# An Instruction Stream Compression Technique

P. Bird and T. Mudge

**CSE-TR-319-96**

November 1996

**Computer Science and Engineering Division**
Room 3402 EECS Building

THE UNIVERSITY OF MICHIGAN

Department of Electrical Engineering and Computer Science
Ann Arbor, Michigan 48109-2122
USA

# An Instruction Stream Compression Technique

Peter L. Bird
Trevor N. Mudge

EECS Department
University of Michigan



## Abstract

The performance of instruction memory is a critical factor for both large, high performance applications and for embedded systems. With high performance systems, the bandwidth to the instruction cache can be the limiting factor for execution speed. Code density is often the critical factor for embedded systems.

In this report we demonstrate a straightforward technique for compressing the instruction stream for programs. After code generation, the instruction stream is analysed for often reused sequences of instructions from within the program's basic blocks. These patterns of multiple instructions are then mapped into single byte opcodes. This constitutes a compression of multiple, multi-byte operations onto a single byte. When compressed opcodes are detected during the instruction fetch cycle of program execution, they are expanded within the CPU into the original (multi-cycle) set of instructions. Because we only operate within a program's basic block, branch instructions and their targets are unaffected by this technique.

We provide statistics gathered from code generated for the Intel Pentium and the Power PC processors. We have found that incorporating a 1K decode ROM in the CPU we can reduce a program's code size between 45% and 60%.

# 1. Introduction

Although it is usually true that a particular algorithm can be most efficiently implemented using hand-tuned assembly code, both the overhead in development, and the software life-cycle has made the use of high level languages essential for large-scale, evolving applications.

Compilers cannot truly *optimize* the generated code for any given program. This "sub-optimality" is manifest through larger code size and slower execution times than would be the case for hand crafted assembly code. The impact of slower execution times is readily apparent, and years of research have evolved better and more sophisticated optimization techniques for compilers. However, the *size* of the executing program also can have a severe impact upon program performance.

Embedded systems often use processors which have small addresses spaces for programs. Within this programming domain, the size of the program is often a limiting factor. Developers currently use assembly code to ensure applications fit within the program space constraints.

In high performance environments, program size is also a limiting factor. Most processors have an *instruction cache* for holding recently used program segments. The larger the program, the less likely it is that significant portions of its code will co-reside in the I-cache. An I-cache *miss* interrupts program execution while the missing code fragment is loaded from main memory, thereby reducing overall performance.

Compressing a program's instruction stream would benefit a range of application domains both in the *size* of the program in memory, and in the *bandwidth* requirements for instruction fetch. In this report, we present a technique which addresses this issue.

For the remainder of this section, we will discuss patterns found in compiled programs. Section 2. gives a description of the hardware mechanism which implement the expansion of compressed programs. Section 3. presents the results of several experiments we performed on instruction pattern collection in a range of programs on different architectures. In Section 4. we provide our conclusions and discuss some future work. Raw data for our experiments can be found in the Appendix.

## 1.1 Instruction Patterns in Compiled Software

Compilers generate code using a *Syntax Directed Translation Scheme* (SDTS). [Aho-86]. Syntactic source code patterns are mapped onto templates of instructions which implement the appropriate semantics. Consider, for example, a schema to translate a subset of integer arithmetic:

```
expr -> expr '+' expr
            {   emit( add, $1, $1, $3 );
                $$ = $1;
            }

expr -> expr '*' expr
            {   emit( mult, $1, $1, $3 );
                $$ = $1;
            }
```

In both of these patterns, the expression subtrees on the RHS of the productions return registers which is used by the arithmetic operation. The register number holding the result of the operation ($1) is passed up the parse tree for use in the parent operation.

Compilers reuse instruction templates throughout all generated programs. The only difference in instruction sequences are the register numbers in arithmetic instructions and operand offsets for *load* and *store* instructions. As a consequence, object modules are generated with many common sub-sequences of instructions. There is a high degree of redundancy in the encoding of a program.

The emergence of RISC architectures exacerbates this problem. A RISC instruction set generally trades efficient encodings for efficient decoding. That is, instructions have fixed width (i.e. always 4 bytes). Those fields which are not required for certain operations are wasted.

Newer programming languages encourage the increase of redundancy in programs. One example of this trend can be seen in applications developed using Object Oriented programming techniques. *Information hiding* is one organizational strategy (among many) used for OOP. The implementation of an object is hidden within the private namespace of the class, with member functions used as the interface to the object. Often, these member functions are simple access routines which reference private data structures. These short code sequences are also pattern templates, similar to the SDTS of a compiler.

# 2. Description of Code Compression Technique

After code generation and register allocation, we analyse the generated code stream to search for patterns. This pattern search is made over all instruction sequences of all basic blocks of the program[1]. All patterns of all lengths are compared. The pattern checker finds all distinct patterns, and counts their frequency of occurrence throughout the code stream. Those patterns with the highest frequency of usage are assigned an opcode, and the sequence of instructions for that opcode is saved in a ROM in the CPU. This decode table could be included as a part of the process state, much like a page table.

1. A basic block starts at a label (or after a branch operation) and ends with a branch operation (or another label).

During instruction fetch, the decoder checks the opcode of the incoming instruction. If the opcode indicates an uncompressed instruction, then instruction decode and execution proceeds in a conventional fashion. When the decoder encounters a compressed instruction, the entire sequence of instructions is retrieved from the ROM and dispatched through the execution pipeline one instruction per cycle. Instruction fetch from the program memory stream is stalled until the sequence completes.

## 2.1 Discussion

Perhaps the biggest limitation of the implementation described above is that it appears to require a *pre-decode* phase of the instruction pipeline of a processor (to determine if the instruction is compressed or not). This limitation might be ameliorated if the processor maintains a short instruction queue whose members were checked for compression. This check could be performed during the μ-op expansion phase of the Pentium-Pro instruction pipeline [Micro-95].

This does not help the case where the compressed instruction is the target of a branch instruction. In this case, if the target instruction were held in the ROM table (following the branch instruction), *pre-decode* could be accomplished without incurring a delay in the instruction pipeline.

Although the ISA form of instructions can be held in the ROM table (resulting in a smaller table), a post-decoded form of the instruction could be used to speed instruction processing.

In our model the ROM table is specific to an application. When this is feasible, it results in the best compression behavior. The disadvantage, of course, is that the processor is thereafter configured to that application. For embedded processors this is not a limitation, since a single application is usually 'burned' into a ROM and run for the lifetime of the system. This is obviously unsatisfactory for more general systems. There are alternative models for implementation.

As mentioned above, the decode table could be made a part of the state of a process (much like the page tables capture the virtual memory structure of a process). This would allow the ROM to readily change for different applications.

A library of sequences could become a part of the ISA for a given implementation of an architecture machine. This could permit a manufacturer to maintain a common ISA for all machines, yet 'tune' a specific processor implementation for a given application domain (such as multi-media). Compilers (or post-compilation processors) could translate instruction sequences into the opcodes of the library for the implementation.

The ROM table itself represents a significant cost in silicon area in the CPU. There are several ways one could implement the ROM:

- Use a fixed width table (i.e. all instruction sequences are the length of the longest sequence).

Clearly, this is wasteful of ROM space, since short sequences will have unused slots.

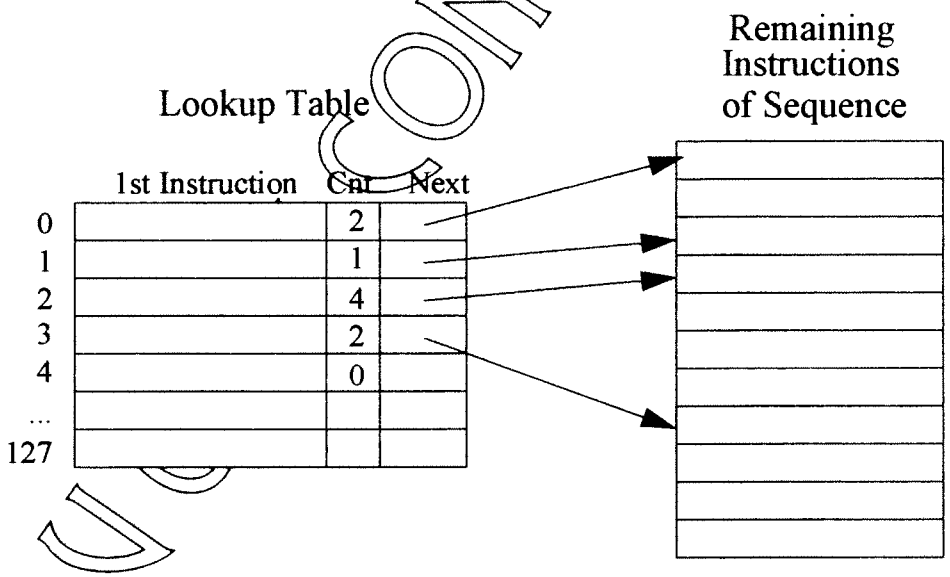
- Partition the ROM table into sets of sequences. The first N slots would be 2 instruction lengths, the next M slots would be 3 instruction lengths, etc.

Although this is less wasteful than the first scheme, it would still be wasteful, since it is unlikely that any given set of patterns would fit neatly into the assigned partitions.

- Use an *index* table to indicate starting ROM position and sequence length.

This would more efficiently use space than the first two schemes, but would require two ROM accesses before the first instruction would be available for execution. This is likely to be an unacceptable overhead (especially in high performance applications).

We need to both minimize its size, and minimize the cost of looking up instructions. We can use a form of the preceding model, where the 1st instruction of the sequence is a part of the index lookup. A compressed opcode is used to index into the *Lookup Table*. The first instruc-




tion of the sequence is drawn from the *lookup table* and sent to the instruction execution pipeline. The *count* of the remainder of the sequence is drawn from the table and using the index of the *next* instruction, the remaining instructions of the sequence are retrieved from a secondary ROM table.

In the simplest case, even single instruction encodings are effective. Consider the following program fragment:

```
for (i = c1; i < n; i++)
    stmt1
    ...
for (j = c2; j < n; i++)
    stmt2
```

Each loop would contain the instruction sequence.

```
Load    Rn,offset{i,j}(Rbase)
Addi    Rn,=1
Store   Rn,offset{i,j}(Rbase)
```

which loads the appropriate iteration variable, increments it and stores it back onto the program's stack frame. Since the `add immediate` operation is common to the two sequences, it's a candidate for compression. If our target architecture uses 16 bit encodings of instructions, we will save 2 bytes (16 bits) out of 12 bytes (~17%) by compressing only the single instruction.

When we permit compression of sequences of instructions the results are more dramatic. If both loops use the same iteration variable (whether on the stack or held in a register), then the two increment sequences would be equivalent, and could be reduced to a single byte. The two 6 byte sequences would be reduced to 2 bytes (~83% savings).

We have intentionally ignored the issue of optimizations in this example. For example, in both loop instances the iteration variable could be assigned to a register and never written back to the stack frame. However, even with optimizations the compression technique discussed here are applicable.

Further compression would be possible were the compiler aware that the optimization was available, and generated a program's code sequences accordingly.

Large register sets are disadvantageous to this technique. Sequence patterns are formed by looking only at the distinct bit patterns of instructions (and not the operations being performed). Using different registers for the same sequence of operations would result in distinct patterns. However, this issue didn't seem to have the impact we expected as our experimental results show for the Power PC (Section 3.).

All of the discussion thus far has focussed upon the *static* characteristics of programs. Although we have not investigated *dynamic* behavior in detail for this paper, it should be clear that dynamic usage counts of instruction traces (collected by a profiler) could be used for pattern construction in conjunction with *static* pattern construction.

As mentioned above, instruction cache pressure is a major constraint for high performance program execution [Uhlig-95]. An I-cache miss will stall the processor, while the instruction memory access bandwidth limits the rate at which new instructions can be delivered to the processor. Because this scheme reduces the size of programs, it effectively increases the effective

size of an I-cache for the same program fragment (a code stream is dynamically expanded within the CPU rather than redundantly occupying precious cache lines). Moreover, since fewer bytes are transferred from program memory to the I-cache, the instruction memory bandwidth requirements of the program are reduced.

We should note that while this technique is orthogonal to instruction level parallelism techniques, they could be combined (see Section 4. below).

# 3. Experiments

We analysed 3 programs on two types of processors to determine the impact of *static* instruction sequence compression upon program size. The three programs were:

- The GO program from the SPEC benchmarks [SPEC 95].
- A compiler for B# (a C-like programming language used in our compiler class).
- A large module from an image processing workbench [C&G 96].

The GO program was compiled with the GCC compiler for a Power PC. The other two programs were compiled using Microsoft's Visual C++, version 4.1, for the Intel Pentium. The compiler is written using only C language features (within the framework of LEX and YACC), while the image processing module draws very heavily from the object oriented components of C++, such as overloading, member access functions, virtual functions, etc.

Although all programs displayed long instruction sequence patterns (up to 20 instructions in the image processing module), we restricted our pattern search to sequences from 1 to 8 instructions. The data for the three programs is summarized in the following table. The sizes of the ROM tables include only the raw instruction sequences and do not incorporate all fields described in Section 2.1 above. Code sizes include the size of the ROM table along with the compressed program.

| | GO | Compiler | Image |
|---|---|---|---|
| Processor | Power PC | Pentium | Pentium |
| Basic Blocks | 10213 | 2044 | 4686 |
| Uncompressed Bytes | 219236 | 32423 | 60273 |
| Total Instructions | 54809 | 8919 | 18815 |
| Distinct Instructions | 13737 | 1914 | 3978 |
| Distinct Patterns (length 1 to 8) | 130952 | 15823 | 23937 |
| Code size after compressing 1st 16 Patterns | 69.8% | 76.1% | 62.5% |
| ROM size (in bytes) for 1st 16 Patterns | 464 | 147 | 124 |
| Code Size using 1K of ROM | 55% | 52.6% | 40.3% |
| Code size after compressing 1st 128 patterns | 43.3% | 49.3% | 40.1% |
| ROM size (in bytes) for 1st 128 patterns | 1920 | 1613 | 1039 |

## 3.1 Discussion

The results of our analysis are encouraging. Compressing only 16 patterns and using a very modest ROM size, we were able to reduce programs by 24% to 38% of their original size. Using a 1K ROM tables, programs were compressed to between 45% and 60% of the original size. These represent very modest investments in silicon area.

# 4. Conclusion and Directions for Future Work

We have outlined a technique for generating dense program encodings. Our technique exploits exactly those instruction sequence patterns which are repeated within a particular program.

We have discussed (in Section 2.1) ways this technique could be enhanced by adding a pre-decode phase of instruction execution. A more detailed study of the problem is required.

We have shown that there is a high degree of regularity of patterns for different applications compiled on different architectures. It would be interesting to see whether patterns exist which are

- idiomatic to specific applications (integer vs. floating point, data base vs. multi-media, etc.)
- specific to compilers
- idiomatic to machines.

If the latter case is true, then this technique could be useful for any general purpose machine in reducing its instruction fetch bottleneck.

We believe it would be very interesting to examine the *dynamic* behavior of a program when analysing patterns. By incorporating a profiler, this process is no more difficult than collecting *static* patterns. Our ROM table (containing a compressed tag following a branch instruction), could augment or even replace a branch target buffer. Essentially, our compressed instruction would become a *highly-likely* program trace path.

It is interesting to note the relationship of our work with *Trace Scheduling* [Ellis-85], a technique for enhancing program parallelism which was derived from microcode compaction. The focus of trace scheduling is to expose all machine operations to the compiler, thereby increasing the set of operations which can be re-combined for parallel execution. One disadvantage of compiler control of micro-operations is the added cost of program memory required to specify the control of all operations every cycle. Although it is fruitful to "open the box" to expose the possible parallel operations, we believe that a final set of common parallel operations can be distilled to a small number of fixed actions. It would make sense to collapse these operations onto a small set of opcodes thereby reducing the instruction memory requirements of the program.

# Bibliography


| | |
|---|---|
| Aho-86 | A. Aho, R. Sethi and J. Ullman, *Compiler: Principles, Techniques and Tools*, Addison-Wesley, 1986. |
| C&G 96 | Image Processing Workbench, C&G Software Engineering, Ann Arbor, MI. 1996. |
| Ellis-85 | J. Ellis, *Bulldog: A Compiler for VLIW Architectures*, ACM Distinguished Dissertation, MIT Press, 1985. |
| Micro-95 | "Intel's P6 Uses Decoupled Superscalar Design," Microprocessor Report **9(2)**, 16 February 1995. |
| SPEC-95 | Spec Benchmarks, 1995. |
| Uhlig-95 | R. Uhlig, *Trap-Driven Memory Simulation*, Ph.D dissertation, EECS Department, University of Michigan, Ann Arbor, MI, 1995. |

# Appendix

To give the reader an idea of the impact of pattern sequences on program compression, we here present the raw data of our analyser for the 3 programs we studied.

## GO Program

| | |
|---|---|
| 54809 | Instructions |
| 219236 | bytes in program |
| 13737 | distinct instructions |
| 4203 | labels |
| 10213 | basic blocks |
| 130952 | distinct patterns of length 1 to 8 |

| # | Bytes in Pattern | Instructions in Pattern | Use Count | Cumulative Bytes Saved | ROM Table Size | Program Size (vs. Original) | Program Size with ROM Table |
|---|---|---|---|---|---|---|---|
| 1 | 4 | 1 | 1900 | 5700 | 4 | 97.4% | 97.4% |
| 2 | 32 | 8 | 177 | 11187 | 36 | 94.9% | 94.9% |
| 3 | 32 | 8 | 174 | 16581 | 68 | 92.4% | 92.5% |
| 4 | 32 | 8 | 169 | 21820 | 100 | 90.0% | 90.1% |
| 5 | 32 | 8 | 169 | 27059 | 132 | 87.7% | 87.7% |
| 6 | 32 | 8 | 154 | 31833 | 164 | 85.5% | 85.6% |
| 7 | 32 | 8 | 154 | 36607 | 196 | 83.3% | 83.4% |
| 8 | 32 | 8 | 129 | 40606 | 228 | 81.5% | 81.6% |
| 9 | 32 | 8 | 129 | 44605 | 260 | 79.7% | 79.8% |
| 10 | 32 | 8 | 107 | 47922 | 292 | 78.1% | 78.3% |
| 11 | 32 | 8 | 107 | 51239 | 324 | 76.6% | 76.8% |
| 12 | 32 | 8 | 97 | 54246 | 356 | 75.3% | 75.4% |
| 13 | 32 | 8 | 97 | 57253 | 388 | 73.9% | 74.1% |
| 14 | 4 | 1 | 928 | 60037 | 392 | 72.6% | 72.8% |
| 15 | 32 | 8 | 76 | 62393 | 424 | 71.5% | 71.7% |
| 16 | 32 | 8 | 76 | 64749 | 456 | 70.5% | 70.7% |
| 17 | 8 | 2 | 288 | 66765 | 464 | 69.5% | 69.8% |
| 18 | 32 | 8 | 62 | 68687 | 496 | 68.7% | 68.9% |
| 19 | 32 | 8 | 62 | 70609 | 528 | 67.8% | 68.0% |
| 20 | 4 | 1 | 595 | 72394 | 532 | 67.0% | 67.2% |
| 21 | 4 | 1 | 563 | 74083 | 536 | 66.2% | 66.5% |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 22 | 12 | 3 | 150 | 75733 | 548 | 65.5% | 65.7% |
| 23 | 32 | 8 | 46 | 77159 | 580 | 64.8% | 65.1% |
| 24 | 32 | 8 | 46 | 78585 | 612 | 64.2% | 64.4% |
| 25 | 16 | 4 | 85 | 79860 | 628 | 63.6% | 63.9% |
| 26 | 16 | 4 | 82 | 81090 | 644 | 63.0% | 63.3% |
| 27 | 32 | 8 | 37 | 82237 | 676 | 62.5% | 62.8% |
| 28 | 32 | 8 | 37 | 83384 | 708 | 62.0% | 62.3% |
| 29 | 16 | 4 | 75 | 84509 | 724 | 61.5% | 61.8% |
| 30 | 20 | 5 | 59 | 85630 | 744 | 60.9% | 61.3% |
| 31 | 4 | 1 | 372 | 86746 | 748 | 60.4% | 60.8% |
| 32 | 4 | 1 | 352 | 87802 | 752 | 60.0% | 60.3% |
| 33 | 20 | 5 | 55 | 88847 | 772 | 59.5% | 59.8% |
| 34 | 24 | 6 | 40 | 89767 | 796 | 59.1% | 59.4% |
| 35 | 4 | 1 | 294 | 90649 | 800 | 58.7% | 59.0% |
| 36 | 32 | 8 | 27 | 91486 | 832 | 58.3% | 58.7% |
| 37 | 32 | 8 | 27 | 92323 | 864 | 57.9% | 58.3% |
| 38 | 4 | 1 | 278 | 93157 | 868 | 57.5% | 57.9% |
| 39 | 24 | 6 | 36 | 93985 | 892 | 57.1% | 57.5% |
| 40 | 4 | 1 | 270 | 94795 | 896 | 56.8% | 57.2% |
| 41 | 4 | 1 | 267 | 95596 | 900 | 56.4% | 56.8% |
| 42 | 16 | 4 | 53 | 96391 | 916 | 56.0% | 56.5% |
| 43 | 4 | 1 | 231 | 97084 | 920 | 55.7% | 56.1% |
| 44 | 4 | 1 | 219 | 97741 | 924 | 55.4% | 55.8% |
| 45 | 4 | 1 | 214 | 98383 | 928 | 55.1% | 55.5% |
| 46 | 32 | 8 | 20 | 99003 | 960 | 54.8% | 55.3% |
| 47 | 32 | 8 | 20 | 99623 | 992 | 54.6% | 55.0% |
| 48 | 32 | 8 | 20 | 100243 | 1024 | 54.3% | 54.7% |
| 49 | 32 | 8 | 20 | 100863 | 1056 | 54.0% | 54.5% |
| 50 | 32 | 8 | 19 | 101452 | 1088 | 53.7% | 54.2% |
| 51 | 8 | 2 | 82 | 102026 | 1096 | 53.5% | 54.0% |
| 52 | 4 | 1 | 187 | 102587 | 1100 | 53.2% | 53.7% |
| 53 | 4 | 1 | 186 | 103145 | 1104 | 53.0% | 53.5% |
| 54 | 32 | 8 | 18 | 103703 | 1136 | 52.7% | 53.2% |
| 55 | 32 | 8 | 17 | 104230 | 1168 | 52.5% | 53.0% |
| 56 | 32 | 8 | 17 | 104757 | 1200 | 52.2% | 52.8% |
| 57 | 4 | 1 | 165 | 105252 | 1204 | 52.0% | 52.5% |
| 58 | 4 | 1 | 161 | 105735 | 1208 | 51.8% | 52.3% |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 59 | 4 | 1 | 155 | 106200 | 1212 | 51.6% | 52.1% |
| 60 | 28 | 7 | 17 | 106659 | 1240 | 51.3% | 51.9% |
| 61 | 4 | 1 | 152 | 107115 | 1244 | 51.1% | 51.7% |
| 62 | 4 | 1 | 145 | 107550 | 1248 | 50.9% | 51.5% |
| 63 | 4 | 1 | 140 | 107970 | 1252 | 50.8% | 51.3% |
| 64 | 12 | 3 | 38 | 108388 | 1264 | 50.6% | 51.1% |
| 65 | 4 | 1 | 139 | 108805 | 1268 | 50.4% | 50.9% |
| 66 | 4 | 1 | 137 | 109216 | 1272 | 50.2% | 50.8% |
| 67 | 4 | 1 | 135 | 109621 | 1276 | 50.0% | 50.6% |
| 68 | 32 | 8 | 13 | 110024 | 1308 | 49.8% | 50.4% |
| 69 | 32 | 8 | 13 | 110427 | 1340 | 49.6% | 50.2% |
| 70 | 4 | 1 | 134 | 110829 | 1344 | 49.4% | 50.1% |
| 71 | 4 | 1 | 133 | 111228 | 1348 | 49.3% | 49.9% |
| 72 | 12 | 3 | 36 | 111624 | 1360 | 49.1% | 49.7% |
| 73 | 4 | 1 | 130 | 112014 | 1364 | 48.9% | 49.5% |
| 74 | 4 | 1 | 128 | 112398 | 1368 | 48.7% | 49.4% |
| 75 | 4 | 1 | 125 | 112773 | 1372 | 48.6% | 49.2% |
| 76 | 4 | 1 | 123 | 113142 | 1376 | 48.4% | 49.0% |
| 77 | 4 | 1 | 110 | 113472 | 1380 | 48.2% | 48.9% |
| 78 | 4 | 1 | 108 | 113796 | 1384 | 48.1% | 48.7% |
| 79 | 4 | 1 | 105 | 114111 | 1388 | 48.0% | 48.6% |
| 80 | 4 | 1 | 105 | 114426 | 1392 | 47.8% | 48.4% |
| 81 | 4 | 1 | 105 | 114741 | 1396 | 47.7% | 48.3% |
| 82 | 8 | 2 | 45 | 115056 | 1404 | 47.5% | 48.2% |
| 83 | 32 | 8 | 10 | 115366 | 1436 | 47.4% | 48.0% |
| 84 | 32 | 8 | 10 | 115676 | 1468 | 47.2% | 47.9% |
| 85 | 16 | 4 | 20 | 115976 | 1484 | 47.1% | 47.8% |
| 86 | 16 | 4 | 20 | 116276 | 1500 | 47.0% | 47.6% |
| 87 | 4 | 1 | 97 | 116567 | 1504 | 46.8% | 47.5% |
| 88 | 16 | 4 | 19 | 116852 | 1520 | 46.7% | 47.4% |
| 89 | 4 | 1 | 94 | 117134 | 1524 | 46.6% | 47.3% |
| 90 | 4 | 1 | 94 | 117416 | 1528 | 46.4% | 47.1% |
| 91 | 4 | 1 | 93 | 117695 | 1532 | 46.3% | 47.0% |
| 92 | 32 | 8 | 9 | 117974 | 1564 | 46.2% | 46.9% |
| 93 | 32 | 8 | 9 | 118253 | 1596 | 46.1% | 46.8% |
| 94 | 32 | 8 | 9 | 118532 | 1628 | 45.9% | 46.7% |
| 95 | 32 | 8 | 9 | 118811 | 1660 | 45.8% | 46.6% |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 96 | 8 | 2 | 39 | 119084 | 1668 | 45.7% | 46.4% |
| 97 | 4 | 1 | 90 | 119354 | 1672 | 45.6% | 46.3% |
| 98 | 4 | 1 | 90 | 119624 | 1676 | 45.4% | 46.2% |
| 99 | 4 | 1 | 88 | 119888 | 1680 | 45.3% | 46.1% |
| 100 | 4 | 1 | 84 | 120140 | 1684 | 45.2% | 46.0% |
| 101 | 4 | 1 | 83 | 120389 | 1688 | 45.1% | 45.9% |
| 102 | 20 | 5 | 13 | 120636 | 1708 | 45.0% | 45.8% |
| 103 | 8 | 2 | 35 | 120881 | 1716 | 44.9% | 45.6% |
| 104 | 16 | 4 | 16 | 121121 | 1732 | 44.8% | 45.5% |
| 105 | 4 | 1 | 78 | 121355 | 1736 | 44.6% | 45.4% |
| 106 | 4 | 1 | 78 | 121589 | 1740 | 44.5% | 45.3% |
| 107 | 4 | 1 | 78 | 121823 | 1744 | 44.4% | 45.2% |
| 108 | 8 | 2 | 33 | 122054 | 1752 | 44.3% | 45.1% |
| 109 | 12 | 3 | 21 | 122285 | 1764 | 44.2% | 45.0% |
| 110 | 20 | 5 | 12 | 122513 | 1784 | 44.1% | 44.9% |
| 111 | 4 | 1 | 74 | 122735 | 1788 | 44.0% | 44.8% |
| 112 | 4 | 1 | 74 | 122957 | 1792 | 43.9% | 44.7% |
| 113 | 4 | 1 | 73 | 123176 | 1796 | 43.8% | 44.6% |
| 114 | 32 | 8 | 7 | 123393 | 1828 | 43.7% | 44.6% |
| 115 | 4 | 1 | 72 | 123609 | 1832 | 43.6% | 44.5% |
| 116 | 4 | 1 | 72 | 123825 | 1836 | 43.5% | 44.4% |
| 117 | 4 | 1 | 71 | 124038 | 1840 | 43.4% | 44.3% |
| 118 | 4 | 1 | 71 | 124251 | 1844 | 43.3% | 44.2% |
| 119 | 4 | 1 | 70 | 124461 | 1848 | 43.2% | 44.1% |
| 120 | 8 | 2 | 30 | 124671 | 1856 | 43.1% | 44.0% |
| 121 | 12 | 3 | 19 | 124880 | 1868 | 43.0% | 43.9% |
| 122 | 12 | 3 | 19 | 125089 | 1880 | 42.9% | 43.8% |
| 123 | 4 | 1 | 69 | 125296 | 1884 | 42.8% | 43.7% |
| 124 | 4 | 1 | 68 | 125500 | 1888 | 42.8% | 43.6% |
| 125 | 8 | 2 | 29 | 125703 | 1896 | 42.7% | 43.5% |
| 126 | 4 | 1 | 65 | 125898 | 1900 | 42.6% | 43.4% |
| 127 | 16 | 4 | 13 | 126093 | 1916 | 42.5% | 43.4% |
| 128 | 4 | 1 | 64 | 126285 | 1920 | 42.4% | 43.3% |

# B# Compiler

| | |
|---|---|
| 8919 | Instructions |
| 32423 | bytes in program |
| 1914 | distinct instructions |
| 867 | labels |
| 2044 | basic blocks |
| 15823 | distinct patterns of length 1 to 8 |

| # | Bytes in Pattern | Instructions in Pattern | Use Count | Cumulative Bytes Saved | ROM Table Size | Program Size (vs. Original) | Program Size with ROM Table |
|---|---|---|---|---|---|---|---|
| 1 | 5 | 1 | 311 | 1244 | 5 | 96.2% | 96.2% |
| 2 | 5 | 1 | 197 | 2032 | 10 | 93.7% | 93.8% |
| 3 | 11 | 3 | 70 | 2762 | 21 | 91.6% | 91.6% |
| 4 | 8 | 2 | 86 | 3334 | 29 | 89.7% | 89.8% |
| 5 | 8 | 2 | 68 | 3810 | 37 | 88.2% | 88.4% |
| 6 | 3 | 1 | 211 | 4232 | 40 | 86.9% | 87.1% |
| 7 | 22 | 8 | 20 | 4652 | 62 | 85.7% | 85.8% |
| 8 | 8 | 2 | 52 | 5016 | 70 | 84.5% | 84.7% |
| 9 | 5 | 1 | 86 | 5360 | 75 | 83.5% | 83.7% |
| 10 | 5 | 1 | 84 | 5696 | 80 | 82.4% | 82.7% |
| 11 | 13 | 3 | 28 | 6032 | 93 | 81.4% | 81.7% |
| 12 | 11 | 3 | 33 | 6362 | 104 | 80.4% | 80.7% |
| 13 | 3 | 1 | 163 | 6688 | 107 | 79.4% | 79.7% |
| 14 | 6 | 2 | 65 | 7013 | 113 | 78.4% | 78.7% |
| 15 | 19 | 4 | 18 | 7337 | 132 | 77.4% | 77.8% |
| 16 | 7 | 4 | 51 | 7643 | 139 | 76.4% | 76.9% |
| 17 | 8 | 2 | 38 | 7909 | 147 | 75.6% | 76.1% |
| 18 | 5 | 5 | 65 | 8169 | 152 | 74.8% | 75.3% |
| 19 | 10 | 2 | 27 | 8412 | 162 | 74.1% | 74.6% |
| 20 | 3 | 1 | 116 | 8644 | 165 | 73.3% | 73.8% |
| 21 | 24 | 8 | 10 | 8874 | 189 | 72.6% | 73.2% |
| 22 | 5 | 1 | 55 | 9094 | 194 | 72.0% | 72.6% |
| 23 | 9 | 3 | 26 | 9302 | 203 | 71.3% | 71.9% |
| 24 | 15 | 6 | 13 | 9484 | 218 | 70.7% | 71.4% |
| 25 | 35 | 8 | 5 | 9654 | 253 | 70.2% | 71.0% |
| 26 | 5 | 1 | 42 | 9822 | 258 | 69.7% | 70.5% |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 27 | 5 | 1 | 41 | 9986 | 263 | 69.2% | 70.0% |
| 28 | 14 | 3 | 12 | 10142 | 277 | 68.7% | 69.6% |
| 29 | 8 | 2 | 22 | 10296 | 285 | 68.2% | 69.1% |
| 30 | 5 | 1 | 38 | 10448 | 290 | 67.8% | 68.7% |
| 31 | 9 | 6 | 19 | 10600 | 299 | 67.3% | 68.2% |
| 32 | 6 | 2 | 29 | 10745 | 305 | 66.9% | 67.8% |
| 33 | 5 | 1 | 35 | 10885 | 310 | 66.4% | 67.4% |
| 34 | 8 | 2 | 20 | 11025 | 318 | 66.0% | 67.0% |
| 35 | 24 | 8 | 6 | 11163 | 342 | 65.6% | 66.6% |
| 36 | 18 | 5 | 8 | 11299 | 360 | 65.2% | 66.3% |
| 37 | 6 | 2 | 27 | 11434 | 366 | 64.7% | 65.9% |
| 38 | 6 | 2 | 27 | 11569 | 372 | 64.3% | 65.5% |
| 39 | 7 | 3 | 22 | 11701 | 379 | 63.9% | 65.1% |
| 40 | 13 | 3 | 11 | 11833 | 392 | 63.5% | 64.7% |
| 41 | 11 | 2 | 13 | 11963 | 403 | 63.1% | 64.3% |
| 42 | 33 | 7 | 4 | 12091 | 436 | 62.7% | 64.1% |
| 43 | 7 | 2 | 21 | 12217 | 443 | 62.3% | 63.7% |
| 44 | 6 | 1 | 24 | 12337 | 449 | 61.9% | 63.3% |
| 45 | 16 | 5 | 8 | 12457 | 465 | 61.6% | 63.0% |
| 46 | 5 | 1 | 28 | 12569 | 470 | 61.2% | 62.7% |
| 47 | 5 | 1 | 28 | 12681 | 475 | 60.9% | 62.4% |
| 48 | 3 | 1 | 55 | 12791 | 478 | 60.5% | 62.0% |
| 49 | 12 | 2 | 10 | 12901 | 490 | 60.2% | 61.7% |
| 50 | 12 | 4 | 10 | 13011 | 502 | 59.9% | 61.4% |
| 51 | 7 | 1 | 18 | 13119 | 509 | 59.5% | 61.1% |
| 52 | 3 | 1 | 54 | 13227 | 512 | 59.2% | 60.8% |
| 53 | 16 | 4 | 7 | 13332 | 528 | 58.9% | 60.5% |
| 54 | 14 | 4 | 8 | 13436 | 542 | 58.6% | 60.2% |
| 55 | 6 | 2 | 20 | 13536 | 548 | 58.3% | 59.9% |
| 56 | 26 | 6 | 4 | 13636 | 574 | 57.9% | 59.7% |
| 57 | 3 | 2 | 47 | 13730 | 577 | 57.7% | 59.4% |
| 58 | 3 | 3 | 47 | 13824 | 580 | 57.4% | 59.2% |
| 59 | 24 | 7 | 4 | 13916 | 604 | 57.1% | 58.9% |
| 60 | 11 | 2 | 9 | 14006 | 615 | 56.8% | 58.7% |
| 61 | 16 | 5 | 6 | 14096 | 631 | 56.5% | 58.5% |
| 62 | 3 | 1 | 43 | 14182 | 634 | 56.3% | 58.2% |
| 63 | 7 | 2 | 14 | 14266 | 641 | 56.0% | 58.0% |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 64 | 15 | 3 | 6 | 14350 | 656 | 55.7% | 57.8% |
| 65 | 5 | 1 | 20 | 14430 | 661 | 55.5% | 57.5% |
| 66 | 5 | 1 | 20 | 14510 | 666 | 55.2% | 57.3% |
| 67 | 2 | 1 | 78 | 14588 | 668 | 55.0% | 57.1% |
| 68 | 2 | 1 | 78 | 14666 | 670 | 54.8% | 56.8% |
| 69 | 3 | 2 | 39 | 14744 | 673 | 54.5% | 56.6% |
| 70 | 27 | 8 | 3 | 14822 | 700 | 54.3% | 56.4% |
| 71 | 8 | 3 | 11 | 14899 | 708 | 54.0% | 56.2% |
| 72 | 3 | 1 | 38 | 14975 | 711 | 53.8% | 56.0% |
| 73 | 4 | 3 | 25 | 15050 | 715 | 53.6% | 55.8% |
| 74 | 7 | 1 | 12 | 15122 | 722 | 53.4% | 55.6% |
| 75 | 5 | 1 | 18 | 15194 | 727 | 53.1% | 55.4% |
| 76 | 7 | 2 | 12 | 15266 | 734 | 52.9% | 55.2% |
| 77 | 13 | 3 | 6 | 15338 | 747 | 52.7% | 55.0% |
| 78 | 10 | 5 | 8 | 15410 | 757 | 52.5% | 54.8% |
| 79 | 8 | 2 | 10 | 15480 | 765 | 52.3% | 54.6% |
| 80 | 7 | 2 | 11 | 15546 | 772 | 52.1% | 54.4% |
| 81 | 6 | 2 | 13 | 15611 | 778 | 51.9% | 54.3% |
| 82 | 3 | 1 | 30 | 15671 | 781 | 51.7% | 54.1% |
| 83 | 3 | 1 | 30 | 15731 | 784 | 51.5% | 53.9% |
| 84 | 6 | 1 | 12 | 15791 | 790 | 51.3% | 53.7% |
| 85 | 13 | 4 | 5 | 15851 | 803 | 51.1% | 53.6% |
| 86 | 31 | 8 | 2 | 15911 | 834 | 50.9% | 53.5% |
| 87 | 31 | 8 | 2 | 15971 | 865 | 50.7% | 53.4% |
| 88 | 30 | 8 | 2 | 16029 | 895 | 50.6% | 53.3% |
| 89 | 30 | 8 | 2 | 16087 | 925 | 50.4% | 53.2% |
| 90 | 5 | 1 | 14 | 16143 | 930 | 50.2% | 53.1% |
| 91 | 3 | 1 | 28 | 16199 | 933 | 50.0% | 52.9% |
| 92 | 15 | 3 | 4 | 16255 | 948 | 49.9% | 52.8% |
| 93 | 15 | 5 | 4 | 16311 | 963 | 49.7% | 52.7% |
| 94 | 29 | 8 | 2 | 16367 | 992 | 49.5% | 52.6% |
| 95 | 29 | 8 | 2 | 16423 | 1021 | 49.3% | 52.5% |
| 96 | 7 | 1 | 9 | 16477 | 1028 | 49.2% | 52.4% |
| 97 | 10 | 4 | 6 | 16531 | 1038 | 49.0% | 52.2% |
| 98 | 10 | 5 | 6 | 16585 | 1048 | 48.8% | 52.1% |
| 99 | 19 | 5 | 3 | 16639 | 1067 | 48.7% | 52.0% |
| 100 | 19 | 5 | 3 | 16693 | 1086 | 48.5% | 51.9% |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 101 | 19 | 5 | 3 | 16747 | 1105 | 48.3% | 51.8% |
| 102 | 28 | 7 | 2 | 16801 | 1133 | 48.2% | 51.7% |
| 103 | 28 | 7 | 2 | 16855 | 1161 | 48.0% | 51.6% |
| 104 | 14 | 5 | 4 | 16907 | 1175 | 47.9% | 51.5% |
| 105 | 27 | 8 | 2 | 16959 | 1202 | 47.7% | 51.4% |
| 106 | 27 | 8 | 2 | 17011 | 1229 | 47.5% | 51.3% |
| 107 | 18 | 3 | 3 | 17062 | 1247 | 47.4% | 51.2% |
| 108 | 6 | 2 | 10 | 17112 | 1253 | 47.2% | 51.1% |
| 109 | 26 | 8 | 2 | 17162 | 1279 | 47.1% | 51.0% |
| 110 | 5 | 1 | 12 | 17210 | 1284 | 46.9% | 50.9% |
| 111 | 5 | 1 | 12 | 17258 | 1289 | 46.8% | 50.7% |
| 112 | 5 | 1 | 12 | 17306 | 1294 | 46.6% | 50.6% |
| 113 | 13 | 2 | 4 | 17354 | 1307 | 46.5% | 50.5% |
| 114 | 13 | 3 | 4 | 17402 | 1320 | 46.3% | 50.4% |
| 115 | 25 | 5 | 2 | 17450 | 1345 | 46.2% | 50.3% |
| 116 | 25 | 5 | 2 | 17498 | 1370 | 46.0% | 50.3% |
| 117 | 17 | 6 | 3 | 17546 | 1387 | 45.9% | 50.2% |
| 118 | 13 | 7 | 4 | 17594 | 1400 | 45.7% | 50.1% |
| 119 | 25 | 8 | 2 | 17642 | 1425 | 45.6% | 50.0% |
| 120 | 25 | 8 | 2 | 17690 | 1450 | 45.4% | 49.9% |
| 121 | 24 | 8 | 2 | 17736 | 1474 | 45.3% | 49.8% |
| 122 | 24 | 8 | 2 | 17782 | 1498 | 45.2% | 49.8% |
| 123 | 24 | 8 | 2 | 17828 | 1522 | 45.0% | 49.7% |
| 124 | 47 | 8 | 1 | 17874 | 1569 | 44.9% | 49.7% |
| 125 | 6 | 2 | 9 | 17919 | 1575 | 44.7% | 49.6% |
| 126 | 16 | 3 | 3 | 17964 | 1591 | 44.6% | 49.5% |
| 127 | 16 | 3 | 3 | 18009 | 1607 | 44.5% | 49.4% |
| 128 | 6 | 4 | 9 | 18054 | 1613 | 44.3% | 49.3% |

# Image Processing Module

18815 Instructions
60273 bytes in program
3978 distinct instructions
2028 labels
4686 basic blocks
23937 distinct patterns of length 1 to 8

| # | Bytes in Pattern | Instructions in Pattern | Use Count | Cumulative Bytes Saved | ROM Table Size | Program Size (vs. Original) | Program Size with ROM Table |
|---|---|---|---|---|---|---|---|
| 1 | 14 | 3 | 267 | 3471 | 14 | 94.2% | 94.3% |
| 2 | 14 | 6 | 232 | 6487 | 28 | 89.2% | 89.3% |
| 3 | 6 | 4 | 411 | 8342 | 34 | 85.8% | 85.9% |
| 4 | 9 | 5 | 216 | 10270 | 43 | 83.0% | 83.0% |
| 5 | 7 | 1 | 268 | 11878 | 50 | 80.3% | 80.4% |
| 6 | 3 | 1 | 736 | 13350 | 53 | 77.9% | 77.9% |
| 7 | 12 | 7 | 118 | 14648 | 65 | 75.7% | 75.8% |
| 8 | 7 | 1 | 208 | 15896 | 72 | 73.6% | 73.7% |
| 9 | 5 | 1 | 267 | 16964 | 77 | 71.9% | 72.0% |
| 10 | 5 | 3 | 267 | 18032 | 82 | 70.1% | 70.2% |
| 11 | 3 | 1 | 486 | 19004 | 85 | 68.5% | 68.6% |
| 12 | 4 | 4 | 257 | 19775 | 89 | 67.2% | 67.3% |
| 13 | 4 | 2 | 245 | 20510 | 93 | 66.0% | 66.1% |
| 14 | 5 | 1 | 156 | 21134 | 98 | 64.9% | 65.1% |
| 15 | 12 | 5 | 50 | 21684 | 110 | 64.0% | 64.2% |
| 16 | 9 | 5 | 67 | 22220 | 119 | 63.1% | 63.3% |
| 17 | 5 | 1 | 129 | 22736 | 124 | 62.3% | 62.5% |
| 18 | 5 | 1 | 129 | 23252 | 129 | 61.4% | 61.6% |
| 19 | 3 | 1 | 238 | 23728 | 132 | 60.6% | 60.9% |
| 20 | 6 | 2 | 85 | 24153 | 138 | 59.9% | 60.2% |
| 21 | 16 | 4 | 28 | 24573 | 154 | 59.2% | 59.5% |
| 22 | 13 | 6 | 35 | 24993 | 167 | 58.5% | 58.8% |
| 23 | 3 | 1 | 183 | 25359 | 170 | 57.9% | 58.2% |
| 24 | 6 | 5 | 73 | 25724 | 176 | 57.3% | 57.6% |
| 25 | 8 | 2 | 49 | 26067 | 184 | 56.8% | 57.1% |
| 26 | 5 | 1 | 84 | 26403 | 189 | 56.2% | 56.5% |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 27 | 5 | 1 | 84 | 26739 | 194 | 55.6% | 56.0% |
| 28 | 9 | 5 | 42 | 27075 | 203 | 55.1% | 55.4% |
| 29 | 9 | 5 | 41 | 27403 | 212 | 54.5% | 54.9% |
| 30 | 12 | 4 | 24 | 27667 | 224 | 54.1% | 54.5% |
| 31 | 19 | 4 | 14 | 27919 | 243 | 53.7% | 54.1% |
| 32 | 19 | 8 | 14 | 28171 | 262 | 53.3% | 53.7% |
| 33 | 5 | 1 | 62 | 28419 | 267 | 52.8% | 53.3% |
| 34 | 8 | 2 | 35 | 28664 | 275 | 52.4% | 52.9% |
| 35 | 5 | 1 | 60 | 28904 | 280 | 52.0% | 52.5% |
| 36 | 3 | 1 | 113 | 29130 | 283 | 51.7% | 52.1% |
| 37 | 16 | 6 | 14 | 29340 | 299 | 51.3% | 51.8% |
| 38 | 7 | 1 | 33 | 29538 | 306 | 51.0% | 51.5% |
| 39 | 8 | 2 | 28 | 29734 | 314 | 50.7% | 51.2% |
| 40 | 8 | 2 | 28 | 29930 | 322 | 50.3% | 50.9% |
| 41 | 7 | 3 | 30 | 30110 | 329 | 50.0% | 50.6% |
| 42 | 8 | 2 | 25 | 30285 | 337 | 49.8% | 50.3% |
| 43 | 7 | 2 | 28 | 30453 | 344 | 49.5% | 50.0% |
| 44 | 7 | 3 | 28 | 30621 | 351 | 49.2% | 49.8% |
| 45 | 4 | 1 | 54 | 30783 | 355 | 48.9% | 49.5% |
| 46 | 12 | 4 | 14 | 30937 | 367 | 48.7% | 49.3% |
| 47 | 10 | 2 | 16 | 31081 | 377 | 48.4% | 49.1% |
| 48 | 7 | 2 | 24 | 31225 | 384 | 48.2% | 48.8% |
| 49 | 19 | 6 | 8 | 31369 | 403 | 48.0% | 48.6% |
| 50 | 12 | 7 | 13 | 31512 | 415 | 47.7% | 48.4% |
| 51 | 11 | 3 | 14 | 31652 | 426 | 47.5% | 48.2% |
| 52 | 11 | 3 | 14 | 31792 | 437 | 47.3% | 48.0% |
| 53 | 12 | 6 | 12 | 31924 | 449 | 47.0% | 47.8% |
| 54 | 10 | 2 | 14 | 32050 | 459 | 46.8% | 47.6% |
| 55 | 22 | 7 | 6 | 32176 | 481 | 46.6% | 47.4% |
| 56 | 5 | 1 | 30 | 32296 | 486 | 46.4% | 47.2% |
| 57 | 10 | 2 | 13 | 32413 | 496 | 46.2% | 47.0% |
| 58 | 9 | 3 | 14 | 32525 | 505 | 46.0% | 46.9% |
| 59 | 7 | 2 | 18 | 32633 | 512 | 45.9% | 46.7% |
| 60 | 3 | 1 | 50 | 32733 | 515 | 45.7% | 46.5% |
| 61 | 4 | 2 | 33 | 32832 | 519 | 45.5% | 46.4% |
| 62 | 6 | 1 | 18 | 32922 | 525 | 45.4% | 46.2% |
| 63 | 7 | 3 | 15 | 33012 | 532 | 45.2% | 46.1% |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 64 | 19 | 6 | 5 | 33102 | 551 | 45.1% | 46.0% |
| 65 | 3 | 1 | 44 | 33190 | 554 | 44.9% | 45.9% |
| 66 | 3 | 1 | 44 | 33278 | 557 | 44.8% | 45.7% |
| 67 | 12 | 7 | 8 | 33366 | 569 | 44.6% | 45.6% |
| 68 | 4 | 1 | 29 | 33453 | 573 | 44.5% | 45.4% |
| 69 | 7 | 2 | 14 | 33537 | 580 | 44.4% | 45.3% |
| 70 | 7 | 3 | 14 | 33621 | 587 | 44.2% | 45.2% |
| 71 | 15 | 7 | 6 | 33705 | 602 | 44.1% | 45.1% |
| 72 | 3 | 1 | 41 | 33787 | 605 | 43.9% | 44.9% |
| 73 | 5 | 2 | 20 | 33867 | 610 | 43.8% | 44.8% |
| 74 | 17 | 7 | 5 | 33947 | 627 | 43.7% | 44.7% |
| 75 | 12 | 3 | 7 | 34024 | 639 | 43.6% | 44.6% |
| 76 | 12 | 3 | 7 | 34101 | 651 | 43.4% | 44.5% |
| 77 | 5 | 1 | 19 | 34177 | 656 | 43.3% | 44.4% |
| 78 | 5 | 1 | 19 | 34253 | 661 | 43.2% | 44.3% |
| 79 | 16 | 8 | 5 | 34328 | 677 | 43.0% | 44.2% |
| 80 | 5 | 1 | 18 | 34400 | 682 | 42.9% | 44.1% |
| 81 | 5 | 1 | 18 | 34472 | 687 | 42.8% | 43.9% |
| 82 | 5 | 1 | 18 | 34544 | 692 | 42.7% | 43.8% |
| 83 | 13 | 5 | 6 | 34616 | 705 | 42.6% | 43.7% |
| 84 | 4 | 1 | 23 | 34685 | 709 | 42.5% | 43.6% |
| 85 | 5 | 1 | 17 | 34753 | 714 | 42.3% | 43.5% |
| 86 | 5 | 1 | 17 | 34821 | 719 | 42.2% | 43.4% |
| 87 | 5 | 1 | 17 | 34889 | 724 | 42.1% | 43.3% |
| 88 | 5 | 1 | 17 | 34957 | 729 | 42.0% | 43.2% |
| 89 | 12 | 3 | 6 | 35023 | 741 | 41.9% | 43.1% |
| 90 | 7 | 3 | 11 | 35089 | 748 | 41.8% | 43.0% |
| 91 | 5 | 1 | 16 | 35153 | 753 | 41.7% | 42.9% |
| 92 | 9 | 3 | 8 | 35217 | 762 | 41.6% | 42.8% |
| 93 | 3 | 1 | 31 | 35279 | 765 | 41.5% | 42.7% |
| 94 | 3 | 1 | 30 | 35339 | 768 | 41.4% | 42.6% |
| 95 | 11 | 2 | 6 | 35399 | 779 | 41.3% | 42.6% |
| 96 | 6 | 2 | 12 | 35459 | 785 | 41.2% | 42.5% |
| 97 | 21 | 7 | 3 | 35519 | 806 | 41.1% | 42.4% |
| 98 | 5 | 1 | 14 | 35575 | 811 | 41.0% | 42.3% |
| 99 | 5 | 1 | 14 | 35631 | 816 | 40.9% | 42.2% |
| 100 | 5 | 1 | 14 | 35687 | 821 | 40.8% | 42.2% |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 101 | 5 | 1 | 14 | 35743 | 826 | 40.7% | 42.1% |
| 102 | 5 | 1 | 14 | 35799 | 831 | 40.6% | 42.0% |
| 103 | 5 | 1 | 14 | 35855 | 836 | 40.5% | 41.9% |
| 104 | 5 | 1 | 14 | 35911 | 841 | 40.4% | 41.8% |
| 105 | 5 | 1 | 14 | 35967 | 846 | 40.3% | 41.7% |
| 106 | 9 | 3 | 7 | 36023 | 855 | 40.2% | 41.7% |
| 107 | 12 | 3 | 5 | 36078 | 867 | 40.1% | 41.6% |
| 108 | 12 | 3 | 5 | 36133 | 879 | 40.1% | 41.5% |
| 109 | 12 | 3 | 5 | 36188 | 891 | 40.0% | 41.4% |
| 110 | 12 | 3 | 5 | 36243 | 903 | 39.9% | 41.4% |
| 111 | 12 | 3 | 5 | 36298 | 915 | 39.8% | 41.3% |
| 112 | 12 | 5 | 5 | 36353 | 927 | 39.7% | 41.2% |
| 113 | 3 | 1 | 27 | 36407 | 930 | 39.6% | 41.1% |
| 114 | 3 | 1 | 27 | 36461 | 933 | 39.5% | 41.1% |
| 115 | 3 | 1 | 26 | 36513 | 936 | 39.4% | 41.0% |
| 116 | 5 | 1 | 13 | 36565 | 941 | 39.3% | 40.9% |
| 117 | 5 | 1 | 13 | 36617 | 946 | 39.2% | 40.8% |
| 118 | 6 | 2 | 10 | 36667 | 952 | 39.2% | 40.7% |
| 119 | 8 | 4 | 7 | 36716 | 960 | 39.1% | 40.7% |
| 120 | 5 | 1 | 12 | 36764 | 965 | 39.0% | 40.6% |
| 121 | 7 | 1 | 8 | 36812 | 972 | 38.9% | 40.5% |
| 122 | 13 | 5 | 4 | 36860 | 985 | 38.8% | 40.5% |
| 123 | 3 | 1 | 23 | 36906 | 988 | 38.8% | 40.4% |
| 124 | 6 | 2 | 9 | 36951 | 994 | 38.7% | 40.3% |
| 125 | 16 | 6 | 3 | 36996 | 1010 | 38.6% | 40.3% |
| 126 | 5 | 2 | 11 | 37040 | 1015 | 38.5% | 40.2% |
| 127 | 12 | 3 | 4 | 37084 | 1027 | 38.5% | 40.2% |
| 128 | 12 | 4 | 4 | 37128 | 1039 | 38.4% | 40.1% |