# Application of Procedural Abstraction and Cross Jumping

Jürgen Wohlmuth
University of Technology, Vienna
e0125400@student.tuwien.ac.at

## Abstract

In the field of embedded systems, where resources are limited, code compressions becomes an important factor. Two popular compression techniques are procedural abstraction and cross-jumping. This paper will examine their application on different stages of the compilation, namely on object code and intermediate representation. First we will present the basic ideas on the example of object code compression according to Cooper and McIntosh. Later we will discuss the differences to the application of basically the same strategies by Nyström and Sjödin. We will close with a comparison and some considerations about concurrent application of both.

## 1 Motivation

Most compiler optimization focus on runtime improvements but only few of them take memory consumption into account. Focusing on runtime sometimes even means making a program longer by loop unrolling for example. [2]
Especially in an environment which is short on resources as embedded systems are, it is economical to focus on both. Generally this topic also affects other environments where the delivery of a program is the bottleneck like in web applications and Java applets. Concentrating on embedded systems like within cellular phones, industrial control units and many other devices compressing code has a direct impact on cost and power consumption. [1]
As the number of such devices quadrupled in the last years the demand was to put even more sophisticated software and functionality in them which leads to even higher memory requirements.
Here is where code compression on any level of generation becomes necessary. [4]
This paper outlines basically two strategies which both originate in the mid-1970s to the mid-1980s but are still valid on current embedded systems, namely cross-jumping and procedural abstraction. For both techniques it is essentially to identify equal or at least similar sequences of assembly code, so called “repeats”.
To illustrate the basic idea of both strategies, we will mainly refer to the presentation of Cooper and McIntosh ([1]), the basic outline of the paper will be as follows. Section

2 describes how repeats are identified and the mentioned strategies are applied. Section 3 will introduce the application in the context of intermediate representation according to Nyström and Sjödin ([3]), identify differences and discuss the application of both. Section 4 will present a concluding statement of the author and personal opinions.

# 2 Compressing assembly code

As already mentioned, finding equal occurrences of blocks of assembly code will be the first task of the compression framework. A two-stage approach is used to combine the construction of a suffix tree and afterwards building the repeat table. [1]
Suffix trees are well known from pattern-matching applications where every edge represents a possible substring of the original text S. Each path from the root to a leaf node therefore represents a certain suffix within S. [5]
By computing a hash-code for each instruction in the program using the opcode, registers and constants a global instruction table is build. The actual "text" is constituted by a list of indices of the global instruction table corresponding to the sequence of instructions in the original program. We can now use Ukkonen's algorithm ([5]) to construct the actual suffix tree in O(N) where N denotes the texts length, in other words the length of the program in instructions. [1]
Note that we now seek for identical substrings (i.e. code fragments) within the suffix tree and general substrings are constituted by arbitrary subtrees of the suffix tree.
The information about identical substrings/subtrees is stored in the repeat table where each entry is constituted by a set of identical code fragments, each of them extended by its length and offset in the original text/program. Considering a repeat table entry with K fragments, the goal is to eliminate (K – 1) of them securely. [1]
Securely in this context means, that there are no circumstances under which a certain fragment cannot be eliminated. In other words, the flow of control has to be preserved.
Note that some fragments might overlap with fragments in other repeats. The default strategy is to calculate the benefit of each repeat and sort the repeat table accordingly before applying the compression in descending order of the repeat's benefit. Also profile based repeat selection is possible. For more details see [1] sect. 4.3.

## 2.1 Control hazards

Control hazards are circumstances which prevent a certain fragment to be part of the repeat, i.e. there is a branch into or out of the fragment. A typical countermeasure is the so called repeat splitting. A given repeat with N fragments a control hazard at offset h will be split by replacing it with two new repeats with again N fragments, one containing the partition before and the other one the partition after the repeat. [1]
Note that it is pointless to keep the hazardous instruction in any repeat, because the hazard itself is not eliminated but merely ignored.
The repeat manager also implements a cost model to decide if a certain split makes sense. This model takes fragment length, type and offset of the hazard into account so that if the split fragments get too small hazardous fragment(s) will be removed entirely from the repeat. [1]

## 2.2 Procedural abstraction and cross jumping

After the compiler's compression framework has identified repeats within the input program it is time to use this information to decrease code size. Based on certain properties of the fragments within one repeat the mentioned replacement strategies are possible.

Procedural abstraction replaces the fragments of the repeat by calls to a new procedure which is basically accomplished by adding the jump-label at start and a return statement at the end. [1, 2]

Note that it is not necessary to save a stackframe on procedure invocation. As the procedure originates from equal fragments the register allocation remains valid.

Figure 1 shows an example.

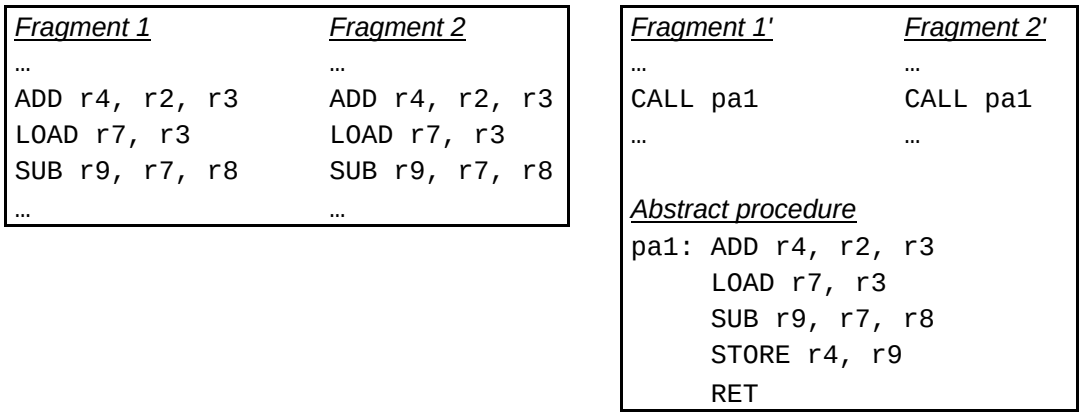


**Figure 1. Procedural abstraction**

Cross-jumping, also sometimes referred to as "tail-merging", is applicable if the identified fragments have a branch instruction to the same destination in common on the last instruction. One fragment is left as it is and the other occurrences are replaced by branch instructions to the start of this fragment. Figure 2 shows the same example as above extended by a common jump in both fragments to enable cross jumping.

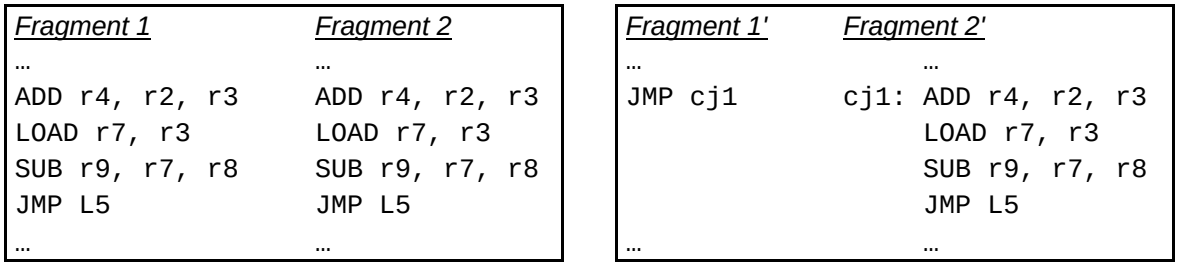


**Figure 2. Cross jumping**

Both strategies have their benefits and costs in code space and execution time. In both cases the compression framework has to add branch instructions at any place where invocation shall take place and in case of procedural abstraction one additional return instruction. [1, 6]

### 2.2.1 Abstracting fragments

The presented constitution of the repeat table is based on lexically identical fragments thus the set of eligible fragments is quite limited. Lexical identity implies identity of all instructions, registers and constants. To provide some degree of freedom to the identification algorithm it is advantageous to replace certain operands and the branches by wildcards. [1]
Branches usually terminate a fragment because of lexical differences in the branch destination labels. To abstract them we have to consider both ways in which they can be identical. Absolute identity of the branch targets is at hand if they reference the identical labels. Branches are relatively equal if they span the same distance in instructions. [1, 6]
We assume that register allocation has already taken place for the program at hand. As it turns out it is often the case that fragments are identical except for the use of registers. The goal now is to rewrite the fragments and adjust the register usage that they get equal and ready to be processed by the matching algorithm. This technique basically requires recoloring of the live ranges with respect to the other fragments. For more details see [1] sect. 4.2.
A possibly interesting future extension would be the abstraction of constants because the current framework is not able to identify similar fragments which only differ in the constants used. One way would be to parameterize the abstract procedures generated by the compiler and use the parameter instead of the immediate value. One precondition would be that there is one free register available per constant. The authors leave this point for future investigation. For further details see [1] sect. 7.

## 3 Discussion and comparison

So far we summarized the basics of suffix-tree based code compression and possible extensions. While some ideas of suffix-tree based compression date back to the 1970s and 1980s (e.g. [2]), relaxed pattern matching opened new possibilities for a compression framework to find repeats as it is not locked in lexical equality any more. While the impact to code size is possible negligible for general purpose CPUs in today's PCs there might still be a noteworthy impact when it comes to embedded systems where one major difference is the availability of resources. [1]
An early paper by Fraser, Myers and Wendt ([2]) already described suffix-tree based compressions but the invention here was the depth of abstraction on the assembly code at hand. Compared to that, Cooper's and McIntosh's research ([1]) aimed more towards gaining abstract information which itself can be incorporated into compression. As mentioned in section 2.2.1 some kind of liveness graph is reconstructed for register abstraction as register allocation has already taken place. The question arises if it would sense to conduct the mentioned compression strategies in other stages of the compilation procedure to reduce or even eliminate the effort to reacquire information which is already lost.
As code compression typically operates on machine code using some kind of string matching algorithm (as the presented one), Nyström and Sjödin investigated the possibilities of applying them to intermediate code. This makes sense as intermediate code is suitable for other optimizations as well and allows identification of semantically equal but after register allocation most likely lexically different

fragments. An eligible precondition which Nyström and Sjödin name explicitly but also applies to machine code compressors is, that the framework (i.e. the compiler) shall have access to the whole program. The main advantage would be that optimization can take place over the - potentially very tight - borders of modules.
Naturally there are many similarities shared by both approaches and they can be mapped directly. By doing so, we can try to identify pros and cons of either based on the possibilities and points of view they provide.

### 3.1 Pros of the intermediate representation

The main point when using intermediate representation is that almost no abstraction from the source is necessary. The effort to abstract information about registers and branches can be saved because this representation is already in an “abstract” form. The approach by Cooper and McIntosh needs to reverse engineer this information from the object code as briefly discussed in section 2.2.1.
The fact that register allocation has already taken place introduced much complexity and possibly drawbacks because of powerful allocation algorithms. [3]

### 3.2 Cons of the intermediate representation

Object code compression can be applied to object code only. Using compression on intermediate code probably leads to a complete recompilation, assuming that some modifications on the intermediate code generation are necessary too.
Possibly a big drawback is the missing of opportunities for optimization after object code generation has taken place. The impact of function prologue and epilogue might be considerable to compression in some cases. [3]

### 3.3 Future prospects

We identified calling of a function as a critical factor when operating on intermediate representation. Since new procedures are generated the used calling conventions are a critical factor. By using a calling convention, which passes parameters and return values in registers, the overhead can be kept low. As discussed in section 3.2, saving the stack frame might still be an opportunity to miss. The usage of interprocedural register allocation potentially can decrease the impact. But the problem still exists that due to introduction of function prologue and epilogue new compression opportunities are introduced. [3]
As these opportunities are visible in object code it would be reasonable to combine both strategies and apply them in sequence. To examine the full potential of the combined application of both strategies, concerning compression ratio as well as compile time, it would be reasonable to analyze several combinations, namely the combination of both described methods and the combination of adapted variants which aim for disjoint sets of compression opportunities.

## 4 Conclusion

Both techniques themselves are quite interesting and also effective. From a detached point of view we would prefer the intermediate code representation as the main point to introduce compression to the input program. The abstract form is far more suitable

to get insight of the code and some semantics. It is not necessary to reconstruct liveness, program flow or other information as it is when applying compression to object code. Nevertheless we will miss some opportunities which are introduced in subsequent steps of object code generation. Therefore it makes sense to use object code compression as described by e.g. [1] or [2] for further compression but we think that an adapted variant will do because of the runtime overhead introduced by reconstructing information from the object code.

## 5 References


[1] K. D. Cooper, N. McIntosh, Enhanced Code Compression for Embedded RISC Processors, 1999.

[2] C. W. Fraser, E. Myers, A. L. Wendt, Analyzing and Compressing Assembly Code, 1984.

[3] S.-O. Nyström, J. Sjödin, Optimizing Code Size through Procedural Abstraction, 2000

[4] S. Debray, W. Evans, R. Muth, Compiler Techniques for Code Compression, 1999

[5] E. Ukkonen, On-line construction of suffix trees, 1995

[6] W. Wulf, et al, The Design of an Optimizing Compiler, 1975