

Fast compression of transportable Tcl agents

Robert S. Gray
Department of Computer Science
Dartmouth College
Hanover, NH 03755

E-mail: robert.s.gray@dartmouth.edu

Abstract

An *information agent* is charged with the task of searching a collection of electronic resources for information that is relevant to the user's current needs. These resources are often distributed across a network and can contain tremendous quantities of data. One of the paradigms that has been suggested for allowing efficient access to such resources is *transportable agents* – the agent is sent to the machine that maintains the information resource; the agent executes on this remote machine and then returns its results to the local machine. We have implemented a transportable agent system that uses the Tool Command Language (Tcl) as the agent language. Each Tcl script can suspend its execution at an arbitrary point, transport itself to another machine and resume execution on the new machine. The execution state of the script – which includes the commands that have not been executed – must be transmitted to the new machine. Although the execution state tends to be small, there will be a large number of agents moving across the network in a large-scale system. Thus it is desirable to compress the execution state as much as possible. Furthermore any compression scheme must be fast so that it does not become a bottleneck between the transportable agent system and the network routines. In this paper we explore several fast compression methods.

1 Introduction

An *information agent* is charged with the task of searching a collection of electronic resources for information that is relevant to the user's current needs. These resources are often distributed across a network and can contain tremendous quantities of data. One of the paradigms that has been suggested for allowing efficient access to such resources is *transportable agents* [KK94] – the agent is sent to the machine that maintains the information resource; the agent executes on this remote machine and then returns its results to the local machine. We have implemented a transportable agent system. The agents are written in the *Tool Command Language (Tcl)* – a high-level command language that was created by Dr. John Ousterhout at the University of California at Berkeley [Ous94]. Tcl is an attractive language for transportable agents. It is popular, easy to use, widely available and platform independent. It can be embedded in other applications due to its dual existence as a stand-alone interpreter and a programming library. Finally Tcl is extendible which means that each resource can provide a set of Tcl commands that an agent can use to access the resource.

In the prototype system a Tcl script can suspend its execution at an arbitrary point, transmit itself to another machine and resume execution on the new machine. The execution state of the script – which includes the commands that have not been executed – must be transmitted to the new machine. Although the execution state tends to be small, there will be a large number of agents moving across the network in a large-scale system. Each execution state should be compressed as much as possible in order to reduce network contention. In addition the compression scheme must be fast since the execution state is compressed on its way to the network; a compression scheme that is significantly slower than the kernel routines that inject packets into the network will become a bottleneck. In this paper we implement several fast compression methods and examine their performance on a small corpus of Tcl scripts. The next section explains the pieces that make up the execution state of a Tcl script. The following sections describe the compression methods and their performance.

2 Tcl state

The execution state of a Tcl script consists of four pieces – a global variable table, the names and bodies of defined procedures, a call frame for each active procedure and a command stack. The variable table contains one entry for each defined variable. Each entry is a five-tuple. The first element is a flag that indicates whether the variable is *defined* or *undefined* – a variable is undefined if the variable has not been assigned a value but has been referenced in some way (such as specifying a “trace” procedure that will be called whenever the variable is accessed). The second element is the name of the variable. The third element is a reference count that indicates how many variables are aliases for the given variable. The fourth element is a flag that indicates whether the variable is an *alias*, a *string* or an *array*. The fifth element is either an integer that identifies the variable to which the alias refers, a string value or a variable table that contains the array elements and their values. The alias ids should be coded with a fixed-length binary code (the upper bound is the total number of variables). The reference counts should be coded with a fixed-length binary code or with a Huffman or arithmetic coder since small reference counts are much more likely than large counts. The flags exhibit no correlation from one entry to the next so they should be coded in binary or with an arithmetic coder if we are willing to sacrifice speed. An arithmetic coder takes advantage of the fact that some flags are much more likely than others – i.e. scalars are much more common than arrays and aliases; defined variables are much more common than undefined variables. In any case we would take advantage of the fact that an array of arrays is not allowed so the *array* flag can not occur in a table of array elements.

Each call frame contains an integer that specifies the dynamic scoping level and a variable table for local variables. The dynamic scoping level should be coded with a phased-in binary code since the dynamic scoping level can not be greater than the number of enclosing scopes. The variable table should be coded in the same manner as the global variable table.

The command stack contains the sequence of *nested* scripts that are currently being evaluated. Each element on the stack contains a Tcl script and a state flag that indicates the current execution status of that script. For example, immediately after a procedure call at the top level, there would be two elements on the stack. The first element would contain all of the remaining commands at the top level (i.e. a closure) and a state flag indicating that execution should proceed to the next command as soon as the procedure call has finished. The second element would contain the body of the procedure and a state flag indicating that execution should proceed to the first command in the body. The state flags tend to be small - i.e. almost always in the range 0 to 2 – but have no fixed upper bound. They should be coded with one of the variable-length integer codes such as γ or δ [BCW90].

The remaining parts of the state are the names and values of variables, the names of procedures, the Tcl scripts that make up the procedure bodies and the Tcl scripts on the command stack. Due to time constraints we will focus on compressing the Tcl scripts. In addition we will focus on compressing the original Tcl script rather than the fragments of the original script that appear in the execution state. This is not unreasonable since the execution state essentially contains two copies of the original script – one copy contains the commands from the original script while the second copy contains these same commands after variable, command and backslash substitutions have been performed. It should be noted that we are ignoring part of the problem by taking this simplified approach – i.e. how to take advantage of the fact that the two copies are nearly identical. This probably involves some manner of “diff” operation and will have to be considered in future work.

3 Compression of Tcl scripts

Several compression methods were considered. Since one of the goals is to compress the Tcl scripts fast enough to keep up with the network routines in the system kernel, only nonadaptive compression methods were implemented.

Method	Comment	parray	dialog	init	menu	Total
B0	No compression	850 (189%)	3,327 (209%)	7,615 (159%)	22,300 (248%)	34,092 (216%)
B1	Strip	449 (100%)	1,585 (100%)	4,787 (100%)	8,961 (100%)	15,782 (100%)
B2	Hash table	308 (68%)	1,038 (65%)	2,706 (57%)	4,564 (51%)	8,616 (55%)
B3	Hash table with Huffman code	237 (53%)	818 (52%)	2,053 (43%)	3,341 (37%)	6,449 (41%)
B4	gzip	256 (57%)	716 (45%)	1,650 (34%)	2,396 (27%)	5,018 (32%)

Table 1: Baseline compression methods (sizes expressed in *bytes*)

Method	Context Size	parray	init	menu
H0	1 character	294 (65%)	2,990 (62%)	5,276 (59%)
H1	2 characters	308 (68%)	2,706 (57%)	4,564 (51%)
H2	3 characters	332 (74%)	2,936 (61%)	4,853 (54%)

Table 2: Effect of context size on hash table compression (sizes expressed in *bytes*)

3.1 Baseline methods

Five baseline methods were implemented. The compression performance of these methods on a test corpus of Tcl scripts is shown in Table 1. Four Tcl scripts were used as the test corpus. *parray* lists the elements of an array, *init* handles undefined commands, *dialog* handles dialog boxes and *menu* handles pulldown menus. The scripts range from 850 characters to 22,300 characters which is relatively large for a Tcl script. The test scripts are not transportable agents since only the most trivial transportable agents were available. It is hoped that transportable agents will be roughly similar to the test scripts in terms of command and symbol usage.

Method B0 performs no compression on the Tcl scripts. B1 removes all comments and extraneous whitespace from the Tcl scripts. B1 is a valid compression method since we are compressing the scripts for transmission to a remote machine where they will be evaluated and forgotten. The remote machine does not need comments or whitespace since it is not interested in recreating the original script. B1 cuts the size of the original scripts in half (it should be noted that the test scripts are heavily commented; the size of the test scripts drops by just ten percent if B1 removes only the extraneous whitespace). In the tables and in the remainder of the paper, compression performance is measured relative to the performance of B1 since B1 eliminates the information that does not need to be transmitted at all.

B2 is based on the hash table scheme in [BJLM92] and [Wil91]. The compression algorithm uses a 4096-element hash table. Each hash bucket contains a 4-character *move-to-front* (MTF) list. Every character in the hash table is initialized to the *null* character. The algorithm performs a linear scan through the script (ignoring comments and extraneous whitespace). For each character the algorithm calculates a hash index based on the 2 preceding characters and looks for the character in the corresponding MTF list. If the character appears in the MTF list, the algorithm outputs a 0 followed by a 2-bit integer that gives the position of the character in the MTF list. Then the algorithm moves the character to the front of the MTF list. If the character does not appear in the list, the algorithm outputs a 1 followed by a 7-bit character code. Then the algorithm puts the character at the front of the MTF list (dropping the least recently used character off the end). B2 approaches 50% compression as the size of the scripts increases.

The number of buckets in the hash table and the number of elements in each MTF list do not significantly affect compression performance (the number of hits at each position in the MTF list drops exponentially as we move towards the end of the list) unless we make the number of hash buckets absurdly small. The context size is more critical. Table 2 shows compression performance as a function of how many characters are used to calculate the hash index. Using only 2 preceding characters is best so all of the compression all of the compression methods use only 2 preceding characters. The hash function for the 2 character case is

$$index = (c1 \ll 4) \wedge c0$$

$index = index \& 4095$

where $c0$ is the preceding character, $c1$ is the character before $c0$, \ll is *shift left*, \wedge is *xor* and $\&$ is *and*. This is similar to the hash function used in [Wil91] except that we have removed a multiplication (at no cost in compression performance). For the 3 character case we add another *shift* and *xor* operation to get

$index = (((c2 \ll 4) \ll c1) \ll 4) \wedge c0$

$index = index \& 4095$

These hash functions have the advantage of being exceptionally fast.

Method B3 combines the hash table of B2 with a static Huffman code. There are 132 codes. There is one code for each position in the MTF list and one code for each possible miss (any one of 128 different characters might not appear in the MTF list). The Huffman code used is the one that minimizes the *total* size of the four compressed scripts – i.e. we obtained the frequency distribution of hits and misses for each script, summed the four distributions and calculated the optimal Huffman code for the summed distribution. B3 approaches 40% compression as the size of the scripts increases.

Method B4 applies “gzip” to each script (after comments and extraneous whitespace have been removed). B4 approaches 30% compression as the size of the scripts increases.

3.2 Syntactic compression

The best compression for *programs* is achieved through syntax-directed compression in which the program is parsed and then the *parse tree* is encoded. This approach has two disadvantages in our situation – (1) the time needed to construct the parse tree is substantial and (2) Tcl can not be parsed easily with only one symbol of lookahead unless its syntax is changed. However it is likely that the syntax of Tcl will be changed anyways for efficiency – i.e. the speed of the Tcl interpreter can be increased substantially by adding a preprocessing phase that parses the entire script and then passes the parse tree to the evaluation routines. Furthermore such a preprocessing phase means that the parse tree would already be available when it came time to compress and transmit the script. A walk through the parse tree would be no more expensive than a linear scan through the original script. Thus it seems reasonable to explore syntactic-directed compression methods for Tcl scripts. We use the modified syntax that is proposed in [SBD94] and used in the Tcl-like Rush language. The main syntactic difference between Tcl and Rush is that Rush enforces consistent use of delimiters – i.e. strings must be delimited by double quotes, subcommands must be delimited by curly brackets and so on. The Rush research group has a program that semi-automatically converts Tcl scripts to Rush scripts. This program is not publicly available so we converted the corpus of test scripts to Rush syntax by hand. This manual conversion was the limiting factor on the size of the test corpus.

[Cam88] and [KPT86] are the primary references for syntax-directed compression. Both of these researchers compress Pascal programs. [KPT86] constructs a parse tree for the Pascal program, removes all nodes that have only one child and then linearizes the parse tree. All user-defined symbols are added to a symbol table as the parse tree is constructed. Associated with each node in the tree is an integer that identifies the grammar production that was used to expand the node. The output of the compression program is the symbol table followed by an encoding of the linearized parse tree – i.e. the symbol and production indices in the parse tree are coded with a fixed-length binary code. [KPT86] achieves 50% compression.

[Cam88] constructs a parse tree for the Pascal program, performs a preorder walk through the tree and uses arithmetic coding to encode the tree during the course of the walk. For each non-terminal the compression algorithm encodes the *probability* of the production that was used to expand the non-terminal. This approach has two advantages – only a few productions apply to each non-terminal so the probability of most productions are zero for a given non-terminal *and* certain productions associated with a non-terminal are far more likely than others. The production probabilities are static and are based on evaluation of a corpus of Pascal programs. All user-defined symbols are coded on a character-by-character basis with an adaptive probability model. The order of the model was not specified. [Cam88] achieves 25% compression using this base technique and 15% compression with multiple symbol tables and grammar reorganization.

Method	Comment	parray	dialog	init	menu	Total
S0	Binary	173 (39%)	881 (56%)	1,814 (38%)	3,149 (35%)	6,017 (38%)
S1	Phased binary	163 (36%)	821 (52%)	1,709 (36%)	2,851 (32%)	5,544 (35%)
S2	Integers	159 (35%)	812 (51%)	1,687 (35%)	2,826 (32%)	5,484 (35%)
S3	Huffman on hash table	156 (35%)	793 (50%)	1,629 (34%)	2,775 (31%)	5,353 (34%)
S4	MTF for symbol table	157 (35%)	795 (50%)	1,607 (34%)	2,645 (30%)	5,204 (33%)
S5	Better Huffman on hash table	139 (31%)	696 (44%)	1,445 (30%)	2,493 (28%)	4,773 (30%)
S6	Arithmetic	143 (32%)	660 (42%)	1,346 (28%)	2,188 (24%)	4,337 (27%)
S7	No symbol table	158 (35%)	801 (51%)	1,626 (34%)	3,160 (35%)	5,745 (36%)

Table 3: Syntactic compression methods (sizes expressed in *bytes*)

3.3 Syntactic compression methods for Tcl

Eight syntax-directed compression methods were implemented – these methods focus on reducing the size of the compressed symbols since even the naive method S0 compresses the parse tree well. These methods are based on the techniques of [Cam88] except that we do not use arithmetic or adaptive coding due to the need for fast compression. An LR(1) grammar was written for Tcl scripts (or more precisely Rush scripts since we are using a modified syntax). All productions that can expand a *given non-terminal* are assigned unique integral indices starting with 0. Each of the eight encoders parses the Tcl script and constructs a parse tree. Then the encoder performs a preorder walk through the tree. For each non-terminal the encoder encodes the index of the production that was used to expand the non-terminal. The decoder starts with the top level non-terminal (e.g. *command list*). This first production index in the compressed script specifies which production should be used to expand this non-terminal and so on. As long as the decoder walks the tree in the same order as the encoder, it will be able to decode the compressed parse tree. Thus there is no need to have a unique integer id for *every* production in the grammar. [Cam88] uses the same technique except that the productions are assigned probabilities instead of indices. The eight methods differ in their handling of terminal symbols and the coding methods used for the production indices.

S0 codes the production indices using a fixed-length binary code (the upper bound is the number of productions associated with the current non-terminal). Strings are coded with the hash table scheme of B2. Symbols – i.e. integers, variable names and procedure names – are added to a global symbol table and coded with the hash table scheme of B2 on their first occurrence. On successive occurrences the symbol table index is coded using a fixed-length binary code (the upper bound is the number of symbols in the table). A single-bit flag indicates whether the next symbol has been seen before. S0 achieves 38% compression.

S1 takes advantage of the fact that the upper bound on the production and symbol indices is rarely an exact power of two. It encodes the indices using the efficient phased-in scheme of [BCW90] – i.e. if a number i is known to be in the range $0 \leq i < p$, then i is coded in $(k - 1)$ bits if it is less than $2^{\lceil \log p \rceil} - p$; otherwise i is coded in k bits. S1 achieves 35% compression.

S0 and S1 treat integers as symbols that are no different than procedure and variable names. S2 treats integers as integers. A signal-bit flag indicates whether the integer is positive or negative; the magnitude of the integer is coded using the γ code of [BCW90]. The savings are small for the four test scripts but should be significant for scripts that contain a larger number of integers. Floating point numbers are not treated separately since the test scripts do not contain any floating point numbers. This must be considered in future work.

S3 adds a static Huffman code to the hash table scheme that is used to encode strings and new symbols. There are five codes. One code indicates that the current character was not found in the MTF list; the other four codes indicate the position in the MTF list in which the character appeared. For each character the appropriate code is output with the code for an *MTF miss* followed by the 7-bit character code. As before

Method	Component	parray	dialog	init	menu
B0	Parse Tree	251 (100%)	464 (100%)	2,346 (100%)	4,152 (100%)
B4	Parse Tree	105 (42%)	120 (26%)	625 (27%)	855 (21%)
S5	Parse Tree	55 (22%)	177 (46%)	514 (22%)	1,139 (27%)
S6	Parse Tree	48 (19%)	113 (24%)	387 (17%)	794 (19%)
B0	Symbols	198 (100%)	1,121 (100%)	2,441 (100%)	4,809 (100%)
B4	Symbols	151 (76%)	596 (53%)	1,025 (42%)	1,541 (32%)
S5	Symbols	84 (42%)	519 (46%)	931 (38%)	1,354 (28%)
S6	Symbols	95 (48%)	547 (49%)	959 (39%)	1,394 (29%)

Table 4: Compression of syntactic versus symbolic information (sizes expressed in *bytes*)

we obtained the frequency distribution of hits and misses for each script, summed the four distributions and calculated the optimal Huffman code for the summed distribution. S3 achieves 34% compression.

S4 adds a move-to-front list to the symbol table. The eight most recent symbols are kept in an MTF list. There is a three-element static Huffman code. One code indicates that the symbol was found in the MTF list; the second code indicates that the symbol was found in the symbol table but not in the MTF list; and the third code indicates that the symbol is a new symbol. The code for a hit in the MTF list is followed by a 3-bit index. The code for a hit in the symbol table is followed by the symbol table index as before. The code for a miss is followed by the character-by-character encoding as before. The optimal Huffman code was calculated as before. S4 achieves 33% compression but the savings are insignificant except on the largest script. A ten-element Huffman code with a code for each position in the MTF list made compression worse since the distribution of hits in the MTF list varied too widely from script to script.

S5 abandons the simple Huffman code of S3 and uses the full Huffman code of method B3 – i.e. there are 132 codes, four codes for hits in the MTF list and 128 codes for misses. S5 achieves 30% compression and seems to be the best that can be done with non-adaptive and non-arithmetic techniques. It is competitive with gzip and with the base technique of [Cam88]. [Cam88] cites 25% compression for the base technique but is measuring compression against programs that contain whitespace. If we count whitespace, the compression percentages for S5 are around 26% for every script except *dialog*.

The most time-consuming aspect of S5 – and the aspect that perhaps will be difficult to implement in hardware – is the memory management and string comparisons associated with the symbol table. Method S7 eliminates the symbol table and encodes *all* symbols on a character-by-character basis. S7 achieves 36% compression and is competitive with gzip except on the largest script.

The main problem with S5 is that binary and Huffman coding is inefficient for small alphabets (especially the production indices since there are only a handful of non-terminals associated with each non-terminal). S6 uses adaptive arithmetic coding for the sake of comparison. S6 uses five adaptive order-0 models. The first model gives the probability that a given production will be used to expand a given non-terminal. The second model gives the probability of hits and misses in the MTF lists in the hash table. The last three models give the probability of positive and negative integers, the probability of a 0 or 1 bit in the integer magnitudes and the probability of hits and misses in the MTF symbol list. All models are initialized so that all symbols are equally probable. S6 is slower but achieves 27% compression. S6 can be improved significantly with grammar reorganization, more complex models, more symbol tables and better initial values for the probabilities as was done in [Cam88].

All of the syntax-directed methods outcompress B3 which uses just the hash table and a static Huffman code. However only S5 and S6 outcompress B4 which is the standard “gzip” program. In addition S5 significantly outcompresses B4 only on *parray* and *init* and actually does worse on the largest script *menu*. Table 4 compares the compression performance of B4, S5 and S6. The table is divided into two sections – bytes needed for the compressed parse tree and bytes needed for the compressed symbols (for “gzip” the symbols and strings were extracted from the scripts and compressed separately to get the *symbol* bytes; the difference between the symbol bytes and the total bytes was taken to be the *parse tree* bytes; this is unscientific but seemed to be the only way to come close to the desired data without modifying the “gzip” source code to

collect statistics). S5 compresses the symbols better than “gzip” which is heartening since S5 uses just a hash table and a global symbol table. However S5 performs worse on the parse trees for *dialog* and *menu*. Part of the problem seems to be that *dialog* and *menu* use the same commands over and over. “gzip” – which just looks for previous occurrences of a string – ends up compressing entire commands into a single pointer while S5 wastes bits by separating the script into syntax and symbols and compressing the syntax separately. In addition *dialog* and *menu* have a higher proportion of user-defined commands which have a high overhead in terms of parse tree nodes. This leads to a large bit overhead since we are using integral numbers of bits to encode the production indices. The problem disappears when we move to arithmetic coding in S6 which compresses *all* the parse trees significantly better than “gzip” (S6 compresses the symbols worse than S5 but this is because all characters start with equal probabilities; for example an initial probability distribution that reflected the Huffman code lengths in S5 would lead to performance no worse than S5).

init and *parray* have more varied command usage and use a higher-proportion of built-in commands. S5 shows significantly better performance on these scripts and outcompresses “gzip”. In addition S5 and S6 give more consistent compression performance than “gzip” – i.e. performance does not vary widely as the size of the script changes. *dialog* is an exception and shows notably worse compression performance than the other three scripts. This is because of a particularly bad parse tree (in terms of the number of nodes per command) and because *dialog* contains more user-defined symbols relative to its length. As shown in table 4 the symbols do not compress as well as the parse trees. In addition *dialog* and *parray* contain far fewer repeated symbols which reflects itself in the poor compression performance on the symbols; *init* has more repeats and better compression performance on symbols; while *menu* has the most repeats and the best compression on symbols.

It is interesting to conjecture whether the typical transportable agent will be closer to *init* or to *dialog*. Transportable agents tend to have large amounts of control wrapped around calls to library routines that are available on the remote machine. This should mean that the agents will exhibit command and symbol usage similar to *init* rather than *dialog* and will compress well.

3.4 Speed

It is difficult to make meaningful speed comparisons since none of the code has been optimized. Currently *all* of the compression methods run twice as slowly as gzip (except for B2 which just strips the comments and whitespace out of a script and runs as fast as gzip). There are four contributing factors – the file I/O and memory allocation is being done in the easiest rather than the most efficient manner (i.e. read a byte at a time and do not preallocate memory); the code is written in C++ rather than C which is a substantial performance hit in and of itself; “flex” does not necessarily produce the fastest lexer for Tcl scripts; and most of the encoding routines can be made tighter or even reimplemented in machine code. It has been shown that the hash table schemes such as B2 and B3 can run much faster than gzip with appropriate implementation [BJLM92, Wil91]. The syntax-directed encoders – with the exception of S6 which uses adaptive arithmetic coding – should run much faster than “gzip” if the parse tree is already available and should at least be competitive if the parse tree must be constructed from scratch. This is especially true for S5 which does not use a symbol table.

4 Conclusions

If the original script is being compressed directly, the hash table scheme B3 or the gzip program B4 would give the fastest compression. S5 would give better compression at the cost of constructing a parse tree for the script. Improved versions of S6 would give the best compression at a substantial speed penalty. If the parse tree is already available, S5 and S7 give the fastest compression with S7 having the edge over S5 since it does not use a symbol table. Improved versions of S6 will again give the best compression. B3 and S7 – the fastest of these algorithms – achieve 35 to 40 percent compression. B4 achieves 32 percent with S5 achieving 30 percent – the performance of these two methods is so close that gzip should be chosen over S5 when compressing the original script directly. Improved versions of S6 should achieve well below 25 percent

compression but would be much slower. In a real application we would need to choose the method that gave the best balance between low network traffic and low network latency.

There are two issues that need to be addressed in future work. The first issue is to optimize the source code for the better methods. In particular we need optimized versions of B3, S5 and S7 so that reasonable execution time comparisons can be made. The second issue is to compress the rest of the execution state. There are clear choices for encoding much of the state as discussed above (such as the flags in the variable table). However it is less clear how to compress the variable names and values and how to compress the script fragments. A reasonable first pass for the variables is to use the same hash table scheme that was used in the script compression. For the script fragments it is necessary to develop the equivalent of a “diff” operation so that two fragments that are almost identical will be compressed efficiently. This is not difficult if the Tcl interpreter is maintaining a parse tree for efficiency reasons since the interpreter can set appropriate pointers as it performs variable, command and backslash substitutions. It is more difficult if only the script fragments are available.

5 Acknowledgements

Many thanks to Professor John Danskin for useful discussions; to my advisor, Professor George Cybenko, for his encouragement and support; and, as always, to Jennifer and Stephen Gray for reminding me that there is life outside graduate school.

References

- [BCW90] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*, chapter A (Variable-length Representations of the Integers), pages 290–295. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.
- [BJLM92] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*. ACM, October 1992.
- [Cam88] Robert D. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, July 1988.
- [KK94] Keith Kotay and David Kotz. Transportable agents. In *CIKM Workshop on Intelligent Information Agents (held in conjunction with the Third International Conference on Information and Knowledge Management)*, Gaithersburg, Maryland, December 1994. National Institute of Standards and Technology.
- [KPT86] Jyrki Katajainen, Martti Penttonen, and Jukka Teuhola. Syntax-directed compression of program files. *Software – Practice and Experience*, 16(3):269–276, March 1986.
- [Ous94] John K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley, Reading, Massachusetts, 1994.
- [SBD94] Adam Sah, Jon Blow, and Brian Dennis. An introduction to the Rush language. In *Proceedings of the 1994 Tcl Workshop*, June 1994.
- [Wil91] Ross N. Williams. An extremely fast Liv-Zempel data compression algorithm. In *IEEE Data Compression Conference*, Snowbird, Utah, April 1991. IEEE.