Summary of Compiler techniques for code compaction

Andreas Ehliar

March 24, 2004

1 Introduction

The paper deals with techniques for code compaction on the Alpha processor. Some of the techniques are usable on other architectures and some of the techniques are rather platform specific.

The main motivation for code compaction as specified in the paper is embedded devices where memory can be one of the most limiting factors.

The authors what they term a whole-system approach to code compaction where they use aggressive interprocedural optimization aimed at reducing the code size. They also abstract similar or identical blocks into one procedure with calls inserted at the appropriate locations.

The authors do not modify a compiler to test their proposed technique. Instead they rely on existing compilers (gcc and the vendor supplied cc) to compile the code. They have implemented an optimizer called *squeeze* that they run on a linked program.

2 Optimizing for code compaction

The authors specify several optimizations that are useful for code compaction.

2.1 Redundant code eliminiation

Repeated computations of the same value can be removed. Repeated computations of the same value often occurs on the Alpha as an artifact of how global variables or function calls are resolved. The compiler cannot know if the same global pointer can be used at compile time. But after linking it is possible to statically resolve this.

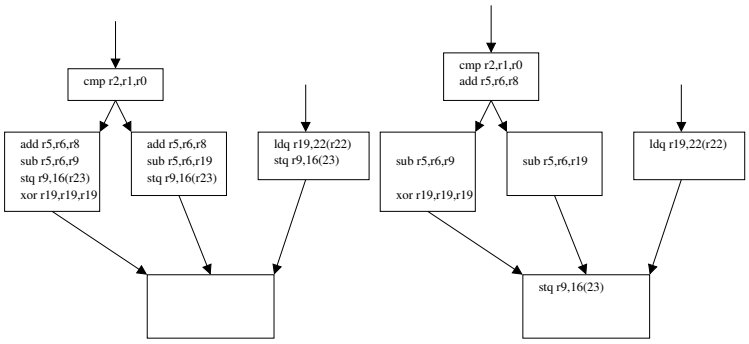

Figure 1: An example of local code factoring.

2.2 Unreachable code elimination

It is quite common that an interprocedural analysis of the entire program will reveal code that is not reachable by any path. This is commonly caused by for example static function call parameters.

2.3 Dead code elimination

An optimization related to the previous optimization is dead code elimination. A computation is dead if the result of the computation is never used. It is common that dead computations occur after running the unreachable code elimination.

2.4 Strength reduction

Strength reduction is commonly to reduce for example an expensive multiplication with several shifts and adds. This is not suitable for code compaction. However, other operations can be reduced in strength. A global function call can be changed from the following generic code;

- load r0 with 64 bit address
- jsr r0

to the following code assuming the destination of the jump is close enough;

- bsr address (pc relative call)

3 Code factoring

The most interesting part of the paper is the part that deals with code factoring. The idea is to find identical code sequences and remove all but one occurence of the code sequence. There are two versions of this optimization, local code factoring and procedural abstraction.

3.1 Local code factoring

Figure 3 shows an example of how certain identical computations can be moved upwards or downwards in the control flow graph. A difference from the usual optimization aimed at high performance is that it is best to move a computation downwards as it will then be reached by a larger number of basic blocks. If a compiler is optimizing for high performance it is better to move the computation upwards so that it will be computed as soon as possible.

3.2 Procedural abstraction

The procedural abstraction is aimed at finding duplicates of a block with a single entry and a single exit point. This is not necessarily a single basic block but it might be several basic block that fulfill the constraint, or even an entire function. After such a block is found, the block is moved to a separate procedure and all occurences of the block is replaced with a procedure call.

This sounds easy, but it is not as trivial as one might think. Different blocks might perform exactly the same function but with different registers. The program implemented by the authors of the paper performed register renaming as long as the register renaming parts of the new procedure call was not long enough to mitigate the gains of the procedural abstraction.

Another problem is that a procedure call on the Alpha requires the return address to be in a register. This requires register liveness analysis of the block to find a free register.

The authors also mentioned that they investigated a method to find partially matched blocks and abstract them. The method was computationally expensive and the gains were quite small.

4 Architecture specific idioms

Certain architectures have certain conventions, either enforced by the hardware or enforced by the ABI. The Alpha ABI have a convention that a function is responsible for saving registers r9-r15 if necessary. Squeeze will abstract the instructions used to save these registers into a separate procedure. Similarly, squeeze will abstract the instructions used to restore the registers.

High performance code for the Alpha will usually contain nop instructions for scheduling reasons. These nop instructions are removed by squeeze to save space.

5 Results

The authors have run squeeze on several programs and compared the results. The average program size savings were 28% – 30% depending on the compiler.

Transformation	Savings
Redundant computation elimination	34.14%
Basic block and region abstraction	27.42%
Useless code elimination	22.43%
Register save/restore abstraction	9.95%
Other IPO	6.06%

Table 1: Code size improvements

Although the savings varied from 15% to 40% depending on the program. Table 5 shows an outline of how the different code compaction techniques performed.

Contrary to the authors expectations, the execution speed was generally improved. This was partly due to the interprocedural optimization and partly due to the reduced load on the instruction cache. The average speedup was 10% for programs compiled with gcc and 16% for programs compiled with cc. Not all programs fared as well, one program was 23% slower while the best program performed 35% faster.

6 Conclusions and critique

The paper was interesting to read and the authors have shown some fairly significant improvements. However, most of the gain came from interprocedural optimizations that should really be done by a good compiler regardless of code size issues.

Another issue is the choice of the processor. While the Alpha have fairly mature tools to do the manipulations required by squeeze, it is not representative of a typical embedded processor. Some of the gains were very Alpha specific. For example, the Arm processor can store (and restore) several registers by using only one instruction. Also, an embedded processor would typically not be affected by the problems the Alpha faces with regards to the access of global variables.

It would therefore be very interesting to see similar experiments with a processor more commonly used in embedded devices like the Arm or perhaps even an 8 bit microcontroller like the AVR or PIC.