Encoding literals in a portable byte-stream interpreter

Reuben Thomas

18th October 1994

Abstract

Three approaches to the problem of encoding literal numbers and addresses in a portable byte-stream virtual machine interpreter are discussed, and one recommended for the case where the interpreter is written in machine-code (or optimally compiled). It is assumed that the instruction fetch loads a word of instructions from memory, a useful optimisation on modern processors. Analysis of the time and space costs for the interpreter program are made with reference to the ARM and 68000 processor series.

1 The problem

In the design of a portable byte-stream virtual processor the method of encoding programs is fixed, so no optimisation can be performed for specific machines. However, it is still necessary to design for optimum execution time and space requirement. Although efficient space usage is less important in modern hardware processors than in previous generations, it is still vitally important for interpreters, as one of their largest overheads is the time taken to perform instruction fetches, because memory references are typically among the most expensive instructions in modern systems.

In a byte-stream interpreter, the instruction codes are usually all one byte long, as this allows a maximum of 256 different instructions, which can be coded in an interpreter small enough to fit in a hardware cache, and hence greatly improve performance. (Smaller than one-byte codes are not byte-stream codes!) Optimisations here are bound up with the design of the processor, and are difficult to treat generally. So we are left with operands. Again, some of these are of fixed length, such as register numbers, and their optimisation interacts with the processor's design. Two, however, may not be of fixed length: relative addresses, and numbers. I shall refer to these collectively as literals.

This paper deals with the optimisation of the encoding of literals in order to minimise the space spent on their storage, and the time spent on instruction and operand fetch. We assume that instructions and immediate operands are mixed, which seems to be the dominant method, and use a simple table-lookup dispatch method which is short enough to be appended to the end of every instruction action routine. This is a common method, which increases the speed of the interpreter at the expense of extra code. However, this usage is not central to the optimisation methods discussed. We further assume that a word is four bytes. The methods discussed would need modification for larger word sizes; the first two may be used for two-byte word machines. On an eight-bit processor, space rather than time optimisation is usually the prime concern, and a simple system with a different opcode for each literal size would be used, with literals being stored directly after their instructions. Of course, extremely common literals such as 0 and -1 may usefully be encoded in the instruction itself.

The discussion is based on optimal machine code, and the idiosyncrasies of particular compilers are not taken into account. It is the author's opinion that interpreters should only be written in a high-level language to facilitate porting to machines on which they do not have a native implementation, and possibly as a starting point for such an implementation. Unless one is prepared to spend a great deal of time studying a particular compiler, or has a particularly good compiler, compiled interpreters will perform much worse than those which are hand-coded.

2 Possible solutions

The most obvious optimisation to make is to allow literals to be stored in different amounts of space. This is most effective with address literals if they are relative addresses, and this will be assumed in the discussion; the same optimisations can be applied to absolute addresses (without the need to handle the sign), but will probably be less useful. Signed numeric literals are probably necessary, although negative numbers are less common than negative relative addresses, and small numbers are more common than small absolute addresses in programs.

If literals are to occupy variable amounts of space, it is most sensible to let them vary in multiples of one byte up to a machine word. As they fit on byte boundaries, they do not interfere with decoding (because instructions and operands do not cross word boundaries), or waste space by leaving bits unused. Longer literals may as well be multiples of one word long, as they are rare and so do not waste much space, neither do they then incur decoding penalties. They may then be treated for the purposes of optimisation as a series of one-word literals, and in any case there is little point optimising them because of their rarity.

Similarly, any literals of one word or longer are best aligned on word boundaries, as on most processors this will result in faster loading, as literals spanning word boundaries have to be loaded as two words and 'stapled' together.

2.1 The base solution

The simplest time optimisation is to read instructions until a literal is needed, then to fetch it from the next word-aligned address, and to continue reading instructions from the word after that. This is obviously inefficient in space, and we will look at ways to plug the gaps without greatly slowing down the literal fetch.

2.2 Simple word-aligned literals

This method works like the naïve solution, except that instructions are always stored in the next free byte, so that no gaps are left in the code. All literals are a word long.

There are two obvious implementations of this method: first, a pointer may be kept to the next word which could hold a literal; second, the interpreter fetches a word of instructions at a time.[1] This leaves the instruction pointer free to point at the next potential literal. We use IP to denote the register holding the instruction pointer, I the register holding the instruction code, LP the literal pointer of the first scheme, and A the instruction accumulator of the second (where each word of instructions is loaded). T denotes the base address of a list of the addresses of the instruction routines; X denotes a temporary register.

The second scheme presents a further choice: how to tell when A is empty. A loop might be used (even unrolled) to count the number of instructions left; however, such a dispatcher is not short enough to put after every instruction in the interpreter. Our solution is to use one or two extra opcodes which perform an instruction fetch. This results in compact code with little time penalty.

The code for the first scheme looks like this:

[1] This may be a good optimisation, as many modern processors can only perform word-aligned fetches, and even on those which can fetch single bytes, it is as fast to fetch a whole word.

68000		ARM	
`move.b (IP)+,I`	get opcode	`ldrb I,[IP],#1`	get opcode
`move.b IP,X`	see if this is the last	`tst IP,#3`	if this is the last instruction
`and.b #3,X`	instruction in the		in the current word,
`cmp.b #0,X`	current word	`moveq IP,LP`	copy LP to IP and
`bne.s skip`	if not, skip add else	`addeq LP,LP,#4`	make LP point to the next word
`move.l LP,IP`	copy LP to IP and	`ldr pc,[T,I,asl#2]`	and dispatch instruction
`adda.w #4,LP`	make LP point to the		
`skip:`	next word		
`asl.w #2,I`	use opcode for table		
`jmp 0(T,I)`	lookup, and dispatch		
	instruction		

On the 68000, the code takes 76 cycles if the branch is taken, and 62 if not (65.5 on average); on the ARM, the code always takes 9 cycles.[2] The code occupies 28 bytes on the 68000 and 24 bytes on the ARM. Adding this code to every instruction adds 7 Kb to the interpreter on the 68000, and 6 Kb on the ARM. This is not acceptable.

The code for the second scheme is:

68000		ARM	
`lsr.l #8,A`	get next instruction	`mov A,A,lsr #8`	get next instruction
`move.b A,I`		`and I,A,#&FF`	
`asl.w I, #2`	make offset and	`ldr pc,[T,I,asl#2]`	and dispatch it
`jmp 0(T,I)`	dispatch instruction		

In addition, extra code is needed for opcode 0:

68000		ARM	
`move.l (IP)+,A`	get next four opcodes	`ldr A, [E], #4`	get next four opcodes
`move.b A,I`	get the first opcode	`and I,A,#&FF`	get the first opcode
`asl.w I,#2`	make offset and	`ldr pc, [T,I,asl#2]`	and dispatch it
`jmp 0(T,I)`	dispatch instruction		

The code for opcode 0 is executed every fourth instruction in addition to the normal dispatch code. On the 68000, the normal dispatch code takes 44 cycles, and the additional code 36, giving 53 cycles on average; the ARM code takes 5 cycles per instruction with an additional 6 for opcode 0, giving an average of 6.5 cycles. The 0 opcode method also makes fewer memory references to the program being interpreted, giving an additional speed advantage. Also, most modern processors have a barrel shifter, unlike the 68000, giving even faster execution.

The space requirements are much lower than for the other method, which also uses an extra permanent register, and an extra temporary register. For the 0 opcode method, only 3 Kb is added to each interpreter. Since the individual instructions will typically also be extremely short, this may be acceptable.

2.3 Word-aligned literals with end-of-word fillers

By extending the instruction set of the interpreter and making a slight change to the dispatch code given above, we can significantly reduce the space occupied by interpreter code.

The change is to allow literals to be stored directly after the instruction opcode whose parameter they are, taking up the remainder of the instruction word. The space available will range from one to three bytes. To do this, two opcodes are needed for every instruction which has a

[2]Some of the cycles will take different times depending on cache state; for simplicity we assume they all take the same time, i.e. that all memory references are found in the cache; this is not unreasonable, as we hope that the interpreter will be held mainly in the cache. References to the program being interpreted will be slower; this is taken into account in the discussion.

literal argument. If there are not enough spare opcodes, it may be worth only doubling those which are used most frequently.

The dispatch code is exactly the same as before, except that if signed literals are required, arithmetic rather than logical shifts must be used. On the two processors used here, and on most other processors, this results in identical timings for the code given (which should have lsr changed to asr). The code for the version of the instruction which takes the literal from the instruction word must now include the opcode 0 case dispatch code. It can access the literal merely by shifting the I register eight bits right (arithmetically or logically as appropriate).

Apart from its effect on the opcode allocation and a slight increase in the size of the interpreter, the only restriction of this method is that all literals must either be signed or unsigned. Thus it is much more useful with relative addressing.

2.4 Variable-size literals

The logical extension of the method in 2.3 is to use separate opcodes for literal sizes from one to four bytes long. However, this is not sensible. Since decoding literals from the middle of an instruction word is slower than decoding those stored at the end, the opcodes introduced in 2.3 should be used whenever this occurs, i.e. for all 3-byte offsets, half of the 2-byte offsets, and one third of the 1-byte offsets. Thus the opcode for 2-byte offsets is used in only a quarter of the cases where the offset could be fitted into two bytes. To decode a 1-byte literal the following code is used:

68000		**ARM**	
`move.b A,T`	get literal	`mov T,A,lsl#24`	get literal
`ext.w T`	sign extend to two bytes	`mov T,T,asr#24`	sign extend it
`ext.l T`	sign extend to a word	`mov A,A,asr#8`	ensure literal is not executed
`asr.l #8,A`	ensure literal is not executed		

This code takes three cycles on the ARM, two more than the shift required to access a literal stored at the end of an instruction word, and 20 cycles on the 68000, 12 more than the end-of-word case.

3 Evaluation

All three methods give both time and speed benefits over the base method of 2.1. Each is suited to different conditions, depending on the type of addressing used. 2.2 is probably the most sensible to use for absolute addressing unless the address space is small, as it the extra optimisations of the other methods will be rarely used, waste valuable opcodes and increase the size of the interpreter unnecessarily. Method 2.3 is probably best if absolute addressing relative to a base register is used, as then most of the addresses will probably fit into three bytes. Method 2.4 may be best for relative addressing, but it is not clear whether the speed loss in processing 1-byte literals will be offset by the compactness of the code. Also, it makes the interpreter larger and uses more opcodes; depending on the processor design and the size of the machine cache on which the interpreter is run, 2.3 may be more efficient.

Finally, there is no reason why two methods should not be combined, and one used for numeric literals while the other is used for addresses. With numeric literals, the advantage of compactness is much greater, since most constants used in programming are small. Especially if method 2.2 is used for address literals, another method can usefully be used to deal with numeric literals, which are often negative. Alternatively, different opcodes could be used for positive and negative numbers.

The author's preference is for method 2.3. Assuming relative addressing is used, which seems sensible in an interpreter precisely so that the advantage of compactness can be gained, method 2.3 offers a significant increase in speed and compactness of code over naïve methods, but with minimal impact on the design and size of the interpreter. If the best method is required, the author suggests that both methods 2.3 and 2.4 be tested to determine which works better in practice.

4 Conclusion

Now that the dominance of eight-bit processors has ended in mainstream computing, it is time to reassess the best ways of building interpreters (and perhaps whether they ought to be built at all; this paper presupposes that they should be). With a little thought it is easy to find optimisations which are applicable to a wide range of modern architectures; a few have been discussed here. It is important to remember that while program size is not as important in RISC processor design as it used to be when address spaces were smaller and memory slower, it is still crucial for the software interpreter which cannot take advantage of hardware pipelining and parallel operation as machine code can.

5 Acknowledgements

Some of the ideas explored here were first pointed out to me by Martin Richards, who also read and criticised the paper.