Runtime Code Generation for Huffman Decoders

Kornilios Kourtis
kkourt@cslab.ece.ntua.gr

09/10/2008

Introduction

Runtime code generation is not a new concept [1] and has been widely used for improving the performance of programs at runtime. Typical examples include Just-In-Time (JIT) compilation [2], [3] and dynamic optimization systems [4], [5]. A class of programs that can benefit from runtime code generation are programs with an execution path that is determined by runtime data, which remain unchanged during execution and their size is constant with regard to the total size of the input data. An example of such a program is a huffman decoder.

Huffman Coding

Huffman coding is an entropy encoding scheme for lossless data compression. It builds optimal prefix codes [1] that achieve size compression by using less bits for frequent symbols. Huffman codes are created based on the frequencies (or equivalently the probabilities) of the symbols of the input data. A binary tree, called the huffman tree, is built and used for both the encoding and the decoding of the data ([6]).

Algorithm 1: Huffman tree creation

```
// S_i:symbols, F_i:frequencies, 1 ≤ i ≤ N
Q = ∅
for i = 1 to N do  Q.append(LeafNode(S_i,F_i))
for i = 1 to N − 1 do
    node = InternalNode()
    node.l = Q.extractMinFreq()
    node.r = Q.extractMinFreq()
    node.frequency = node.l.frequency + node.r.frequency
    Q.append(node)
end
```

[1] prefix codes map input symbols to variable length bitstrings, in which no bitstring is also the prefix of another bitstring. This property simplifies the decoding, since it eliminates ambiguities.

In a huffman tree each input symbol is represented by a leaf node. Moreover, each node is associated with a frequency number: for the leaf nodes it is the symbol frequency, while for the internal nodes it is the sum of the frequencies of their two children. The creation of a huffman tree is presented in Algorithm 1. The algorithm operates on a set of sub-trees, which initially contains all the leaf nodes. At each step the two sub-trees with the minimum frequencies are extracted from the set and used as children of a new internal node. The new node is inserted into the set and the procedure is repeated until there is only one node left.

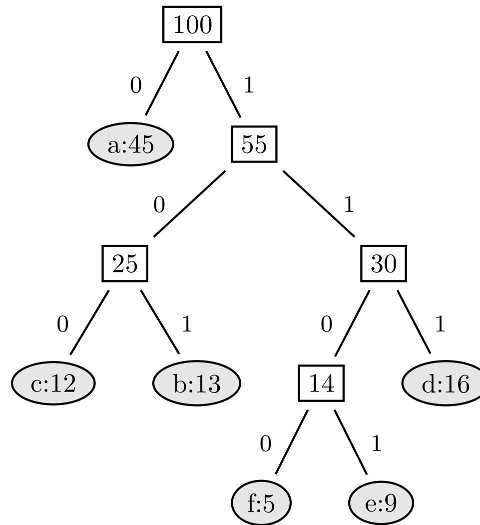

Figure 1: Example of a huffman tree for symbols (a,b,c,d,e,f) with corresponding frequencies $(45, 13, 12, 16, 9, 5)$

An example of a huffman tree (taken from [6]) is presented in Figure 1. The encoding process maps each symbol to a bitstring that is appended to the encoded output. The bitstring for each symbol is determined by the path from the root node to the corresponding leaf node. For example, given the huffman tree of Figure 1, the bitstring for symbol a is "0", while the bitstring for symbol e is "1101". Decoding of a huffman code is performed by iterating the tree based on each bit (left for 0, right for 1) until a leaf is encountered (see Algorithm 2).

Algorithm 2: Huffman Decoding

```
n = tree
repeat
    n = (Data.nextBit() == 0) ? n.l : n.r
    if n.isLeaf() then
        output n.getSymbol()
        n = tree
    end
until Data end
```

The huffman decoding algorithm has two inputs: the huffman tree and the encoded bitstring. The encoded data is the main input of the algorithm in terms

of size, while the huffman tree is used to guide the decompression procedure. Its possible to construct a specialized decoder to perform the decompression of the input data by incorporating the huffman tree into the decoding procedure.

LLVM Compiler Infrastructure

The Low Level Virtual Machine [7] (LLVM) project is a modular compiler infrastructure software suite. It is built around an Intermediate Representation (IR) called the LLVM virtual instruction set, which is a: "low-level object code representation that uses simple RISC-like instructions, but provides rich, language-independent, type information and dataflow (SSA) information about operands"[2]. The highly modular nature of the LLVM codebase makes it ideal for implementing the code generation for the huffman decoders.

The LLVM software provides rich interfaces for building, optimizing, compiling and linking dynamically into the current process modules and functions. Each function contains a number of basic blocks, which consist of a list of instructions that execute sequentially (e.g. they do not contain branches) and are finalized by a terminator instruction (e.g. branch or return).

Dynamic Code Generation for Huffman Decoders

The code for the huffman decoder is generated by traversing the huffman tree and creating the appropriate basic blocks. Algorithm 3 shows a simplified version of the recursive depth-first iterating of the tree. We assume that two basic blocks have been created: `bbRoot`, which is the basic block that corresponds to the root node of the tree and `bbRestart`, which checks if there is more input available and either jumps to `bbRoot` again or exits. Initially the `iterateTree()` is called with `bbRoot` and the root of the tree as arguments. If the node is a leaf, then the code generated appends the corresponding symbol to the output and jumps (unconditionally) to the `bbRestart`. If the node is an internal node, then two new basic blocks are created along with the code for the conditional branch to one of these, based on the input. The `iterateTree()` is called recursively for these two new basic blocks.

The actual implementation is more complex, since it requires the creation of ϕ nodes in the SSA form (`phi` instruction in LLVM) for keeping track of the remaining data that need to be decompressed. After the creation of the function that implements the decoder a number of optimization passes are called to optimize it (e.g inlining) and then the function is compiled at runtime. The creation of the decoder can be performed at the compression side of the communication channel. Instead of the huffman tree, a serialized version of the optimized LLVM code can be transmitted along with the compressed data.

[2] `http://www.llvm.org`

Algorithm 3: Code Creation

```
iterateTree(node, bb) {
    if node.isLeaf() then
        bb.insertCode(output node.getSymbol())
        bb.insertCode(br bbRestart)
    else
        bbl = BasicBlock()
        bbr = BasicBlock()
        bb.insertCode(test = Data.nextBit())
        bb.insertCode(br test bbl bbr)
        iterateTree(node.l, bbl)
        iterateTree(node.r, bbr)
    end
}
```

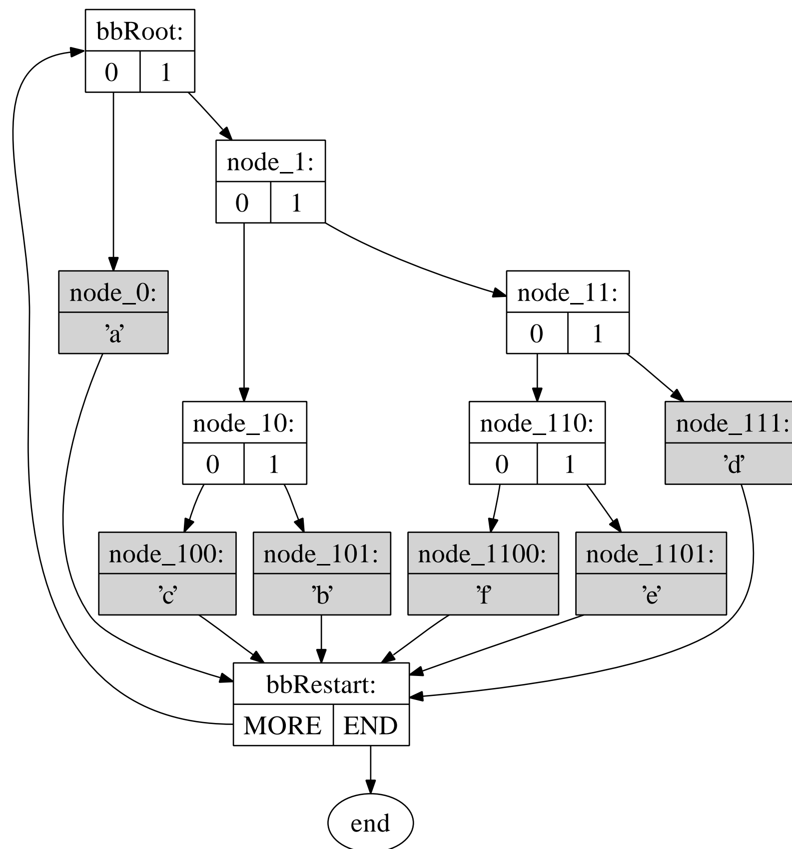

Figure 2: Call flow graph of the resulting huffman decoder for the example of Figure 1

Experimental Evaluation

It is expected that the runtime generation of specialized huffman decoders will improve the performance of the decompression, because it will reduce the number of branches and allow for more effective branch prediction. To evaluate and qualify this improvement we performed a series of tests on randomly created files (512 MB each) with entropy values ranging from 1.5 to 8. The entropy value of a file constitutes the theoretical limit for the compression ratio (bits per symbol required) and thus give us a metric for the compressibility of the file.

The experiments were performed on a machine with an Intel Core2 processor (2.0 GHz), 8 GB of RAM and running a 64-bit Linux OS. The times measured do not include the time spent for the code generation, since we consider that this procedure takes place during the compression. The results are presented in Figure 3 and include the time speedups ($\frac{t_{orig}-t_{cgen}}{t_{orig}}$) for the entropy values of the files.

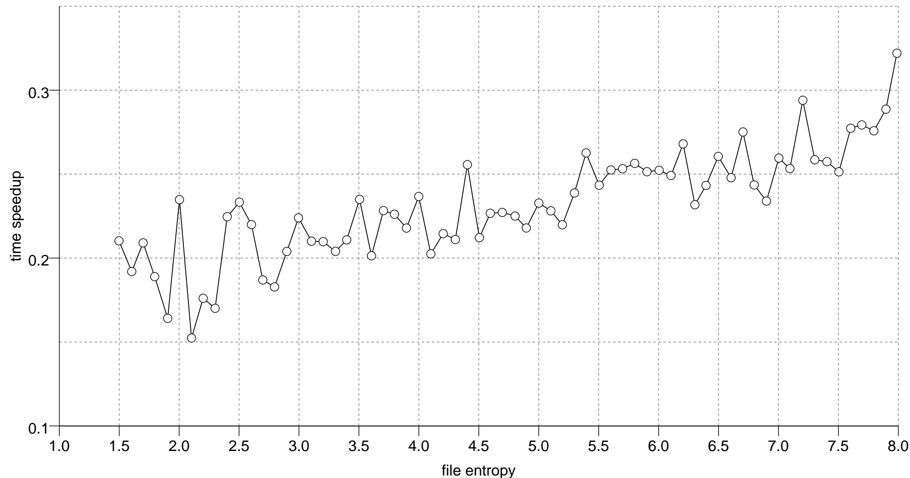

Figure 3: Performance improvement ($\frac{t_{orig}-t_{cgen}}{t_{orig}}$) from applying dynamic code generation for the huffman decoder for various randomly created files

The speedup improvement is 23.2% at average and ranges from 32.2% to 14.2%. Another observation to be made is that, although there isn't a strict relation between the entropy of the file and the speedup achieved, the results indicate that higher entropy values lead to larger speedup. This happens because as the entropy increases, so does the number of branches and branches mispredections making the code generation technique more beneficial.

References

[1] D. Keppel, S. J. Eggers, and R. R. Henry, "A case for runtime code generation," Tech. Rep. UWCSE 91-11-04, University of Washington Department of Computer Science and Engineering, November 1991.

[2] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, "Compiling java just in time," *Micro, IEEE*, vol. 17, pp. 36–43, May/Jun 1997.

[3] J. Aycock, "A brief history of just-in-time," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 97–113, 2003.

[4] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *CGO '03: Proceedings of the international symposium on Code generation and optimization*, (Washington, DC, USA), pp. 265–275, IEEE Computer Society, 2003.

[5] T. Kistler and M. Franz, "Continuous program optimization: A case study," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 4, pp. 500–548, 2003.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

[7] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, (Palo Alto, California), Mar 2004.