Working With Compression

Fabian 'ryg' Giesen

farbrausch

Breakpoint 06

Outline (part 1)

Outline

Motivation and Overview (1)

- 4k and 64k intros stopped being purely about code years ago.
    - Less one-effect intros.
    - Far more “complex” data in intros.
- People expect more and more from intros nowadays.
    - Fancy materials and shading
    - Complex meshes
    - Good textures
    - and so on...
- That takes a *lot* of time to produce.
    - Intros are now $\geq$ 4 months in development.

Motivation and Overview (2)

- It also takes a lot of space.
- Luckily, packers have gotten *much* better lately.
  - Crinkler: Spectacular compression, no .BAT droppers!
  - kkrunchy: Current version packs fr-08 into 50.5 KB.
- It's not that easy, though...
  - To get good compression, data must be stored in a suitable format.
  - ...a somewhat "black art".
- To get good compression ratios, you need to know how compression works.
  - Not in a detailed fashion, just the basics.
  - What's the general idea behind what my packer does?
  - Which types of redundancy are exploited?
- I'll answer these questions in this seminar.

Prerequisites

You'll need...

- Some programming experience.
- Familiarity with mathematical notation can't hurt.
  - Don't worry, no fancy maths in here!
  - I'll talk you through everything anyway.

Outline

Run Length Encoding (1)

- Most of you probably know this already.
- **Idea:** Replace runs of identical symbols by a symbol/run length pair.

  `aaaaabccc` $\rightsquigarrow$ `a5b1c3`

- Simple and fast, but lousy compression for most types of data.
- Often used as pre- or postprocessing step (details later!).
- Also often used to encode runs of one special symbol (usually 0).
  - When this symbol occurs very often...
  - ...and other symbols don't (at least not in runs).
- The basic encoding mentioned above sucks in most cases.
- Better encodings:

Run Length Encoding (2)

- **Packet based**: When a lot of symbols occur only once.
  - Storing run lengths for every symbol is a waste of space.
  - Instead, group into variable-sized *packets*.
    - Each packet has a size, n.
    - *Copy* packets just contain n raw symbols.
    - *Run* packets repeat a symbol n times.
  - This is used in some graphics formats (e.g. TGA).

Run Length Encoding (3)

- **Escape codes**: Use a special code to tag runs.
    - Assuming you have an unused code, this is a no-brainer.
    - No expansion on uncompressible data.
    - Reduces compression gains because escape codes take space.

I mention those schemes because variants of both will become important later on with more involved compression techniques.

Delta Coding

- Another very well-known scheme.
- Not an actual compression algorithm, but a *transform*.
- **Idea**: Don't code the symbols themselves, but *differences* between adjacent symbols.

$$\texttt{1,1,1,2,3,4,5,2} \rightsquigarrow \texttt{1,0,0,1,1,1,1,-3}$$

- Good for *smooth* data that varies slowly: differences are smaller than the actual values and tend to cluster around zero.
- We'll soon see how to take advantage of the latter.
- Again, very simple and fast.
- And again, not of much use for general data.

Quantization (1)

- In this context: using less bits to represent values.
- The main lossy step in all lossy compression algorithms.
- **Scalar quantization**: Using less bits for individual numbers.
  - Uniform: Value range gets divided into uniform-sized bins.
    - 8 bit vs. 16 bit sampled sound
    - 15 bit High Color vs. 24 bit True Color
    - and so on...
  - Nonuniform: "Important" ranges get smaller bins.
    - Floating-point values (more precision near zero).
    - and other examples you probably don't know...
- **Vector quantization**: Code several values at once.
  - A "Codebook" maps codes to encoded values.
  - Codebook needs to be stored (typically quite small).
  - Paletted images: 256 colors out of 2^{24} (RGB triples).
  - Some (old) video codecs: code 4×4 blocks of pixels.

Quantization (2)

- Scalar quantization is usually fine for intros.
- No clever tricks, keep everything simple and byte aligned.
  - Simple code $\Rightarrow$ no subtile, hard to find bugs!
  - Also better for compression, we'll get there soon.
- Example: Throw away least significant byte of 32bit floats.
  - Reduces mantissa size from 23 to 15 bits: enough for most data!
  - Or just use Cg/HLSL-style "half" (16bit floats).
- Another example: Camera (or other) splines.
  - Rescale time to $[0, 1]$, store "real" length seperately.
  - Then store time for spline keys with 12 or even just 8 bits.
- Rounding correctly during quantization is important.
  - Can cut quantization error to half!
  - Compare face in Candytron Party to Final...

Reordering

- Probably the simplest technique of them all.
- Interestingly, also often the most important one.
- Example: kkrunchy x86 opcode reordering
  - After packing with the kkrunchy compression engine...
  - kkrieger code w/out reorder: 77908 bytes
  - kkrieger code with reorder: 65772 bytes
  - Near 12k difference by reordering (mostly)!
  - I'll explain how it works in part 2.
- **Main idea**: Group data by *context*.
  - Values that mean similar things or affect the same parameters should be stored together.
  - So the packer can exploit underlying structure better!
  - Also increases efficiency of delta coding and other simple schemes.
  - $\Rightarrow$ Even more compression at very low cost!
- Not clear yet? Don't worry, there'll be lots of examples.

Example: V2 modules

- For those who don't know, V2 is our realtime softsynth.
- Music made with V2 gets stored as *V2 Modules* (V2M).
- V2 Modules consist of two main parts:
  - *Patches*, basically instrument definitions.
  - And the music, which is a *reordered and simplified MIDI stream*.
- The patches are stored basically unprocessed.
- The music data is a lot more interesting.

V2 music data

- Stored as *events*.
- Every event corresponds to a state change in the player.
  - Note on/off, Volume change, Program (Instrument) change, etc.
- Events come in two flavors:
  - *Channel events* affect only one MIDI channel.
  - *Global events* are mainly for effects and work on all channels at once.
- Every event has a timestamp and type-dependent additional data.
- Channel events are grouped by type, each group is stored seperately:
  - Notes (both actual notes and “note off” commands)
  - Program (Instrument) changes
  - “Pitch Bend” (can be used to change sounds in various ways)
  - Controller changes (velocity, modulation, etc.)
    - ⋆ Again grouped by controller type.

V2 music data (2)

- Note we've already done some reordering:
  - Seperation in global/channel events
  - Further grouping for channel events
- So, how much does that save us?
- fr-08 main tune (MIDI): 20898 bytes.
  - ...after packing with the kkrunchy compression engine.
  - I'll always use packed sizes from now on.
- fr-08 main tune (proto-V2M): 81778 bytes.
  - Oops.
  - Well, so far, it's all very explicit.
  - There's lots of stuff we can do from here on.

Getting it small

- To be sure, that's unrealistically bad.
- First target: Timestamps.
  - That's a 24bit number per event.
  - Which just gets bigger and bigger over time.
  - $\Rightarrow$ Delta-code it!
- With time deltas: 11017 bytes.
  - Much better :)
- Most of the command parameters change smoothly.
  - So delta-code them too.
- Delta everything: 4672 bytes.
  - Pretty impressive for a few subtractions, huh?

Going on

- Let's see what we can do to improve this even further.
- We now went on to reorder the event bytes aswell.
- Right now, we store complete events:
  - Time delta (3 bytes)
  - Parameter 1 delta (1 byte)
  - Parameter 2 delta (1 byte)
- Change that to:
  - Time delta LSB (byte) for Event 1
  - Time delta LSB for Event 2
  - ⋮
  - Time delta LSB for Event *n*
  - Time delta next byte for Event 1
  - ⋮
  - Paramater 2 delta for Event *n*
- **Idea**:
  - Time and parameter deltas aren't related, so seperate them.
  - Usually, the higher time delta bytes are 0, so we get long runs.

Results

- With reordering: 4995 bytes.
  - vs. 4672 bytes without.
  - Again, oops.
- ...and that's why you should always test transforms seperately.
  - We added deltas+reordering at the same time, so we never noticed.
  - Until about a week ago, that is :)
- Still, it shows how useful such simple transforms are.
- Actually, none of the transforms I can recommend are complex.
  - The biggest difference is transforms vs. no transforms.
  - Fine-tuning can give you another 10%
  - ...at the expense of more code.
  - Seldomly worth it!
- Anyway, on toward more compression basics...
  - So we can develop a feel for what's worth *testing*.

Outline

Codes (1)

- Actually *binary codes*.
- A binary code is basically a table that maps symbols to bit strings.
  - Or a function $C : \Sigma \to \{0,1\}^+$ if you prefer it formal.

Example binary code

$$\begin{array}{lcl} \mathrm{a} & \mapsto & 0 \\ \mathrm{b} & \mapsto & 10 \\ \mathrm{c} & \mapsto & 11 \end{array}$$

- You code strings of symbols by concatenating the codes of individual symbols (In the example: aabc $\to$ 001011).
- A code is *uniquely decodable* if there's a unambiguous backward mapping from there.
  - Everything else is useless for compression.

Codes (2)

- A *prefix code* is a code you can decode from left to right without lookahead.
  - Not a formal definition, but good enough for us.
  - The example code on the last slide was a prefix code.
  - For all uniquely decodable codes, there are prefix codes of equal length.
  - So we just use prefix codes, since they're easy to decode.
- So how do we get good codes?
  - There are rule-based codes:
  - Unary code: $1 \mapsto 0$, $2 \mapsto 10$, $3 \mapsto 110$, ...
  - Other codes: Golomb code, γ-code, ...
  - All good for certain value distributions.
- Or generate them from your data...

Huffman coding

- One of the “classic” algorithms in CS.
- Usually taught in introductory CS classes.
- **Input**: Occurence counts (=*frequencies*) of symbols.
- **Output**: Optimal binary code for that distribution.
  - No binary code can do better.
- Won’t describe the algorithm, look it up when interested.
- More important for us: type of *redundancy* exploited.
  - More frequent symbols get shorter codes.
  - Remember Delta coding? “Values cluster around zero”.
  - Huffman codes are great for that.

Arithmetic coding (1)

- Improvement compared to Huffman codes.
- "Didn't you say Huffman codes are optimal?"
- Partially true: They're *optimal binary codes*.
- But we can do better than binary codes.
- Key problem: binary codes always allocate *whole* bits.
- How can one use partial bits?

Arithmetic coding (2)

- Say you have a string of three symbols $\mathrm{a,b,c}$.
- All three are equally likely.
- Huffman code: $\mathrm{a} \mapsto 0$, $\mathrm{b} \mapsto 10$, $\mathrm{c} \mapsto 11$.
  - Or some permutation of that.
- String is $3n$ characters long.
  - $n \times$ 'a', $n \times$ 'b', $n \times$ 'c'.
- Coded string is $\underbrace{n}_{n\times\text{'a'}} + \underbrace{2n}_{n\times\text{'b'}} + \underbrace{2n}_{n\times\text{'c'}} = 5n$ bits long.

Arithmetic coding (3)

- Now let's try something different:
- First, assign (decimal) numbers to our characters.
  - $\mathrm{a} \mapsto 0$, $\mathrm{b} \mapsto 1$, $\mathrm{c} \mapsto 2$.
- Then, we can stuff 5 characters into each byte:
  - Character values are c_1 to c_5.
  - Byte value is $c_1 + 3c_2 + 3^2 c_3 + 3^3 c_4 + 3^4 c_5$.
  - Uses 243 out of 256 characters — good enough.
  - Decoding is several divisions (or just a table lookup).
- 5 characters per 8 bits $\Rightarrow 8/5 = 1.6$ bits/character.
- So $1.6 \cdot \underbrace{3n}_{\text{String length}} = 4.8n$ bits for the whole string.
- *That's* how you get fractional bits.

Arithmetic coding (4)

- To summarize: Idea is to treat our bytestream as a *number*.
- No fixed code assignments, just arithmetic on that number.
  - Different arithmetic coding algorithms are about doing this efficiently.
  - Most are still quite slow compared to binary codes.
- You're not working with code strings, but *probabilities*.
- Unlike Huffman, it's no big deal to change probability distribution.
  - "Adaptive Arihmetic Coding".
  - How accurate that distribution is determines compression ratio.
- Can also use different models based on *context*.
  - We'll explore that idea further in part 2.

End of part 1

Questions so far?

Outline (part 2)

4. Dictionary Schemes and Context Models
   - Dictionary methods
   - LZ77 and variants
   - Context modeling

5. Reordering case studies
   - Operator systems
   - x86 Opcode reordering

Outline

4. Dictionary Schemes and Context Models
   - Dictionary methods
   - LZ77 and variants
   - Context modeling

5. Reordering case studies
   - Operator systems
   - x86 Opcode reordering

Dictionary methods (1)

- In most types of data, there are many repetitions.
  - Lots of small 2- or 3-byte-matches.
  - A few really long ones.
- So how can we make use of that?
- **Idea**: Code strings with a *dictionary*.
  - Not the bulky hardcover kind :)
  - Built from data preceding current byte.
- If current input matches with dictionary...
  - We store a reference to the dictionary.
  - Usually smaller than the string itself.

Dictionary methods (2)

- Dictionary can be explicit or implicit.
  - Explicit: Algorithm maintains a list of strings.
  - Implicit: Well, anything not explicit :)
- Dictionary is *not* stored along with data!
  - Instead, it's being built on the fly from (de)coded data.
- Explicit dictionary methods are quite dead.
  - Implicit is easier to combine with other algorithms.
  - All good compression methods combine several techniques.
- But for completeness...
  - LZW is an explicit dictionary scheme.
  - Used in Unix `compress`, GIF format, V.34 protocol.
  - Used to be patented (now expired)
    - GIF trouble...
  - Not popular anymore.

LZ77 and variants (1)

- Abraham Lempel and Jakob Ziv, 1977.
- Basis for...
  - APack (DOS 4k Packer), NRV (used in UPX).
  - with Huffman coding: LHA, ARJ, ZIP, RAR, CAB.
  - with Arithmetic coding: LZMA (7-Zip, MEW, UPack), kkrunchy
  - Lots more ...
- *Everyone* has used this already.
- Uses a *sliding window* of fixed size.
  - e.g. ZIP: last 32 KB seen.
  - Matches for the current input string are searched in that window.
  - Obviously, longer matches are better (but longest not necessarily best).
  - If no match, code current input character as-is.

LZ77 and variants (2)

- Result is a sequence of “literal” or “match” tokens.
  - Compare to packet-based RLE.
- This token sequence gets coded:
  - Plain LZ77 uses fixed-size codes for everything.
  - LZSS: “match” bit, offset+length if match, else raw symbol.
  - ZIP: Huffman coding on top.
  - Newer algorithms: Even fancier coding.
- Take-home lesson: you want it to find long matches.
  - So make data repetetive.
  - One of the reasons reordering is helpful.

Context modeling

- A class of modeling techniques for Arithmetic coding.
- **Idea**: Predict next symbol based on *context*.
  - "Context" here: last few characters seen.
  - Collects statistics for different-length contexts...
  - ...then builds an overall prediction from that *somehow*.
- That *somehow* is the main difference between algorithms.
  - Plus the details of collecting statistics.
- That paradigm has produced some of the best compressors out there.
  - and some of the slowest, too :)
- Modeling details are rather technical...
  - ...but not that important to us anyway.
  - As "user", treat it like you would LZ77+Arithmetic coding.

Outline

4. Dictionary Schemes and Context Models
   - Dictionary methods
   - LZ77 and variants
   - Context modeling

5. Reordering case studies
   - Operator systems
   - x86 Opcode reordering

Operator graphs

- You've seen this if you ever used a FR demotool.
- Others use a similar representation internally.
  - Even if the user interface is quite different.
- Successive operations on data are stored with their parameters.
  - Nothing special so far, other apps do this too (Undo).
  - But our operation history is a tree.
    - More precisely, a DAG (directed acyclic graph).
- That operation history is then stored in the intro.
  - So let's make it pack well.

Storing operator graphs

- First, standard techniques for trees apply.
  - More precisely, postorder traversal.
    - Writing out operator type IDs as you go.
  - As said, not technically a tree, but a DAG.
  - Pretend it's a tree, use escape codes for "exceptions".
- That's the most efficient way to store the graph *structure*.
  - But what about the operator parameters?
- Straightforward way: Together with op type in graph.

Optimizing operator storage

- kkrieger beta dataset: 29852 bytes (packed).
  - With the simple encoding mentioned above.
- But: Ops of same type tend to have similar data.
- We can exploit that:
  - Store all ops of the same type together.
  - We need to seperate tree and op data for that.
- With those changes: 27741 bytes.
  - Or about 7% gain in compression ratio.
  - Changes to “loading” code in intro are trivial.
  - No huge gain, but an easy saving.

x86 Code

- Source code is usually highly structured.
- The compiled x86 Code looks quite unstructured.
  - When viewed in a Hex Editor.
- Possible explanations:
  1. Compiling inherently destroys that structure.
  2. x86 instruction encoding hides it.
- Both are true in part.
  - We can't do anything about the former, so concentrate on the latter :)
- There'll be some x86 assembly code in this section.
  - If you can't read that, just ignore it.

x86 Code example

```
e8 6a 13 02 00     call   sCopyMem4
8b 4b 1c           mov    ecx, dword ptr [ebx+1Ch]
83 c4 0c           add    esp, 0Ch
8b c5              mov    eax, ebp
d1 e8              shr    eax, 1
8b f5              mov    esi, ebp
8d 14 40           lea    edx, dword ptr [eax+eax*2]
8b 44 24 1c        mov    eax, dword ptr [esp+1Ch]
83 e6 01           and    esi, 1
8d 3c 96           lea    edi, dword ptr [esi+edx*4]
39 44 b9 1c        cmp    dword ptr [ecx+edi*4+1ch], eax
75 77              jne    short L15176
⋮
```

Opcode+ModRM — Jump Offset — Displacement — SIB — Immediate

x86 Code observations

- Not that systematic.
  - Original x86 set was 16bit and had less instructions.
  - Lots of extensions, hacks to for 32bit, etc.
- Different types of data in one, big stream.
  - Actual instruction opcodes
  - Addresses
  - Displacements
  - Immediate operands
  - etc.
- Completely different distributions of values in stream!
- How can we improve on that?

Making x86 code smaller

- Same approch as before:
  - Try to group related data.
- We *don't* want to store elaborate metadata for decoding!
  - Overhead!
  - Can't we get away without?
- **Idea**: Stay with x86 opcode encoding.
  - Processor can decode that $\Rightarrow$ we can, too.
  - But: We split into several streams (again).
- Needs a disassembler.
  - To find out which data is which.
  - And determine instruction sizes.
- Sounds like total overkill.
  - But a simple table-driven disassembler fits in 200 bytes.
  - (Plus 256 bytes of tables)

Further processing

- Another interesting point: `call` instructions.
  - Used to jump into subroutines.
- `call` instruction format:
  - 1 byte opcode.
  - 4 bytes relative target offset.
    - Relative so programs can be moved in memory easier.
- But: relative addressing hurts compression!
  - One function may get called several times.
  - *Relative* offsets are different each time!
- So: replace it with absolute offsets.
  - Improves compression by a few percent (I don't have numbers).
  - Used by UPX and other EXE packers.

Compressing call offsets (1)

- We take that one step further:
  - A lot of functions get called several times.
  - Coding the offset every time is inefficient.
  - ⇒ Keep a list of recent call locations.
  - If offset is in the list, code list index instead of offset.
- Saves about 2% code size on average.

Compressing call offsets (2)

- But we're not through yet:
  - Visual C++ pads spaces between functions with int3 (Breakpoint) opcodes.
  - To trigger a debugger should execution ever flow there...
  - Not an opcode you use in normal code flow!
- How is that useful?
  - We add all memory locations just after a string of int3 opcode to the call list.
  - Hoping that they actually *are* functions.
  - Worst case, that entry will be unused, so no problem there.
  - But we can usually "predict" call targets very succesfully like that.
- Does it really help?
  - Well, another 1.5% improvement :)

Wrapping it up

- I gave lots of rough introductions to compression algorithms.
- Don't trip over details, just try to get the big picture.
  - What type of redundancy is it trying to exploit.
  - What works well, what doesn't.
- Transforms are the key.
  - Can make a huge difference.
  - Don't overdo it.
  - Simple is best.
- Reordering very powerful for LZ/Context-based schemes.
  - Grouping data by structure.
  - Optionally delta-coding etc. afterwards.
  - Measure, measure, measure!
- Examples are some of the things I did.
  - I hope the ideas came through.
  - Heavily dependent on type of data.
  - Your mileage may vary :)

That's it!

Questions?

ryg (at) theprodukkt (dot) com
http://www.farbrausch.de/~fg