur to reuse the memory occupied by B.

To succesfully apply these techniques, the compiler must understand how the RTOS configures stack sizes and dynamic allocation pools, and be able to retune them after these transformations. A compiler able to do this would be very useful for embedded software development; manual derivation of stack and pool sizes is very tedious, and a software designer under time pressure (are there any others?) will often prefer to use a large safety margin, which can significantly increase memory footprint.

4. RESULTS

The memory footprint reduction (tables 1 and 2) achieved by whole program optimization techniques is much larger than the footprint reduction achieved by the local optimization techniques implemented in the ARM C++ compiler (6% on the ADSL software).

	Code (bytes)	Data (bytes)	Code +Data
Unoptimized footprint	248552	402080	650632
Compiler optimization	-36652	0	-36652
Remove debug code	-17160	-225676	-242836
Interprocedural optim.	-34252	-28000	-62252
OO-specific optim.	-3408	0	-3408
Data allocation optim.	-448	-55428	-55876
Optimized footprint	156632	92976	249608

Table 1: ***Memory footprint reductions achieved by different optimization techniques on the ADSL software.***

	With debug code	Without debug code
Original footprint (bytes)	613980	371144
Remove debug code	40%	0%
Interprocedural optim.	10%	17%
OO-specific optim.	0%	1%
Data allocation optim.	9%	15%
Total footprint reduction	59%	33%

Table 2: ***Percentage footprint reduction achieved by different whole program optimization techniques. All percentages are relative to the footprint of the original code with maximum compiler optimization (-O2 -Ospace).***

About 40% of the original footprint is due to debug functionality. Manually removing the debug functionality took about one man-week. One could argue that this debug functionality should be retained even in a production release under some circumstances; our results are therefore presented both with and without debug functionality.

Manual application of the whole program optimization techniques took about three man-months, excluding learning time. For most applications, this means that manual optimization is not worthwhile. To make these techniques widely applicable, it is essential that they are automated in the context of a whole program compiler.

The impact of OO-specific optimizations is negligible. This is probably due to the fact that in C++, only functions that are explicitly declared `virtual` use dynamic binding; in other words, the designer already optimizes a lot by making most functions non-virtual. The problem with this approach is that it makes classes less reusable. In Java, all functions use dynamic binding by default. Another fact explaining the small impact of OO-specific optimizations is that the traditional technique of replacing polymorphic calls by an `if`-tree often only improves performance, and increases memory footprint. A footprint reduction is only achievable in a few special cases described in the previous section.

5. CONCLUSION

By applying whole program optimization techniques to the ADSL software, we have achieved a memory footprint reduction of at least 33%; if it is acceptable to remove debug functionality, 59% is achievable. This compares to a 6% memory footprint reduction by local compiler optimizations for the same code.

We have used three kinds of techniques: traditional interprocedural optimization techniques, OO-specific techniques and data allocation techniques. The first two kinds are well-known and published, but rarely used in compilers for embedded systems, and usually target performance optimization rather then memory footprint reduction. The data allocation techniques have not been used in any compilers or published to our knowledge.

All proposed optimizations can be done by rewriting the C++ source, but doing this manually ruins modularity and reusability. To be useful, these optimizations must be implemented in a compiler. The data allocation techniques require the compiler to understand how the RTOS implements dynamic allocation and stacks. The implementation of a whole program compiler automating these optimizations is not a trivial job, but the potential gains are also large.

We believe therefore that implementation of a whole system compiler for embedded software is both feasible and worthwhile. We also believe that the main benefit of a whole program compiler for embedded software would not be a reduced memory footprint, but the fact that it will allow designers to write more modular and reusable code without worrying about the associated implementation overhead.

Acknowledgement

This research was partly supported by the flemish IWT in the context of project number 980314. Their support is gratefully acknowledged.

6. REFERENCES

[1] `http://www.loria.fr/projets/SmallEiffel/`

[2] S. Collin et al. “Type Inference for Late Binding. The SmallEiffel Compiler”, *JMLC'97,* pages 67–81.

[3] O. Zendra et al. “Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler.” *OOPSLA'97,* Volume 32, Issue 10 - Atlanta, GA, USA, October 1997, pages 125–141.

[4] D. Colnet et al. “Optimizations of Eiffel programs: SmallEiffel, The GNU Eiffel Compiler.” *TOOLS Europe'99,* IEEE Computer Society, Nancy, France, June 7-10, 1999, pages 341–350.

[5] G. Aigner and U. Hoelzle. “Eliminating Virtual Function Calls in C++ Programs” *ECOOP'96,* SpringerVerlag LNCS 1098, pp. 142–166.

[6] U. Hoelzle et al. “Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches”. *ECOOP'91,* p. 21–38, Geneva, July 1991. Springer-Verlag LNCS 512.

[7] Dave Grove. *Effective Interprocedural Optimization of Object-Oriented Languages.* Ph.D. Thesis, University of Washington, 1998.

[8] TowerJ Java compiler. `http://www.towerj.com`

[9] OSUIF project at UCSB. `http://www.cs.ucsb.edu/~osuif`

[10] Virtuoso RTOS from Eonic Systems. `http://www.eonics.com`

[11] MOVE project. `http://ce-serv.et.tudelft.nl/MOVE/tasks.html`

[12] J. Sjdin et al. “Static Allocation of Global Data Objects in On-Chip RAM.” *CASES'98*, December 1998.

[13] `http://www.oonumerics.org/oon/oon-list/archive/0328.html`

[14] M. Awad et al. *Object-Oriented Technology for Real Time Systems: A Practical Approach Using OMT and Fusion*, Prentice Hall, 1996. ISBN 0132279436.