Compacting Object Code via Parameterized Procedural Abstraction

by

Michael Joseph Zastre
B.Sc., Simon Fraser University, 1993

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. R. N. Horspool, Supervisor (Dept. of Computer Science)

[redacted]

Dr. D. M. Hoffman (Dept. of Computer Science)

[redacted]

Dr. G. C. Shoja (Dept. of Computer Science)

[redacted]

Dr. R. Vahldieck (Dept. of Electrical and Computer Engineering)

Supervisor: Dr. R. N. Horspool

Abstract

Current *compiler optimization* research focuses on program transformations that increase code speed. Decreasing object file size is sometimes a goal since less code often means faster code. In this thesis we examine *parameterized procedural abstraction*. This is an extension of an optimization whose sole purpose is to reduce code size. Previously published implementations of *procedural abstraction* have produced space savings where instruction sequences are exact matches (i.e. instruction operations *and* operands are the same between code sequences). Researchers felt that parameterizing differences had too high a cost, but no investigations have been made into the potential space savings from exploiting inexact matches. We show that permanent space savings (*compaction*) are possible when (1) covering matches with more than one procedure and (2) carefully choosing which inexact match instances are covered by each procedure. Our heuristic algorithms choose a mix of parameterized and unparameterized procedures that yield substantially better space savings in comparison to unparameterized procedures only.

Examiners:

[redacted]

Dr. R. N. Horspool, Supervisor (Dept. of Computer Science)

[redacted]

Dr. D. M. Hoffman, Departmental Member (Dept. of Computer Science)

[redacted]

Dr. G. C. Shoja, Departmental Member (Dept. of Computer Science)

[redacted]

Dr. R. Vahldieck, External Examiner (Dept. of Electrical and Computer Engineering)

Table of Contents

Abstract ii
Table of Contents iii
List of Tables iv
List of Figures v
Acknowledgments vi
Dedication vii

1: Introduction 1

2: Previous Work 3
2.1: Compiler Optimizations Leading to Space Savings 3
2.2: Exploiting Similarity of Code 10
2.3: Matching Code Representation to Code Properties 11
2.4: Compilation to Compact Code 14
2.5: Analyzing and Compressing Assembly Code 21
2.6: Looking Forward 26

3: Compaction and its Problems 28
3.1: The Problem 29
3.2: Preliminaries to Procedure Searching 31
3.3: Procedure Searching 35
3.4: Quick Comparison of Different Methods 37
3.5: Summary 39

4: Searching for Subroutines 41
4.1: Definitions 41
4.2: Visualizing Operands 43
4.3: Ω-array Partitions 45
4.4: Heuristics 51
4.5: Further transformations that could be pursued 60
4.6: Summary 61

5: Summary and Conclusions 63
5.1: Summary 63
5.2: Future Work 64
5.3: Conclusion 65
References 67

Glossary 70

List of Tables

Table 2.1 Common subexpression elimination — example 4
Table 2.2 Peephole optimization — examples 6
Table 2.3 Code Hoisting — example 8
Table 2.4 Copy propagation — example 8
Table 2.5 Closed and open fragments — example 25
Table 3.1 Substring table from suffix tree 36
Table 3.2 Possibilities during candidate examination 39
Table 4.1 Vertical components for candidate in Figure 4.3 48
Table 4.2 Horizontal partitions for example in Figure 4.2 49
Table 4.3 Possible number of partitionings 52
Table 4.4 FMW compaction — % savings 55
Table 4.5 Heuristic 1 compaction — % savings 56
Table 4.6 Heuristic 2 compaction — % savings 57
Table 4.7 Heuristic 3 compaction — % savings 59
Table 4.8 Best of heuristic — % savings 59
Table 4.9 "Best of heuristic" improvement over FMW 60

List of Figures

Figure 2.1 Conditional coding — encoder/decoder design 13
Figure 2.2 Position of compressor in system — High-level subroutine recognition 15
Figure 2.3 Algorithm for High-level subroutine recognition 16
Figure 2.4 Position of compressor in system — Low-level subroutine recognition . 18
Figure 2.5 Schematic of mechanism for tailored interpreter. 22
Figure 2.6 Suffix tree example — "ababaabb$" 24
Figure 3.1 Compaction example 28
Figure 3.2 Candidate instances — one parameter required 29
Figure 3.3 Candidate instances — turned into a subroutine 30
Figure 3.4 Instances supporting several procedure mappings 31
Figure 3.5 Suffix tree for example — prefix traversal 37
Figure 4.1 One parameter example — Ω-array form 43
Figure 4.2 Complicated partitioning — Ω-array 45
Figure 4.3 Tieing example — Ω-array 46
Figure 4.4 Algorithm 2 — Finding Vertical Partition Components 47
Figure 4.5 Finding a valid tieing partition 50
Figure 4.6 Algorithm 3 — Find tieing valid for partition instances 51
Figure 4.7 Algorithm 4 — partitioning heuristic 1 55
Figure 4.8 Algorithm 5 — partitioning heuristic 2 57
Figure 4.9 Algorithm 6 — partitioning heuristic 3 58

Acknowledgments

Dr. R. N. Horspool is not boilerplated onto this page, and his supervision has helped me learn more of both the craft of research and the communication of it's results. He has constantly preached that technical writing is a skill which must be practiced and perfected, and his suggestions throughout the writing of this thesis have been lessons in themselves. I hope that the reader benefits from my attempts to digest them. I also thank Dr. R. Harrop and Dr. S. Atkins at Simon Fraser University for their continued encouragement and sage advice.

Great hosannas are aimed at whatever force brought Jim Uhl across my path. I thank Jim for always being available for a chat or brainstorming session. On several occasions, his mastery of systems programming explained something that puzzled or frustrated me. He has also explained abstruse concepts to me with great clarity, and this was a great comfort when trying to decipher a paper or book.

Special thanks go to Jan Vitek for his suggestions and enthusiasm, especially since it came at a time when I sorely needed it. Philippe McLean happened to be enrolled in a combinatorics course at the same time I was trying to figure out a combinatorics problem, and I thank him for pointing out the solution in his textbook.

Finally, I thank my wife Susanne. All the books on graduate life maintain that being married enhances one's chance of completing a post-graduate degree. Some people may feel this requirement is a little drastic, but I literally could not have completed this degree without her.

To Joel and Darrel

1 Introduction

Computer programs that are powerful but small have always been desirable. Research has focussed on gains in speed as internal storage constraints recede into the background. Increasing chip densities and falling prices are encouraging larger binaries. However, this satisfies no one. Computer programmers and users often find that there is not enough memory. This is partly due to the quality of code produced by modern compilers, but some observers note that the current memory crunch began with the demand for feature-laden software systems [29]. In the end, smaller binaries need smaller system memories. All storage savings are a benefit if they do not result in slower computer systems.

Internal storage is not the only reason to seek smaller code. Distributed systems depend more and more on code coming in from servers via a network. Smaller binaries decrease the time spent in communication. Stand-alone systems need less time to load from an internal disk. This is already done by network hardware that implements compression and decompression through on-board hardware. Most personal computers now use software-based compression to obtain more room on hard drives, but at a cost in performance. Can a program be written to exploit instruction code redundancy with an eye to preserving performance, and rewritten to take less room? The approach is attractive as it eliminates the decompression stage. The compressor output is an executable binary, not a smaller file that would be unreadable without the appropriate decompressor.

We have implied that the code generators in today's compilers focus on increasing speed because memory is not the main issue. This is slightly unfair as some compiler optimizations for speed also result in less code. However, the general observation is that there is a trade-off between space and speed optimizations. Even if attention is paid to space, today's compilers generate code procedure by procedure as interprocedural data flow analysis is still expensive. The code within procedures is of a high quality, but achieving further space optimizations becomes expensive. Systems with multiple modules add other complications during analysis.

Research has shown that compilers generate code sequences that appear many times throughout the final object file [8][21][9]. We can reduce the size of the object file by keeping only one copy of the sequence, and then replacing all other copies with a procedure call. This technique is called *procedural abstraction*, and is one method for reducing the size of binaries. Throughout the thesis, we use the term *compaction* to describe the space savings achieved through procedural abstraction because the smaller binary is immediately executable, thereby distinguishing the results from *compressed* binaries produced by such UNIX utilities as `compress` or `gzip`; the compressed binaries must be *decompressed* before execution.

We expect to see many more close instruction sequence matches than exact instruction sequence matches. By parameterizing the differences, we can replace all copies with one procedure. Most compaction schemes in the literature acknowledge the possibility of using inexact matches by parameterizing the replacement procedures, but because of the cost of extra storage instructions to copy data into and out of parameters, parameters are ruled out. They therefore constrain instruction matches to be exact. However, if instruction sequences are grouped together such that each group is covered by a procedure with the smallest number of parameters, then greater space savings than parameterless schemes are expected. This raises two questions: how do we group instruction matches into procedures without performing an exhaustive search of the solution space? and how much better is the space savings?

Chapter 2 outlines the previous work on reducing the size of executable code. Chapter 3 explains the compaction and parameterization problem in greater detail. Chapter 4 is a treatment of procedure parameterization, including heuristics for assigning similar sections of code to procedures and selecting procedure parameters with an eye towards good space savings. Chapter 5 summarizes our results and suggests future research directions.

2 Previous Work

As mentioned in the introduction, compiler optimizations for space have received less attention in recent years than those for speed. There are far fewer published papers for the former. Some of the compiler optimizations for speed also save space, and some of these are reviewed. Other techniques use the results of information theory by choosing instruction and data representations that result in the smallest executable images given the statistical properties of a set of target programs; one such approach will be covered. However, assuming that we do not want to add compiler optimizations that transform and rearrange the internal representation, and cannot entertain the complexity and expense of adding hardware that decodes custom instruction sets, then space savings gains are achieved through procedural abstraction. Sections of executable code that match each other can be replaced by subroutine calls, replacing the many sections with one subroutine body.

2.1 Compiler Optimizations Leading to Space Savings

A compiler's back end converts front-end output (the intermediate representation, or IR) into executable code. Early compilers emitted machine code that was far less efficient than hand coded equivalents. Successive compiler writers employed heuristic solutions that improved code quality. These were applied at all compiler stages. Some transformed the source statements and others the machine code, but most concentrated on the IR. In the late 1960s and early 1970s, IR transformations called *optimizations* began to attract more attention, were seriously studied, and, in some cases, were formalized. Most of these optimizations aim to increase the speed of code and, since a guiding intuition is that "the less code there is, the less time is spent executing," space savings are often an extra benefit. Compiler optimizations for modern high performance architectures show that the intuition is not always true, especially in many loop optimizations for code intended for parallel architectures. However, many of the optimizations performed within current compilers continue simultaneously to increase code speed and to reduce code size.

2.1.1 Common Subexpression Elimination (CSE)

Ever since it was first published, this optimization has remained one of the most profitable [7]. Source code expressions which appear in different parts of a program or procedure, and whose argument values do not change between appearances, can be replaced by a temporary variable. In the example from Table 2.1, the expression "`x * y`" appears several times in the C fragment. At each appearance of the expression, the values of `x` and `y` have not changed. Recomputing the expression is unnecessary. It is also a subexpression of "`x * y * z`". This larger expression, which appears again in the last statement, cannot be replaced by a temporary variable. The last argument, `z`, changes value, and the larger expression must be recomputed at "`f`". At no point are the values of `x` and `y` changed, so the temporary `T1` may be used throughout the example. A machine code version of the transformation is shown. Two machine instructions are saved, or 20% of the unoptimized code.

Table 2.1 Common subexpression elimination — example

	Before	After

pseudo source	`c = x * y;`
	`d = x * y * z;`
	`z = 2;`
	`e = x * y;`
	`f = x * y * z;`
		`T1 = x * y;`
		`c = T1;`
		`d = T1 * z;`
		`z = 2;`
		`e = T1;`
		`f = T1 * z;`

assembler	`ld r1, (x)`
	`ld r2, (y)`
	`mul r3, r2, r1`
	`ld r4, (z)`
	`mul r5, r2, r1`
	`mul r5, r5, r4`
	`ld r4, #2`
	`mul r6, r2, r1`
	`mul r7, r2, r1`
	`mul r7, r7, r4;`
		`ld r1, (x)`
		`ld r2, (y)`
		`mul r3, r2, r1`
		`ld r4, (z)`
		`mul r5, r3, r4`
		`ld r4, #2`
		`ld r6, r3`
		`mul r7, r3, r4`

Although space savings through CSE can be substantial, the optimization must still be carefully applied. For example, some numerical algorithms depend on the rounding behavior of a floating point representation. A small difference between the representation expected by the algorithm and that forced by the compiler may cause an algorithm to

produce incorrect results. This occurs if the algorithm no longer terminates because of the change, or if the change results in an algorithm with an error bound exceeding algorithm specifications [10]. If the rounding method were to be changed after the calculation of `d` in the example above, the expression assigned to `e` would need to be recalculated. Replacing the second calculation of "`x * y`" by a load from the temporary variable would then be a modification of the algorithm. This is unacceptable, as any compaction of an executable must avoid changing the meaning of that program.

Another case where savings may be lost occurs if there are significant demands on registers (*register pressure*). As registers are a precious high-speed resource, their allocation is important and difficult. If a CSE transformation stores temporary results in a register, then the number available decreases, leading to situations where the register requirements exceed supply. Spill code is the result, increasing the size of the program [3]. Therefore, if not used wisely, CSE may decrease speed and increase space requirements.

2.1.2 Peephole Optimization

This transformation examines and modifies the machine code generated by the compiler [20]. Despite the excellence of code generated for separate functions and procedures, looking at all of the code at one time may expose redundancies and inefficiencies. These usually occur at locations where the code for one procedure ends and another begins. Redundancies are removed by replacing them with equivalent, shorter instruction sequences. In essence, many different optimizations are applied, such as identification of redundancies, "unreachable code", "flow of control" optimizations, but in a very local region and at a low level [1].

A small window (the peephole) passes over the code, limiting improvements to the code visible within the window. A few examples are shown in Table 2.2. Example 1 replaces a load-store pair with a single load (no need to store a value already known to be in memory). Example 2 removes the unconditional branch: the destination is the next instruction. (This code was probably generated for the last option of a "case" statement.)

The final example changes the conditional branch/unconditional branch pair into an equivalent instruction. Static peephole optimizations nearly always result in space savings.

Table 2.2 Peephole optimization — examples

	Before	After
1	`ld X, r5` `st r5, X`	`ld X, r5`
2	`br A` `A: ...`	`A: ...`

3	`beq A`
	`br B`
	`A: ...`
		`bne B`
		`A: ...`

2.1.3 Unreachable Code Elimination

Programmers often insert debugging statements into code, surrounding them with conditional statements. The conditional expression tests a programmer-specified compile-time constant, and the code is or is not executed at run-time depending on the value of the constant. Some program generators may emit statements with certain parts enabled or disabled, and this usually depends on programmer specified compile-time constants. If the language preprocessor does not support the removal of statements from the stream sent to the compiler (e.g. "`#ifdef ... #endif`" in C), then the final program will contain sections that will never be executed. This code is called *unreachable*. Eliminating it from the program will not change the output, and may save considerable space [2].

2.1.4 Useless Code Elimination

In the 1970s, data flow analysis methods were formalized and used in compilers. Relationships between the values of variables are found through these methods. One example is *live variable analysis* [12]. A variable is said to be live at a specific point in the program if its value is used after that point at least once on some execution path. If the variable on the left hand side of an assignment is not live after that operation, then the computations on the right hand side may be eliminated, including the assignment. Since

there is no longer a need to compute the expression or store its value, space can be saved by omitting both the operation and the assignment from the final executable [1]. Unreachable and useless code is sometimes referred to as *dead code.*

2.1.5 Code Hoisting

A generalized form of code motion, it moves *very busy expressions* to a point where space is saved, but not necessarily time. Intuitively, an expression is very busy at a specific point in a program's execution if (a) it will be evaluated on all possible execution paths, and (b) all evaluations occur before any change in the value of an expression argument. That is, there is at least one evaluation of the expression, no matter what control flow path is taken after the very busy point. Instructions required to compute the expressions are removed from each of the evaluation locations, and are replaced there by an assignment from a temporary variable. At the very busy point, one copy of the code for the expression is inserted, and the result is assigned to the temporary variable [1].

In the pseudocode example from Table 2.3, the exponentiation expression "`c ** d`" is very busy at a point just before the "`if`" statement. Every possible flow of control from this point to the bottom of the fragment will evaluate the expression. As exponentiation may require many instructions, a space savings occurs when the code is hoisted to the very busy point. Line counts are deceptive in this example: code hoisting is sometimes mistaken for a general form of loop-invariant code motion. However, what is important is that the body of code needed to calculate "`c ** d`" appears four times in the code before it is transformed, and only once after.

The results are similar to CSE, but the motivation for code hoisting is space saving, not speed improvement.

2.1.6 Copy Propagation

Copy propagation eliminates redundant names for the same value (see example in Table 2.4). In turn, fewer variables may be needed. Although this appears to be the substitution of many variables by one variable, which by itself does not reduce the number of instructions, copy propagation directly enables other space saving optimizations. Register

Table 2.3 Code Hoisting — example

before	after

`if ( expr1 ) {`
`  switch( expr2 ) {`
`  case 1: a = c ** d;`
`  break;`
`  case 2: a = c ** d;`
`  a = a ** 2;`
`  break;`
`  default: b = c ** d;`
`  break;`
`  }`
`  a = b * 2;`
`} else {`
`  a = c ** 4;`
`  f = c ** d;`
`}`
`print a;`
	`T1 = c ** d;`
	`if ( expr1 ) {`
	`  switch( expr2 ) {`
	`  case 1: a = T1;`
	`  break;`
	`  case 2: a = T1;`
	`  a = a ** 2;`
	`  break;`
	`  default: b = T1;`
	`  break;`
	`  }`
	`  a = b * 2;`
	`} else {`
	`  a = c ** 4;`
	`  f = T1;`
	`}`
	`print a;`

pressure is lightened as fewer values need to be kept in registers, reducing the contribution of register spill code to the executable's size [3].

Table 2.4 Copy propagation — example

before	after

`a = x * x;`
`y = a;`
`z = foo( y - 4 );`
`z = a;`
`b = foo( z - 9 );`
	`a = x * x;`
	`z = foo( a - 4 );`
	`b = foo( a - 9 );`

2.1.7 Leaf-Procedure Optimization

If a procedure does not call any other routines, it is said to be a leaf-procedure because these appear as leaves in the call graph. Several observations lead to space optimizations [3]. For instance, the register which holds the return address need not be saved or restored in anticipation of another procedure call, thereby saving two instructions. Allocating stack

space is unnecessary if no local variables are used, eliminating the code needed to set up the stack frame.

2.1.8 Parameter Promotion

In some languages such as FORTRAN, all procedure parameters use the "call-by-reference" mechanism, and their values are read from memory by the procedure. Computations need these values, so they are typically held in registers. If the procedure calls other procedures, and passes in some of the same parameters, the memory reads are duplicated. Register demands will increase because the compiler cannot determine that the data is already stored in registers. Why read the same memory locations twice? Why store the newly loaded values back into memory because of register spillage? Instead we can use the register itself to pass in the data to procedures receiving the parameter. Changes to the parameters can be applied to the registers, and saved back to memory when exiting the procedure level that first received the parameter.

Readers interested in detailed examples of *leaf procedure* and *parameter promotion* optimizations are encouraged to refer to "Procedure Call Transformations" section in the survey of optimization techniques for high-performance computers by Bacon, Graham and Sharp [3].

2.1.9 Span-Dependent Instruction Optimization

Unlike RISC opcodes, CISC opcodes vary in size. For instance, branch instructions with nearby destination addresses may need only one or two bytes to indicate a destination address. Far destinations require more than two bytes to specify the address, extending the length of branch instructions to three or four bytes. If code blocks are carefully placed, then the contribution of the branch instruction sizes to execution size can minimized. Ideally, all branches would require only two bytes, but as this is not always possible, algorithm exists that maximize the short branches [27]. The larger the number of short branches, the greater the space savings.

2.2 Exploiting Similarity of Code

There was much research into compiler optimizations in the late sixties and early seventies. However, optimization algorithms were still described by their implementations. Identifying common links between different transformations was difficult.

A formulation describing optimization techniques in a machine independent manner was developed by Geschke [9]. It expressed the optimizations through the relationships between source language statements. Statement orderings and dependencies are found: e.g. whether statement B comes before statement A in the source text; whether B is needed by A; and whether the order of computing A and B is invariant. Most optimizations of the day could then be specified and implemented in a programming language via these relations [30].

One of the intriguing ideas in the dissertation by Geschke is the *strongly similar subroutine* optimization. It appears promising, but seems to have not been pursued beyond the dissertation. Two sub-trees in the intermediate representation (which is itself a tree) are arguments for a similarity function. This function's return value indicates the desirability of covering the two sub-trees with the code of one subroutine, with low values indicating good space savings potential. Put another way, the value is an indication of the cost of providing one subroutine body that will implement two different portions of the code. Several similarity functions are presented; one takes the parameter costs into consideration; another returns larger values if the differences between the subtrees occur in the middle of code, and low values if the differences are at the beginning or end.

Finding sub-trees that are close matches and suitable for subroutines is not the same as actually turning them into subroutines. As Geschke phrases it, "what is desirable may not be feasible." Procedural abstraction, or more accurately "procedural extraction", must build the subroutine such that space and/or time of the emitted code is lower than if the subroutine was not formed. Even though the dissertation discussed the promise of the approach, the optimization was not included in a subsequent related work [30]. Other

researchers have picked up on the idea of procedural abstraction, but there are still other ways of exploiting code redundancy to save space.

2.3 Matching Code Representation to Code Properties

Compiler optimizations and procedural abstraction transform the executable code such that it requires less space to store and/or less time to execute. If the instruction decoding hardware can accept instructions with sizes varying by units of a bit, then there are additional possibilities. An information theoretic approach by Hehner [13] analyzes and recodes executable files based on the statistical properties of a sample of programs. These transformed files are smaller than the originals, but they require special hardware for instruction and data fetches (usually a custom decoder) and/or run-time support (interpretive execution).

Part of the method revolves around finding the fewest number of bits necessary to represent the opcodes. A simple scheme examines a sample program set, finding the frequency of each opcode's use. Frequencies are turned into probabilities, and these in turn may be used to find the smallest possible encodings for the opcodes, such as those provided by Huffman encoding [15]. Frequently occurring opcodes are represented by fewer bits than those which are rarely used. The instructions are re-encoded, a dictionary of encodings attached, and a smaller executable is the result.

However, instructions are usually not randomly distributed through an executable. Analysis of sample programs will show that some instruction pairs appear more frequently than others. In some architectures, one such pair may be a load into a register before an arithmetic operation. An analogy can be taken from the spelling of English words: if the letter "q" occurs in a word, then there is a high probability that a "u" is the next letter. Successors with a high probability are encoded in fewer bits than those with low probability.

Probabilities may be found for m-tuples of instructions (for some fixed $m>1$), and used to find minimally sized encodings of the tuples. Re-encoding occurs as before. Taken to its extreme, m may be the number of instructions in the executable, in which case the whole

program can be represented by one opcode, but the dictionary will have only one entry — the program itself. There would be no space savings. The value of *m* must be chosen such that the size of the program's new encoding plus the size of the dictionary of encodings is minimal.

Hehner examines two methods that exploit conditional probabilities between opcodes: iterative pairing and conditional coding.

Iterative pairing makes repeated passes over the executable. Candidates are pairs of opcodes that, when replaced by a single new opcode, provide a space savings over and above a limit specified by the user. At each pass, the best compression candidate is found. (The pairs may be generalized to *m*- tuples.) When no more candidates exist, iteration stops. Having created an augmented instruction set — the original opcode set along with opcodes for compressed pairs — the algorithm recodes the executable, appending a dictionary of the new opcodes.

A more aggressive algorithm recognizes that the final encoding may represent a local maxima of savings in the solution space of compression. By continuing the pairings where the previous algorithm left off, the compression of the candidates will *increase* code size. However, new candidates with excellent space savings may eventually appear. The final result could be a better compression of the executable than if the search for candidates had stopped in accordance with the original algorithm.

Conditional coding (see Figure 2.1) compresses by using *m*-tuples, for which conditioning classes of order *m*-1 conditional probabilities are needed. In the figure, the instruction to be encoded is in the operation register (upper right). Previous instructions are in the context register (upper left), with the register acting as a sliding window over the code. When the data from these two registers are passed to the encoder, the minimum redundancy encoding of the *m*-tuple is placed in the code register (bottom) and is output to the new executable file.

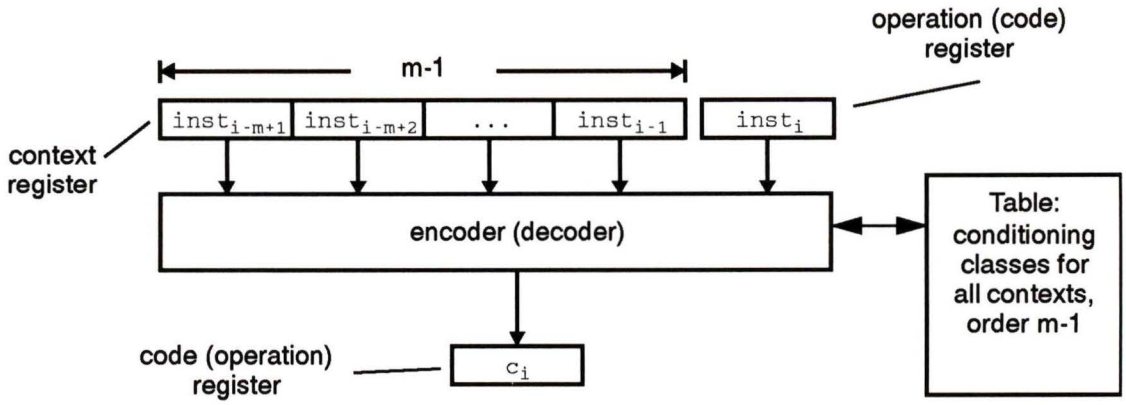

Figure 2.1 Conditional coding — encoder/decoder design

Let the probability that $inst_i$ follows $inst_{i-(m-1)}$ $inst_{i-(m-2)}$... $inst_{i-1}$ be equal to p_i. The size of code c_i corresponds to the number of bits required to represent the opcode in the context given, which is [6]:

$$-\log_2 p_i$$

An encoding whose size is equal to this number of bits is a minimal redundancy encoding. The decoding operation is similar to encoding, and the role of registers in decoding is indicated in parentheses in the figure.

Experimental results were obtained by compressing the code of the IBM 360 version of XCOM, a compiler for the XPL programming language. An earlier transformation of the code changed instruction operands from a fixed sized to a variably sized representation. Using iterative pairing, the re-encoding of the XCOM code resulted in a 77.4% reduction in the space required for opcodes. (Data such as addresses and constants were encoded using another space saving algorithm.) Conditional coding using 4-tuples yielded a 80.2% savings in opcode space.

These results do have a cost. It is difficult and expensive to design and implement an instruction decoder for opcodes which vary in size down to a bit. Speed is required; therefore the decoding logic must be implemented in hardware. However, until the late 1970s, when internal memory implementations switched from magnetic core technology to integrated circuits, significant reductions in memory requirements of programs were worth

the cost of increased decoder complexity. As well as being useful for reducing the size of executables, this research also helped computer architecture design. It applied rigour to the selection of instruction sets and opcode sizes, analyzing the properties of programs that are expected to run on a specific machine design.

2.4 Compilation to Compact Code

Using PL/I on an IBM System/370 as the experimental platform, Marks examined three code compaction methods [21]. These approaches compress executables by replacing long code sections with short subroutine calls.

High-level subroutine recognition identifies redundancy in a programmer's source. The analysis uses the internal representation available immediately after lexing and parsing, with results matched against the original source statements. Each subroutine candidate, which is made up of source language statements, is assessed by the programmer who must then accept or reject it. *Low-level subroutine recognition* identifies repeated sections in the object code where the repetition has been introduced by the actions of the compiler. Compilers tend to produce the same instruction sequences over and over, and the method exploits this. Both the high- and low-level approaches modify some representation of the program, but after code generation, the smaller executables are loaded and run by the computer without any extra processing.

Tailored interpretation takes place at the same level as low-level subroutine recognition, but takes advantage of very small sequences rejected as "not profitable" by the low-level scheme. The original object code sequences are replaced by custom opcodes, and the compressed code must be interpreted at run time. This produces very high space savings but at a great cost in speed.

2.4.1 High-Level Subroutine Recognition

To improve productivity, some programmers often write code with the assistance of tools such as `lex`, `yacc`, and others that generate source language stubs handling communication or message passing details for some development environments. The

programmer saves time, and the generated source code is guaranteed to match the programmer's specifications. If these tools are used several times during development, the same source code sequences might turn up several times. Programmers also repeat themselves when writing their own code, using idioms, and often within the same procedure. High-level subroutine recognition takes advantage of these types of repetition. In many ways it is easier to compress at this level as we are still some distance away from the complexity of the machine code. Ease in finding candidates comes at a cost, as they tend to be short sequences of source lines, and replacing these sequences by subroutine calls is not very profitable. Nearly identical sequences could be covered by one subroutine, but the exact cost of parameterizing a routine is difficult to calculate because of the distance from the machine code. Figure 2.2, shows the position of the compressor within the compiler.

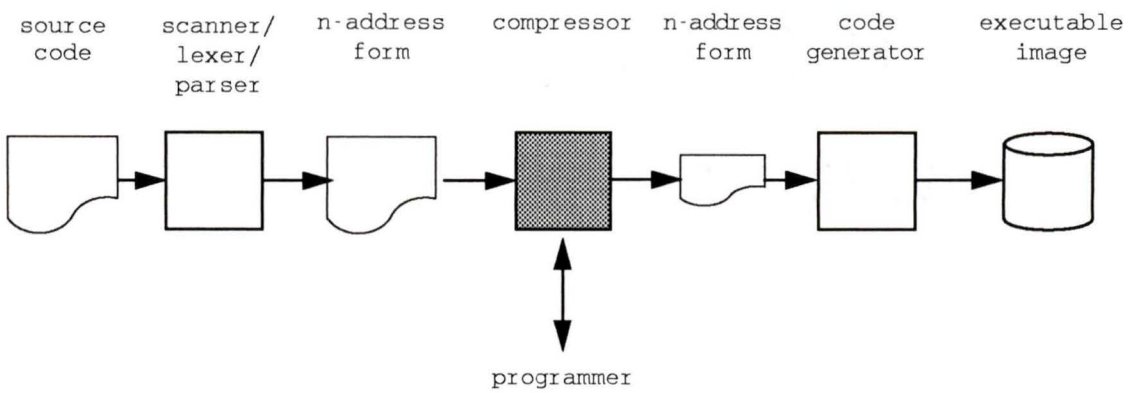

Figure 2.2 Position of compressor in system — High-level subroutine recognition

Subroutine recognition: The PL/I compiler's *n-address* Object-Machine Independent Representation (OMIR) was used by Marks. In its search for candidates, the algorithm treats the n-address form as a string of operators and operands. That is, each operator and operand is a "character", and the whole program is one long string. Substrings match if they:

- agree in all their operators (if substrings *A* and *B* belong to the same candidate, then the i^{th} operator "character" in *A* matches the i^{th} operator "character" in *B*, for all *i*); and

- differ in at most three other "characters" (this allows for up to three parameters).

```
for each operator Op do
    M[] := empty;
    P1 := first occurrence of Op in the OMRI;
    while there is a position P2 after P1 in the OMRI do
        m := LongestPossibleMatch( P1, P2 );
        add P1, P2 to list M[m];
        P1 := P2;
    end while;
    find the best subroutine candidate B, and query the programmer;
    if programmer accepts B then
         make subroutine based on B;
         replace OMIR instructions matching positions in B;
    end if;
end for;
```

Figure 2.3 Algorithm for High-level subroutine recognition

For each operator in the instruction set, Figure 2.3's algorithm examines all substrings that begin with that particular operator. Since operators must match, constraining the start of substrings to operator positions is a useful optimization. Pairs of positions are taken together, and the longest possible matches (using the two-part definition from above) that start from these positions are recorded.

Evaluating candidates: After searching for all matches, the algorithm finds the best subroutine candidate from the matches that begin with a particular operator. The programmer is allowed to accept or reject this candidate after seeing the line numbers corresponding to the suggested subroutine. If accepted, the subroutine body is added to the end of the program, and all matches in the OMIR are replaced by a subroutine call, each call including the appropriate parameters.

All operators are examined in turn. To keep the algorithm simple, influences between subroutine choices are ignored — the effect of compressing one subroutine before another is not examined. Marks does not indicate how much potential space savings are lost by this design decision.

Any manipulation of the source text is avoided even though rearranging the text may improve the opportunities for compaction. One reason given for this is that it requires flow analysis, but surely this should be simple for *n*-address code. A more convincing reason against manipulation is that rearrangement might increase the distance between the source code and the transformed OMIR. The greater the distance, the more difficult it is for the programmer to assess the effect of accepting the subroutine. Marks notices that there are some situations where rearrangements are overkill. For example, two sections may differ where one contains an extra statement in the middle. Splitting the sections into two is the simplest solution, with similar code before and after the statement forming the new subroutines. A set of heuristics such as this could identify and exploit similar situations without the need for data flow analysis.

Results: The space savings of this scheme are poor — 1% to 2%. No mention is made of the influence of PL/I, or the intermediate representation, on the results. It is unclear how the results would transfer to languages currently in use, such as C or C++. One suggestion is that the programmer's time may be better spent rewriting or redesigning the code than in accepting or rejecting compaction candidates! At this level, automatic methods of accepting subroutine candidates were not examined.

2.4.2 Low-level subroutine recognition

Intuition leads us to expect more code repetition at the machine level than higher up in the source code. Marks follows this thinking by applying the high-level subroutine recognition algorithm to a version of machine code that he called *low-level code*. Features of this representation include instruction lengths that correspond to the final code lengths. Branch destinations are not resolved, but since the lengths of final instructions are known, estimates of space savings are accurate. Registers have already been allocated, and their reassignment would be expensive. Arguments must be passed into routines via registers, but if all registers are tied up, then subroutine parameterization is expensive. To avoid any register reassignment during subroutines calls, parameters are not allowed.

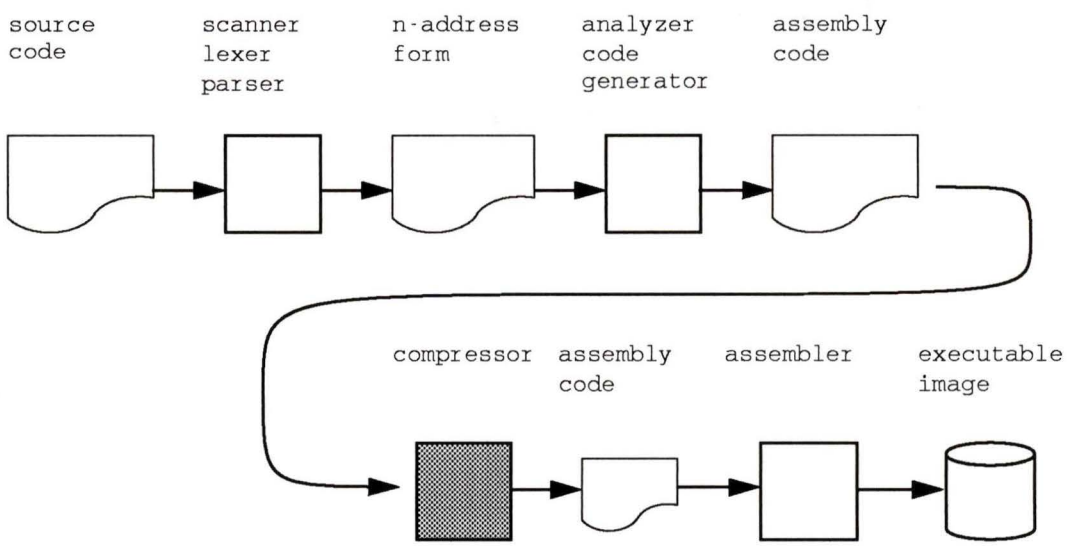

Figure 2.4 Position of compressor in system — Low-level subroutine recognition

Subroutine recognition: The algorithm differs slightly from that in Section 2.4.1. Before finding matches, a hash chain is constructed for each opcode, linking together all positions where that opcode occurs. This reduces the time spent searching through the code. Each hash-chain is examined in turn, with matches performed on every pair of positions from the chain. Only identical code matches are accepted, although extra processing is added to examine "if—then—else" constructs. After a match is made, the pair of opcode positions, the length of the match, and the opcode value itself are formed into a group.

When all hash-chains have been processed, transitive closure merges groups together. After the merge, each group contains all the code positions that start with the same opcode and match to the same length. The groups become subroutine candidates.

Evaluating candidates: Groups are placed into a list, and the list is ordered by each group's space savings potential. Those groups corresponding to subroutines which are too short or too infrequent are eliminated as they result in little or no space savings. Overlapping and nested groups are retained even though some compaction choices become difficult.

To achieve optimal compaction, searching through all the possible orderings of subroutine construction would be necessary. The number of orderings increase if groups are allowed to be broken up. New compaction possibilities appear when additional groups are formed through the splitting of overlapping groups. Deciding whether and which nested routines should be compressed will make analysis even more difficult. A moderately sized program would end up needing a lot of computing power during compaction. Since finding an optimal solution is too costly, a greedy heuristic is used, which, among other things, does not consider the effect of subroutine choices on the speed of the final code.

Subroutine selection begins by taking the group at the head of the list and adding it to a collection of "best groups." Proceeding down the list, groups are added to the "best" collection in the order they appear in the list, but with some constraints. None of the groups in the collection are allowed to be nested, and no groups may overlap. Marks pointed out that it is short sighted to eliminate less profitable groups because they overlap a more profitable group. The algorithm is searching for the best net gain: if rejecting the good group and accepting the overlapping bad groups leads to a better net gain, then this is the decision taken. Different orderings of the original list (best gain to worst; worst gain to best; randomized) do not result in great variations in the final space savings.

The possibility of using parameters in subroutines was raised, but it was suggested that the benefit from this depended on a machine's architecture. If registers are used to pass parameters into subroutines, then these expensive resources are not available to the rest of the code. A result would be poorer overall code quality for the rest of the program. This is reasonable given that register allocation is more difficult when some registers have already been used for parameter passing. Any other parameter passing mechanism would be more space intensive, wiping out some of the benefits of the space savings.

Results: Reported compaction ratios were encouraging, with space savings ranging from 5.0% to 20.1%, averaging out at 15%. Compressed code executes 15% slower. Larger programs were better compressed, but the correlation between compaction ratios and low-level code size was non-linear. More efficient instruction matching techniques would

reduce the contribution of compaction to compile times. No numbers were given for the potential loss in compaction if parameters were allowed— it may indeed depend on the machine architecture, but a rough estimate would have been helpful.

2.4.3 Tailored Interpretation

After examining many code examples, Marks observed that groups rejected during low-level compaction could be used, but only if a cheaper subroutine calling mechanism could be found. This is intuitively appealing. We expect many matches that are two or three instructions long, many more than those that are, say, ten instructions long. However, all of the subroutine calling mechanisms require space. As the size of the average subroutine gets smaller, the limit to compaction becomes the size of the calling sequence.

A solution to the calling sequence problem is to *tailor* the instruction set, matching the characteristics of the code. The tailored set is implemented through software.This is similar in spirit to Hehner's "iterative pairing" [13]. Marks acknowledges the influence of the idea, but instead of putting the effort into the hardware encoder/decoder, he puts it into an interpreter.

Opcodes not used by the program take on a new meaning, by becoming indices into a table of the subroutines. The subroutines are unique to each program. Compressed code sizes include the space needed for the compressed code, the interpreter, and its tables. Figure 2.5 is a schematic of the system as it interprets one instruction.

Compressed code will now be made up of three different types of instructions:

- Ordinary machine instructions executed directly by the hardware with no interpretation.
- Parameterless subroutines.
- Parameterized subroutines.

Arguments for the last instruction type are placed after the opcode, with the interpreter copying the argument value into the subroutine code. As shown in Figure 2.5, R2 in the unparameterized code corresponds to the parameter position in the parameterized routine. The grey data value from the compressed code's instruction is copied into the routine. A

maximum of one parameter per subroutine is allowed, a reasonable requirement considering the small size of the subroutines.

Subroutine recognition: The method is taken from low-level compaction, but with the exception that matches may allow one operand position to differ. Nested subroutine calls are also permitted.

Evaluating Candidates: The selection of candidates is more complicated because of parameters and nesting, and is implemented through a multi-pass algorithm:

1. A list of groups is created in order of descending space savings potential.
2. Groups are considered in list order, provided that the group (a) has the largest space savings of those lower in the list and (b) does not overlap a selection with a greater space savings. As each group is selected, it is converted into a subroutine. Occurrences in the main program are replaced by a call to the new subroutine using the tailored instruction set.
3. If groups were rejected because of step 2, repeat the algorithm from step 1.

After all possible groups are compressed, the interpreter table will be complete, and the compressed code is saved along with the table and the code for the interpreter itself.

Results: At 50%, the compression ratio is impressively high, and is due to the use of parameters and to the utilization of smaller (two or three instruction) matches. Compression in one test case could have exceeded 50%, but the lack of unused opcodes imposed a limit. However, the code may run up to 15 times slower, and is an indication of the cost of interpreting each tailored instruction. Not all instructions depend on the interpreter — operating system calls and noninterpreted subroutines run at full System/370 speed. If space savings are needed, but specific sections of the code must run at full speed, then the "full speed" code could be tagged, and the compiler would not compress the tagged code.

2.5 Analyzing and Compressing Assembly Code

In a conference paper by Fraser, Myers and Wendt [8], code is compressed at the same level as Marks' low-level algorithm. The target machine is a PDP-11, and the compressor runs

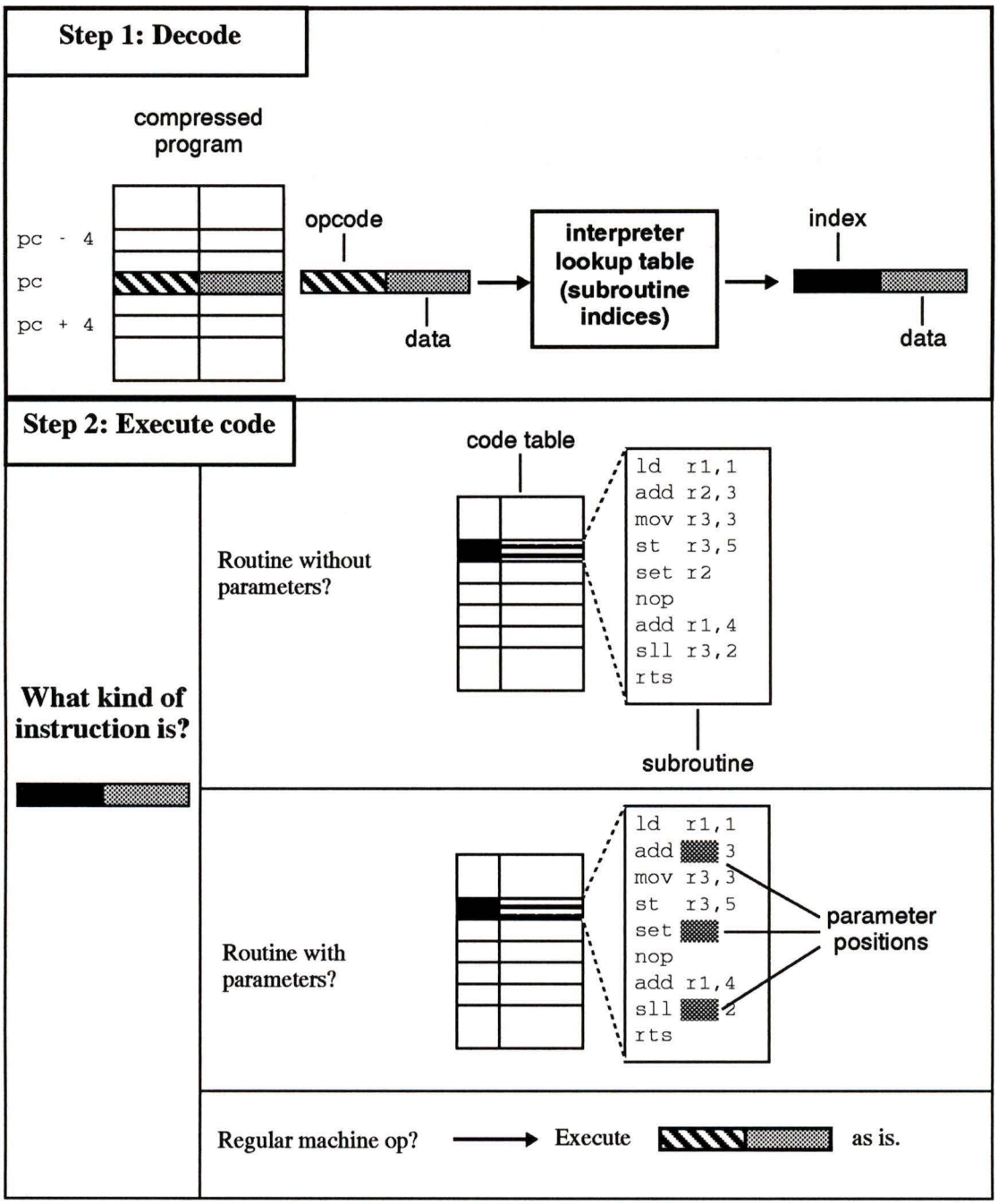

Figure 2.5 Schematic of mechanism for tailored interpreter.

on a VAX 11/780, but there is no description of the high-level language or its compiler. Compaction occurs at the point after the generation of assembly code and before the assembly. Their technique (abbreviated to "FMW") differs from that of Marks in:

- the algorithm for subroutine recognition, based on the construction of a *suffix tree*; and
- the algorithm for identifying and classifying candidates as *open* or *closed*, where the cost of converting the former into subroutines is less than that for the latter.

Subroutine recognition: The major computational effort during compaction takes place while finding and evaluating subroutine candidates. Borrowing an idea from text compression, Fraser et al. reduce the search costs by building a suffix tree. While there are many efficient string searching algorithms, repeated substring searches over the same string benefits from an auxiliary index to the main string. The suffix tree is an example of such an index (see Figure 2.6 for an example).

A substring is described by the path from the suffix tree root node to an inner node or to a leaf node. The text of the substring is formed when the path edge labels are concatenated together. If an inner node has *m* leaf nodes descendants, then there are *m* occurrences of the substring described by the path from the root. Leaf nodes hold the starting positions of these substrings, and there are as many of these nodes as there are characters in the main string.

For example, the suffix tree in Figure 2.6 has been built for the string "`ababaabb`", which is terminated by the end-of-string symbol "`$`". The path of heavy edges from the root node to a leaf corresponds to the substring "`abb`", which starts at position 6 in the main string. Three heavily circled leaf nodes containing `3`, `1` and `6` are the leaf node descendants of the shaded internal node. Notice that the path from the root to the shaded node constructs "`ab`". This short substring appears three times in the main string, starting at positions 1, 3 and 6. Suffix trees for assembly language program are much larger, and require algorithms that build the index in linear time and reasonable (polynomial) space [19]. As it contains indexes to all substrings, we can also get answers to such questions as, "What is the longest

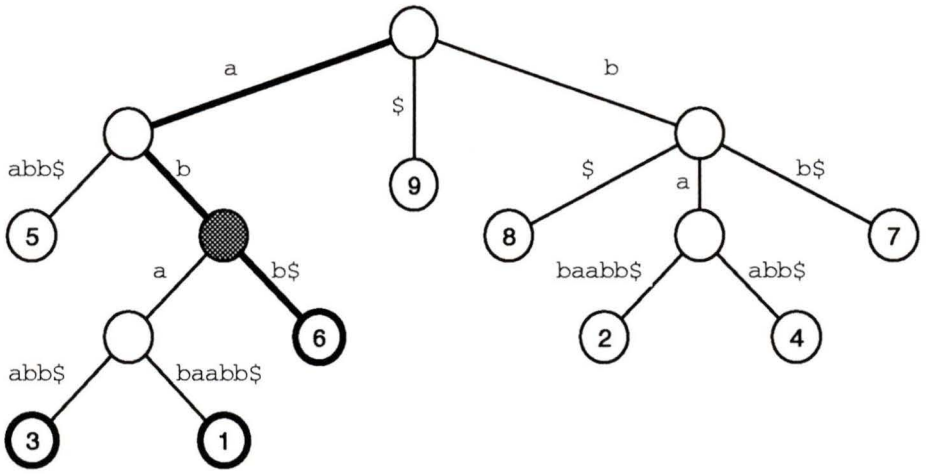

Figure 2.6 Suffix tree example — "ababaabb$"

substring that occurs in *k* places?" [25] Other researchers have used the suffix tree as the basis for regular text compression [23].

By treating the assembly language text as one string, substrings with several occurrences correspond to potential subroutines. Assembler directives are flushed directly to the compressed file as these must be left unmodified. For the rest of the assembly code, the authors restrict the possible matches in the text: two instructions at different positions are the same if the instruction texts are identical. Instructions that differ by one or more operands are rejected, so this technique does not allow for parameterized subroutines. This restriction is relaxed for branches, allowing for the possibility that different labels may refer to the same destination.

Evaluating candidates: After finding the substrings, or "fragments," each is evaluated for subroutine potential. As with Mark's low-level subroutine recognition, infrequent or short fragments are discarded. The remaining fragments are grouped into two types. *Open fragments* end with a non-relative branch. When turned into subroutines, they are accessed by an unconditional branch, which is inexpensive. *Closed fragments* exit to the instruction following the fragment. To preserve the original control flow when using closed fragments, only the subroutine call may be used to access its corresponding routine. (See Table 2.5 for examples of open and closed types, where fragments are in boldface.) Because of the extra

instructions needed when converting to a subroutine, closed fragments will not always result in a space savings. An *evaluation function* estimates the space savings potential of a fragment, taking into account these restrictions. Fragments of either type that produce "negative" savings are discarded.

Table 2.5 Closed and open fragments — example

closed	open
...	...
push D, [sp]	st B, r6
ld r1, A	**push D, [sp]**
ld r2, B	**push E, [sp-4]**
mul r3, r2, r1	**br L5**
st r3, B	L1: add r3, r3, r4
...	...
push E, [sp]	sub r2, r1, r4
ld r1, A	**push D, [sp]**
ld r2, B	**push E, [sp-4]**
mul r3, r2, r1	**br L5**
mul r4, r4, r3	L2: add r4, r1, r1
...	...

A fragment's control flow *falls off the end* if its last instruction is not an unconditional branch or subroutine return. *Falling through to a fragment* occurs when it is not accessed via a branch or a subroutine call. Since open fragments are accessed by an unconditional jump or a fall through, they cannot exit by falling off the end. Relocating the fragment's routine will change instruction order as we cannot guarantee that the code following the routine has not changed. Closed fragments are entered by a subroutine call, so falling into them or entering them by a branch will result in incorrect code. Open and closed fragments must contain the destination of any internal relative branch, otherwise the flow of control is not guaranteed to reach the exit of the fragment. Overlapping fragments are discarded. Finally, closed fragments must pop off the stack exactly what they pushed on. Without this final restriction, the return address may not be on the top of the stack at fragment exit (subroutine return).

An optimal algorithm is not used to choose fragments. Instead, a greedy algorithm orders the fragments by the value of the evaluation function, examining them in order of

decreasing space savings. If a fragment satisfies all criteria, it is converted into a subroutine. The code for the body is placed out-of-line. All occurrences of the fragment in the assembly text are replaced by unconditional branches (open fragment) or calls (closed fragment) to the subroutine. The process is repeated until there are no fragments left. Different fragment orderings may result in better compaction, but such orderings were not investigated by the authors.

Results: After applying the compaction algorithm to a set of over 150 UNIX utilities, the authors reported space savings ranging from 0% to 39%, averaging at 7%. The compressed binaries took 1 to 5% more CPU time, but this was offset by reduced loading time, and the result was a 11% time savings. However, the wide range of compaction ratios raises questions. Why does some code compress well? Does it matter if programs are large or small? It is unclear how well the figures would hold for instructions sets of architectures currently in use. Much has changed since 1984, both in machine architectures (RISC, superscalar) and compiler technology (instruction scheduling, software pipelining). When compared to the results of Marks's low-level scheme, the average of 7% is poor. This may be due to the size of UNIX utilities: if applied to the PL/I compiler code, the average compaction might improve. How much the FMW algorithm would benefit from parameterized subroutines is unclear.

2.6 Looking Forward

There are three parts to the code compaction problem:

- finding subroutine candidates; and
- evaluating subroutine candidates; and
- choosing subroutine candidates.

Greedy algorithms solving the last problem were used in the two previous papers. They were used despite the possibilities for better space savings possible through careful ordering of subroutine construction. The impact that small fragment differences have on compaction rates was ignored. Solutions to the second problem, when applied at the level of assembly code, could not deal with candidates that needed parameters.

The first problem is solved efficiently through the use of suffix trees, and this is what we use to find subroutine candidates. We are interested in seeing the effect on code compaction on a search of the solution space of subroutine parameterizations, which we investigate in Chapters 3 and 4. A Sun 4 SPARC workstation was the platform for our experiments and our algorithm evaluation. UNIX utility binaries were used as input for the compaction algorithms. (The specific UNIX utilities used in the FMW paper were not specified, but that will not rule out comparisons with their results). Since the executable code is disassembled, no compilers need to be modified: all compaction is performed on code that is already loadable and executable. Chapter 5 summarizes the results; it suggests possible improvements and future research directions.

3 Compaction and its Problems

Procedural abstraction replaces many instructions with a call to one procedure. This achieves *compaction*, which is a transformation of a program into a smaller binary that is immediately loadable and executable. The compacted binary does not need "uncompaction" nor run-time support (e.g. interpreters as in Marks "tailored interpretation"). Since "similar sections of code" is the intuition behind procedural abstraction, we can visually identify repeated appearances of the same code, thereby easily developing transformations leading to space savings. For instance, the left-side program in

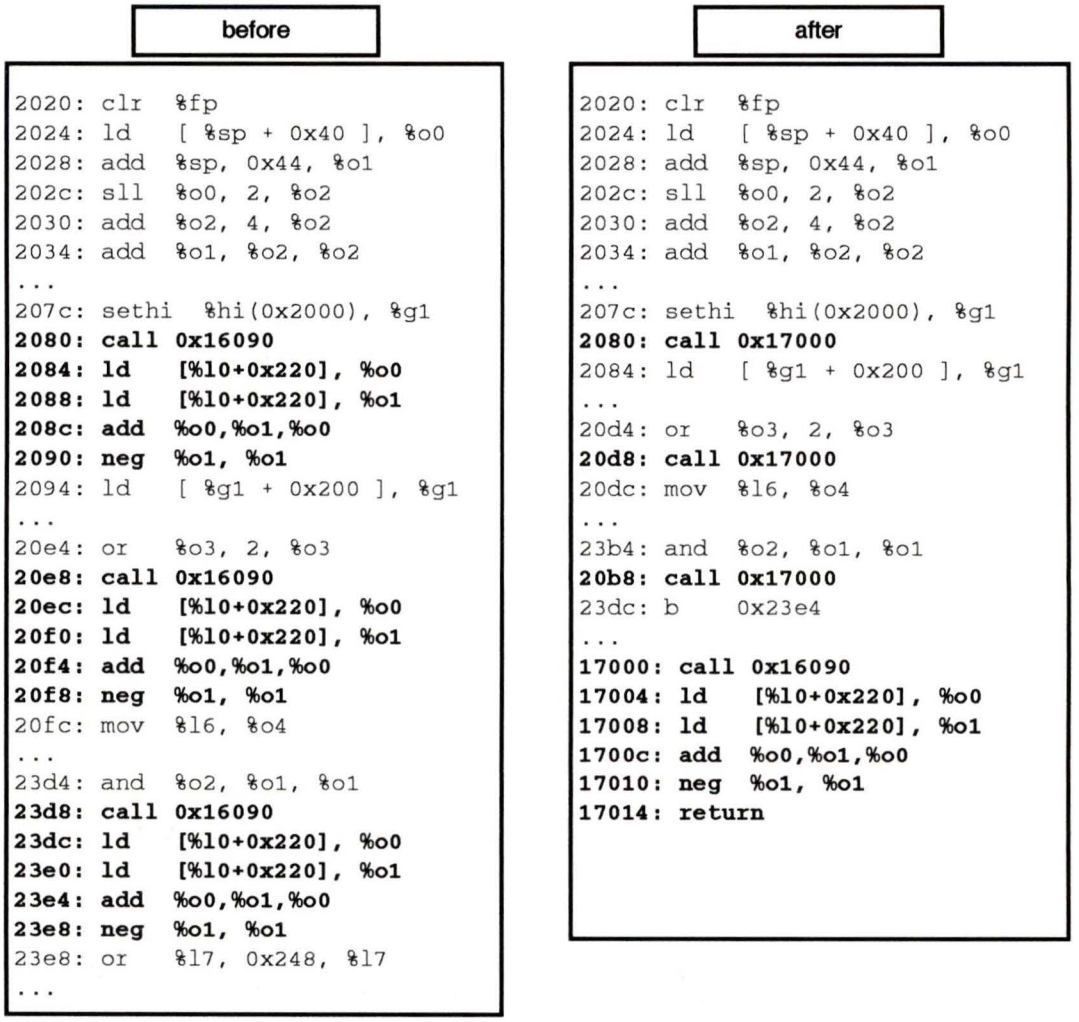

Figure 3.1 Compaction example

Figure 3.1 is an abbreviated printout of a disassembled binary, and similar sections of code

are highlighted by boldface type. On the right-hand side, the boldfaced code from the left has been replaced by subroutine calls, and one copy of the body attached to the end of the program.

Space savings are modest: 15 instructions are removed from the original, and 9 are added, resulting in a net savings of 6 instructions. Each of the replaced segments of code are identical and need no parameters. If many such transformations are applied, substantial savings may result.

3.1 The Problem

The FMW technique has already handled the "no parameter" case. What happens to compaction if the sections of code are similar but not identical? We have copied some similar sections of code from a program and placed them in Figure 3.2. Their differences are highlighted by boldface type. In the same manner as a high-level language procedure call, we pass in the differing value to the new routine via a *parameter*. Figure 3.3 shows the result of procedural abstraction, including the copying of values into the parameter passing variable. The procedure body has been placed in the region beyond the "end of text" of the original program.

```
2540: ld [%o0+0x30],%o0
2544: mov 0x9400,%o1
2548: mov %l4, %o5
254c: b 0x16210
```

```
25f0: ld [%o0+0x30],%o0
25f4: mov 0xa000,%o1
25f8: mov %l4, %o5
25fc: b 0x16210
```

```
263c: ld [%o0+0x30],%o0
2640: mov 0xa000,%o1
2644: mov %l4, %o5
2648: b 0x16210
```

```
25d0: ld [%o0+0x30],%o0
25d4: mov 0xa400,%o1
25d8: mov %l4, %o5
25dc: b 0x16210
```

```
2610: ld [%o0+0x30],%o0
2614: mov 0xa400,%o1
2618: mov %l4, %o5
261c: b 0x16210
```

```
2800: ld [%o0+0x30],%o0
2804: mov 0xb000,%o1
2808: mov %l4, %o5
280c: b 0x16210
```

Figure 3.2 Candidate instances — one parameter required

In Figure 3.2 it is easy to find the number and the positions of parameters by simply comparing operand values for all instances, one operand position at a time. If two instances have different values at one position, then a subroutine covering the instances needs a

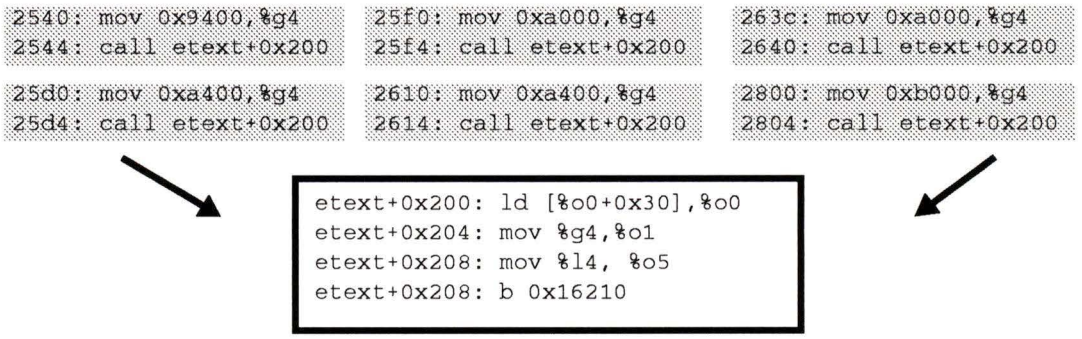

Figure 3.3 Candidate instances — turned into a subroutine

parameter at that location. Our example above uses one parameter for the *first* operand of the *second* instruction. As more instances are covered by one subroutine body, the compaction rate improves. One way of covering many instances by one body is by passing in the differences to the procedure. In our example, this is done for differences in operand values, but it could also be applied to differences in instruction operations. This thesis will concentrate on the former.

The compaction problem reveals a trade-off between coverage and parameters: if a procedure covers too many instance differences then parameter costs will erase space savings, but allowing no parameters can eliminate many opportunities for compaction. We propose a compromise by reducing the number of parameters by covering the instances with *several* procedure bodies (not just one), mapping instances to procedures in such a way that the number of parameters lead to costs that are tolerable.

For example, covering all five instances in Figure 3.4 by one procedure would require many parameters. However, there is no need to turn all of them into procedure calls. Excluding E, we could group together the others (A, B, C and D) and cover them with one procedure. Operand values that are identical across each instance require only one parameter, thereby reducing the number of instructions needed for parameter copying. Another grouping takes advantage of the fact that instances B and C are identical, and covering these two by a procedure will not require parameters: excluding A, D and E from

```
A
587c: sethi  %hi(0xa800), %o7
5880: ld  [ %o7 + 0xc8 ], %o7
5884: sll  %o5, 2, %o5
5888: st  %o0, [ %o7 + %o5 ]
588c: sethi  %hi(0xa000), %o0
5890: ldsb  [ %o0 + 0x2ec ], %o0
5894: tst  %o0
```

```
B
5b70: sethi  %hi(0xa800), %o3
5b74: ld  [ %o3 + 0xc8 ], %o3
5b78: sll  %o1, 2, %o1
5b7c: st  %o0, [ %o3 + %o1 ]
5b80: sethi  %hi(0xa000), %o4
5b84: ldsb  [ %o4 + 0x2ec ], %o4
5b88: tst  %o4
```

```
C
5bec: sethi  %hi(0xa800), %o3
5bf0: ld  [ %o3 + 0xc8 ], %o3
5bf4: sll  %o1, 2, %o1
5bf8: st  %o0, [ %o3 + %o1 ]
5bfc: sethi  %hi(0xa000), %o4
5c00: ldsb  [ %o4 + 0x2ec ], %o4
5c04: tst  %o4
```

```
D
5ce0: sethi  %hi(0xa800), %o7
5ce4: ld  [ %o7 + 0xc8 ], %o7
5ce8: sll  %o5, 2, %o5
5cec: st  %o0, [ %o7 + %o5 ]
5cf0: sethi  %hi(0xa000), %o2
5cf4: ldsb  [ %o2 + 0x2ec ], %o2
5cf8: tst  %o2
```

```
E
65d4: sethi  %hi(0xa800), %o5
65d8: ld  [ %o5 + 0xc8 ], %o5
65dc: sll  %o5, 2, %o4
65e0: st  %o0, [ %o5 + %o4 ]
65e4: sethi  %hi(0xa000), %o7
65e8: ldsb  [ %o7 + 0x2ec ], %o7
65ec: tst  %o7
```

Figure 3.4 Instances supporting several procedure mappings

compaction may or may not produce a space savings. These combinations must be examined and evaluated before making a compaction decision.

Finding the best space saving requires looking beyond exact matches. Through clever grouping of instances and use of procedure parameters, we can outperform the "exact match only" FMW algorithm. Our task is to find these groupings without an exhaustive search. In the next chapter we will present and evaluate different heuristic solutions, but first there are some preliminary details to be covered. The rest of this chapter covers the disassembly and procedure search phases of our compaction scheme.

3.2 Preliminaries to Procedure Searching

Compaction through procedural abstraction changes the order of machine instructions within the binary file, but does not alter the result of computation. Marks [21] and Fraser et al. [8] compact the assembly language representation of the program. Many compilers

already translate source code into this representation (e.g. `gcc` and `lcc`), passing it on to an assembler as the last phase of processing. Therefore, compaction could be added as part of a compiler.

In the previous two approaches, compaction can only be performed if the source code is available (i.e. "no source = no compaction.") A disassembler avoids this restriction. As it not only makes available the assembly code from the binary and processes all of the code linked together as a multi-module system, disassembled instructions could correspond exactly to the code found in the binary. What we see (the binary) should be what we get (the disassembled code).

Disassembly has one drawback: separation of code from data is not always possible. The best we can do is find specific sections of the binary are guaranteed to be code. Symbol table information, data from binary interface headers, and knowledge of compiler conventions provide the precise locations of some instructions. However, it is expensive to look for code beyond the information from these sources. Even with techniques such as abstract execution of code, it is still impossible to guarantee that all instructions have been found [14]. Thankfully, the output from the disassembler using only symbol table information is still quite large. Rather than worry about increasing the coverage of disassembled instructions, a economical solution in this situation is to compact what is guaranteed to be code, and leave the rest unmodified.

3.2.1 Disassembly

The disassembler from GNU's source-level debugger `gdb` was used after several simple modifications [22][24]. `gdb`'s code is very portable, supporting many different hardware platforms, and each supported platform has a corresponding machine description file. Whereas Sun 4 SPARC's description consists of 1300 non-comment, non-blank lines of code, machines with more complicated instruction sets (e.g. Motorola 68030, Intel 80486) have larger descriptions.

SPARC's fixed instruction alignment on 4-byte boundaries makes possible a one pass disassembler, and because symbol tables are not absolutely necessary, stripped SPARC

binaries may be compacted. There may still be occasions during disassembly where function entry addresses, data locations and other information from the symbol table would make it easier to distinguish between code and data. On CISC machines with varying instruction sizes, the symbol table is a necessity. There would be little progress without the table as a given section of the binary may be disassembled several ways depending on which address is the start of an instruction. Entry points and function addresses from the table can resolve this type of ambiguity.

Only SPARC binaries were compacted in this project. We assumed that the symbol table information was not available. However, if the binary is not stripped, then function addresses in the symbol table must be updated to reflect the positions of the functions in the smaller binary.

3.2.2 Separating Code from Data

The binary code was mapped into memory, and every instruction separately disassembled by a call to `print_insn_sparc()`. Arguments for the procedure include the address of the instruction to be examined. The results returned are the readable text of the instruction and the address of any branch destination. *Illegal* instructions are those which the disassembler does not recognize and are automatically marked off as data. After the binary is disassembled, all basic block information is examined, and sections that do not fall within a basic block are also marked as data.

Some data sections may contain jump tables. Since compaction will change the position and location of code, the table entries may need updating to reflect the addresses of the moved jump targets. Both Sun's C compiler `cc` and GNU's C compiler `gcc` place the table immediately following the jump instruction. Finding the length of the table helps separate code from data. Instructions indicated by `cc` jump table entries precede the code that performs table lookups. For `gcc`, the instructions are located immediately following the last entry in the jump table. However, compilers for other languages may position instructions and tables according to different conventions, so the problem of finding jump table locations may prove difficult. One solution using *program slicing* has been described

in the literature, and does not depend on knowledge of specific compiler implementations. Program slicing finds the subset of instructions which determine the value of the indirect branch register. From this subset, the location of the jump table is usually deducible [18][28]. However, we make the simplifying assumption that the binaries given to the compactor were produced by either `cc` or `gcc`.

Delay slots on RISC architectures add a complication. Some compilers move into this slot the instruction immediately preceding a branch or jump. Therefore a fragment containing a branch or call must keep the branch instruction and its delay slot in the same procedure. However, there are cases where a delay slot instruction is also a branch destination, which usually occurs when two code blocks are fused together by an optimizer (a space saving optimization). If the slot belongs to a branch instruction, and if that branch is the exit from a procedure, then a simple solution to the problem is to make another copy of the delay slot instruction, keeping one copy in the procedure and the other within the original program [5].

3.2.3 Difficult Code

Several classes of code are difficult to compact because it may not be possible to guarantee the correctness of their compacted versions.

Self-modifying code treats some address contents as both code and data. For example, a *direct dynamic linker* patches in the address of the called subroutine at run-time by modifying the program [16]. Self-modifying schemes usually assume a fixed distance between the instruction to be changed and the instructions performing the change. Moving either instruction will overwrite some other address. Not only is it difficult to recognize self-modifying code, but it will also be hard to ensure that it will work after compaction.

Arithmetic on branch destinations, usually performed on indirect branches, requires some data flow analysis for detection. Jump table lookups are easy to recognize, but other arithmetic on registers might not be used until some instructions later. Determining the possible destinations using some data flow analysis and arithmetic is necessary since there is the potential of some destinations located in sections considered data which should now

be viewed as code. So far, no code that has been disassembled has contained indirect branches of this sort, and we assume that this will not change.

Some programs *calculate an address based on the checksum of the instructions*, usually for security purposes — either for copy protection, or to check against tampering of a file during transmission across a network. By definition, these files will not work after the slightest program change, and this rules out their compaction.

3.3 Procedure Searching

We have used McCreight's suffix tree algorithm [19] to find procedure possibilities. This algorithm is also the basis for the FMW procedure search, but our point of departure is in the treatment of the program. Where FMW is looking for instances of exact textual matches in the assembly code, we are only looking for instances where *operations* match and allow *operands* to differ between instances.

Some properties of the suffix tree data structure have already been investigated in Section 2.5. Our implementation of the suffix tree algorithm accepts a string of operations as input, and constructs a tree with all possible substrings, and their positions, within the larger string. All substrings and positions are then examined (see next section), and passed to the algorithms that attempt to find the groupings of instances that will produce the best space savings given the substring instances (next chapter).

For example, suppose we are given the following contrived example:

```
add   %sp, 0x44, %o1
sll   %o0, 2, %o2
add   %o2, 4, %o2
add   %o1, %o2, %o2
sethi %hi(0x4000), %o3
save  %sp, -152, %sp
call  0x21bc
sll   %o1, 4, %o1
add   %o3, 5, %o3
add   %o1, %o2, %o4
sethi %hi(0x8000), %o2
save  %sp, -104, %sp
call  0x22c0
return
```

which yields a string of operations:

op	add	sll	add	add	sethi	save	call	sll	add	add	sethi	save	call	return
pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13

The suffix tree for this little program can be found in Figure 3.5, and is printed as a prefix traversal of the nodes. Each line of the figure corresponds to a node, and the string of instructions listed on each line is the label on the edge leading into the node. Numbers on the left hand side indicate the level (depth) of the node ("0" indicates "root"), and numbers on the right mark the terminal nodes for each suffix starting at that position in the original string. All other nodes are internal, and are also referred to as "non-terminals".

Our list of substrings uses this last set of numbers (Table 3.1). Concatenating the labels encountered during a downward traversal from the root to any non-terminal forms the substring; all terminals descending from the non-terminal provide the starting positions of the substring. For instance, the substring "`sethi save call`" starts in two places: string positions 4 and 10. Notice that substrings in one row may be nested within others from a different row. Substring positions may be easily converted into real instruction locations given the fixed sized of SPARC instructions. Later analyses will examine these substrings, and the data gathered during disassembly will be needed.

Table 3.1 Substring table from suffix tree

substring	starting positions
`add sethi save call`	**3, 9**
`add add sethi save call`	**2, 8**
`sll add add sethi save call`	**1, 7**
`sethi save call`	**4, 10**
`save call`	**5, 11**
`call`	**6, 12**
`add`	**0, 2, 3, 8, 9**

The McCreight algorithm has some important properties. It constructs the suffix tree using linear space, and executes in linear time. Each string position, which implies each

```
0:{empty}
1: add
2:      sll add add sethi save call sll add add sethi save call return (0)
2:      sethi save call
3:          sll add add sethi save call return (3)
3:          return (9)
2:      add sethi save call
3:          sll add add sethi save call return (2)
3:          return (8)
1: return (13)
1: call
2:      sll add add sethi save call return (6)
2:      return (12)
1: save call
2:      sll add add sethi save call return (5)
2:      return (11)
1: sethi save call
2:      sll add add sethi save call return (4)
2:      return (10)
1: sll add add sethi save call
2:      sll add add sethi save call return (1)
2:      return (7)
```

Figure 3.5 Suffix tree for example — prefix traversal

suffix (from that string position to the end of the string), adds one terminal node and at most one non-terminal node. Similarly, each suffix adds at most two new edges. Therefore, each step in the algorithm requires constant time. The algorithm also reduces the contribution of scanning edge labels to linear time. Each scan tries to determine where to split an edge and add a non-terminal. Unfortunately, linear time only applies once the edges suitable for scanning have been found. The effort to find the edge when the alphabet has n (in Figure 3.5, $n = 6$) characters yields a potential $O(n^2)$ algorithm. A hash table implementation could bring this down to $O(n \lg n)$. Therefore the algorithm is not linear, but it does find all substrings, including those nested within others.

3.4 Quick Comparison of Different Methods

There are some differences in the three methods (Marks [21], FMW [8], and ours) when examining the strings generated by the procedure search. This examination is necessary before passing the strings on to the instance grouping heuristics. Table 3.2 summarizes the differences between the three methods, with each category described below.

1 — Different procedure invocation methods: Using an unconditional branch to transfer control to a procedure is less expensive, in terms of both time and space usage, than using a subroutine call. However, the former method only works if the last instruction of the candidate instance is a branch to the same absolute address in every instance. If control flow proceeds directly to the instruction physically located after the candidate's last, then the subroutine mechanism must be used. Subroutines will behave correctly only if code for the candidate leaves a balanced stack (see the next category). Therefore, the instructions in the procedure determine how it is invoked within the main program.

Invocation methods may be classified by their effect on a compacted program's space and speed. "Low space — high time" categorizes Marks' tailored interpretation, where subroutines are invoked through one instruction, but only with the support of an interpreter. FMW purposely sets out to use a "low space — low time" method, rejecting the use of parameters, thereby requiring only one instruction for the procedure invocation. "High space — low time" describes our approach, which needs extra instructions to copy parameters into registers when invoking a procedure.

2 — Check for stack balancing: With most machine architectures, a subroutine call places registers and return addresses onto the user stack. If the code for a candidate does not push the same number of items that it pops, then the stack is returned to the caller in a state different than at call time, and the subroutine code is called "unbalanced". Aggressive balance checking schemes will also try to prune candidates (shortening by removing offending instructions from either end) to remove instructions until the stack is guaranteed to be balanced. Further complications involving stacks include expectations of data at a certain depth from the top of the stack, but pushing on a compacted routine's return address changes that depth. The compacted procedure instructions that use the stack must be changed to reflect the new depth.

3 — Maximum number of parameters per procedure: If two instances differ in one or more operand positions, and if both of these instances are to be covered by the same procedure, then parameters will be needed by the procedure. Before the procedure call,

these operand values are copied into parameter passing variables (usually registers) and used by the subroutine. Procedure results may be returned via these parameters, so their values must be copied back into their proper locations before processing continues with the next instruction following the call. We expect to see more "nearly identical" matches than "identical", so parameters appear unavoidable. In the past, some researchers have rejected parameters out of hand as "too expensive" (FMW), while another allows a single parameter (Marks — tailored interpretation). The actual expense of parameterization has not been examined.

Table 3.2 Possibilities during candidate examination

compaction Method	Marks	FMW	Ours
1. Different procedure invocation methods	no	yes	yes
2. Check for stack balancing	yes	yes	yes
3. Maximum number of parameters per created procedure	1	0	depends on heuristic

We are most interested in item three. If there are too many parameters, then the cost of their use eliminates any space savings. If there are too few, then many compaction candidates will be rejected. We are interested in determining the minimum number of parameters that will lead to space-savings when compacting a set of instruction instances. A parameterizing heuristic could fix the maximum number of parameters, rejecting all those exceeding this maximum but at the risk of finding no space savings. Conversely, another heuristic could allow as many parameters as needed as long as space savings are the result, but this might produce smaller savings than if the number of parameters were fixed to a small number. These possibilities are investigated in the next chapter.

3.5 Summary

We have presented our view of the compaction problem, and have suggested combining parameterization with the grouping of matches into procedures as a way to outperform a scheme prohibiting parameters. Our task is to develop these grouping algorithms such that

parameter costs are minimized and space savings maximized. Some of the preliminary requirements for compacting binaries were reviewed, including a description of those programs which cannot be guaranteed correct after compaction. Finally, we reviewed the substring construction algorithm (suffix tree) that provides the raw material from which to choose procedures. Not all substrings can be used, and we presented a few criteria that must be met. Now it is time to inquire into the benefits of parameters, at the same time examining heuristics which can select procedures such that parameter costs are minimized.

4 Searching for Subroutines

Our compaction method for object code (binaries) has two phases:

- converting the binary into a form in which we can locate repeating code sections; and
- examining this code for suitability as procedures, rejecting or transforming them as needed.

The previous chapter covered the first phase and this chapter examines the second. Given a set of instruction strings that are similar to each other, we will now suggest some heuristics that can convert these substrings into space saving procedures even if the strings are not identical. A heuristic approach avoids exhaustive searches of the partitioning solution space. This chapter continues with the introduction of some notation, followed by a quick overview of the criteria which compaction candidates must satisfy. The procedure parameterization problem is presented, and several relations are introduced that will support our heuristic algorithms. Finally, the heuristic algorithms themselves are motivated, presented, and their performance examined.

4.1 Definitions

In the past chapters, we have used such terms as "substrings", "instructions" and "operands" in an informal manner. To reduce ambiguity, we introduce these definitions.

- If x is an instruction, then define $|x|$ to mean the number of operands contained in the instruction. If x is a string, then define $|x|$ to be the number of instructions in the string.
- Define I to be an **instruction** consisting of an **operation** op and a (possibly empty) list of $|I|$ **operands**, $\omega_1, \omega_2, \ldots, \omega_n$. For instance, the instruction "`sll %o0, 2, %o2`" has op = sll, $|I| = 3$, ω_1 = %o0, $\omega_2 = 2$, and ω_3 = %o2. The "no operation" instruction "`nop`" is represented by op = nop and $|I| = 0$ (i.e. no operands). If there is no ambiguity, the operation for instruction I' is op', and the instruction's i^{th} operand is ω_i'.

- Define **program** P to be a sequence of m instructions $I_1, I_2, \ldots, I_m$. A **substring** starting at position i and with length L is denoted by $I_{i,L}$.

- We write $I' \cong I''$ to denote an **instruction match** between instructions I' and I'', where two instructions may have different operand. This is the same as writing:

$$|I'| = |I''|$$
$$\text{and } op' = op''$$

- An **exact match** occurs between instruction $|I'|$ and $|I''|$, and is written as $I' = I''$, if:

$$|I'| = |I''|$$
$$\text{and } op' = op''$$
$$\text{and } \omega_j' = \omega_j'' \quad 1 \le j \le |I'|$$

- Two substrings, $I_{i,L}$ and $I_{j,L}$, are said to be **instances of a (procedure) candidate** if they do not overlap and if each operation is the same between instances; however, operand values may differ between instances. The number of instances in a candidate C is equal to $|C|$. Using our notation, two substrings are instances if:

$$\{i, i+1, \ldots, i+L-1\} \cap \{j, j+1, \ldots, j+L-1\} = \emptyset \qquad \text{and}$$
$$I_{i+k} \cong I_{j+k} \qquad 1 \le k \le L$$

- Define a **partition component** κ ("component" for short) to be a subset of instance indices from a procedure candidate (i.e. $\kappa \subseteq \{1, \ldots, |C|\}$). Each component is associated with a procedure body, and each instance in the component may be replaced in the original program by a branch or call to the associated procedure. There will be cases where a candidate instance will be excluded from any procedure, and left as found in the original program, because of the cost of the instance's inclusion. These excluded instances make up κ_ε.

- Define a **partition** Π of the procedure candidate C to be a group of disjoint components. Each component corresponds to one procedure body, and component members are instances that are replaced by branches or calls to the procedures. If the partitioning does not cover all instances, then those excluded are left as found in the original object file.

Using the notation for a partition, we define a partitioning Π containing n partitions as $\Pi = \{\kappa_\varepsilon, \kappa_1, \kappa_2, \ldots, \kappa_n\}$, with the partitions satisfying the following conditions:

$$i \neq j \Rightarrow \kappa_i \cap \kappa_j = \emptyset \quad \text{and}$$
$$\kappa_\varepsilon \cap \kappa_i = \emptyset \quad 1 \leq i \leq n \quad \text{and}$$
$$\kappa_\varepsilon \cup \left(\bigcup_{i=1}^{n} \kappa_i \right) = \{1, \ldots, |C|\}$$

4.2 Visualizing Operands

Thinking about the parameterization problem is difficult when the number of candidate instances is large. An easier method for visualizing the relationship of operand values between instances is by converting the candidate instances into an array of operand values. Each row of the array will correspond to a candidate instance, and a column contains the values of operands at one specific position across all instances. Operand text is replaced by an ordinal value (i.e. "v1" for "value 1"). Our implementation leaves out the instruction operations from the array, as these are guaranteed (as per our restriction that instructions must be candidate matches) to be the same between instances. However, as an aid to connecting array rows with candidates instances, the instruction operations will be shown in our figures in shaded columns. After substituting ordinal values for operand text from Figure 3.2 of Section 3.1, we have an array representation in (Figure 4.1). Row and column indices are provided to improve readability. Scanning down each of the eight operand

	operands											
inst		1	2	3		4	5		6	7		8
i1	ld	v1	v2	v1	mov	v3	v4	mov	v5	v6	b	v7
i2	ld	v1	v2	v1	mov	v8	v4	mov	v5	v6	b	v7
i3	ld	v1	v2	v1	mov	v9	v4	mov	v5	v6	b	v7
i4	ld	v1	v2	v1	mov	v8	v4	mov	v5	v6	b	v7
i5	ld	v1	v2	v1	mov	v9	v4	mov	v5	v6	b	v7
i6	ld	v1	v2	v1	mov	v10	v4	mov	v5	v6	b	v7

Figure 4.1 One parameter example — Ω-array form

columns, we see that one parameter is needed for the operands in column 4, and this corresponds to the first operand of the "mov" instruction. Since all other operands have the same values across all instances, they do not require parameterization.

We call this two dimensional array the Ω*-array*. Instances of the same candidate, made up of n substrings, are transformed into an array containing only the ordinal values for operand values. Each row Ω_i contains the equivalent of the ω values for the instruction operands in i^{th} substring starting in position p of the program (call it $I_{p,L}$).

Operand values are positioned in the row in the same order in which they appear within instructions and between instructions. That is, if $\Omega_{i,j} = \omega'$, $\Omega_{i,k} = \omega''$, and ω' appears before ω'' in the $I_{p,L}$ (either because ω' is in an earlier instruction, or appears earlier in the same instruction) then $j < k$ must hold. Since we know how many operands may be found in each of the substring's instructions (i.e. $|I_p|, |I_{p+1}|, \ldots, |I_{p+L-1}|$), any parameter choice based on the analysis of the Ω-array can be related back to the instance's instructions and their associated operands.

As stated earlier, we require that instruction operations to be identical with each other between substrings. A more flexible scheme relaxes this restriction, allowing differences in operations between instances. Given this relaxation, Geschke's "strongly similar subroutine" optimization [9] could be implemented. Parameterizing for operation and operand differences, with the parameter value is used to select a compacted procedure statement from the set of possibilities, could produce procedures that cover very long strings. However, this would add another dimension to the parameterization problem, and even more space would be required both at the procedure call site and in the procedure body itself for parameters. Future work could examine the space savings when allowing for instruction operation and operand mismatches. However, we will concentrate exclusively on operand mismatches.

4.3 Ω-array Partitions

4.3.1 Motivation

So far we have visually identified the positions of parameters. However, suppose we must parameterize something moderately large, as in Figure 4.2. What partitioning produces the

	operands															
inst		1	2	3		4	5	6		7	8	9		10	11	12
i1	ld	v1	v2	v3	mov	v4	v5	v6	ld	v7	v6	v3	mov	v4	v8	v6
i2	ld	v3	v9	v10	mov	v9	v11	v3	ld	v3	v9	v12	mov	v9	v13	v3
i3	ld	v3	v9	v10	mov	v9	v11	v3	ld	v3	v9	v12	mov	v9	v13	v3
i4	ld	v3	v4	v14	mov	v4	v5	v3	ld	v3	v9	v15	mov	v4	v16	v3
i5	ld	v3	v4	v15	mov	v9	v17	v3	ld	v3	v9	v18	mov	v9	v19	v3
i6	ld	v3	v4	v20	mov	v9	v21	v3	ld	v3	v9	v22	mov	v9	v23	v3
i7	ld	v3	v4	v8	mov	v9	v24	v3	ld	v3	v9	v25	mov	v9	v26	v27

Figure 4.2 Complicated partitioning — Ω-array

best compaction? If instances `i1` and `i4` are placed in κ_ε (i.e. left uncompressed), then the other five instances could be covered by one procedure using 5 parameters. If they are not excluded, then the procedure would need 12 parameters, and space savings would be very unlikely. Before finding the minimum number of partitions, we must find out how many operand values are identical in each column. This is a scan that proceeds up and down (i.e. *vertically*) each of the columns of the Ω-array, *partitioning* the set of instances into components with identical operand values.

Another (somewhat contrived) example is shown in Figure 4.3. Scanning down each column reveals that there are no operand matches between instances. However, only one operand value appears in each instance. A procedure covering all instances need only receive one parameter, and all operand positions are *tied* together by the same value. We have reduced the number of parameters from 7 to 1. As it is possible to reduce the cost of parameters by tieing together column positions sharing the same value within an instance, we scan each row of the Ω-array from left to right (i.e. *horizontally*), *partitioning* the set of operand positions into components with identical operand values.

instance	operands											
		1	2	3		4			5	6		7
i1	ld	v1	v1	v1	tst	v1	nop	sll	v1	v1	b	v1
i2	ld	v2	v2	v2	tst	v2	nop	sll	v2	v2	b	v2
i3	ld	v3	v3	v3	tst	v3	nop	sll	v3	v3	b	v3
i4	ld	v4	v4	v4	tst	v4	nop	sll	v4	v4	b	v4
i5	ld	v5	v5	v5	tst	v5	nop	sll	v5	v5	b	v5
i6	ld	v6	v6	v6	tst	v6	nop	sll	v6	v6	b	v6

Figure 4.3 Tieing example — Ω-array

An exhaustive search of the partitioning solution space will always find the parameterization producing the best space savings, but this is not efficient. A heuristic algorithm still seems attractive, and we could use the components of vertical and horizontal partitions to guide our search. Vertical partition components appearing many times in array columns will suggest candidate groupings producing procedures with fewer parameters than those components which appear only a few times. Similarly, horizontal partition components containing a large number of operand positions and many instance appearances will reduce parameterization by eliminating the duplication of parameter values.

4.3.2 Constructing Vertical Partitions

Vertical partitions span all instances of a candidate, and the members of each partition component are row indices from the Ω-array. These members have the same operand value in a specific column (operand position) and therefore are grouped together. Values such as a vertical component's size (number of instances), and the number of times in which the component appears in a partitioning for a column (number of columns) are used by the heuristics. As these two numbers grow larger, the corresponding component becomes a desirable guide for grouping instances into procedures.

Our main interest is in any vertical partition's components. Each component is uniquely determined by the Ω-array rows that it covers (instances), and this "name" is

stored in another array called Γ. This name refers to a set of array *row* indices. Each entry of Γ has a corresponding entry in the array G. Elements of G are a a set of Ω-array *columns* (operand positions) in which the component appears as part of the vertical partitioning. Therefore, Γ and G are arrays of sets, with each set of rows (Γ_z) associated with a set of columns for which the instances have the same ω-values (G_z).

An algorithm to construct the vertical partitions and store the components, given an Ω-array, is shown in Figure 4.4, and the results can be found in Table 4.1. After initializing

```
procedure buildVerticalPartitions(Ω, VAR Γ; VAR G)

1   Initialize all elements of arrays Γz, Gz to ∅
2   for j = 1 to numOperands /* number of operands in instance */
3       T = {1, 2, ..., |C|}
4       while T ≠ ∅ do
5           select any t ∈ T
6           U := {t}
7           for each i such that Ωi,j = Ωt,j
8               U := U ∪ i
9           find z such that Γz = U
10          if not found, construct such Γz at next unused value of z
11          Gz := Gz ∪ j
12          T := T-t
13  return Γ, G
```

Figure 4.4 Algorithm 2 — Finding Vertical Partition Components

all of the elements (line 1) in Γ and G, the algorithm examines each operand column of the Ω-array in turn (line 2). All instances with the same ω-value in this column are grouped together, one unique value at a time (lines 3 — 12), forming the vertical partition of column j. Each component (U) is used to find a z such that $\Gamma_z = U$ (line 9). If this is the first time that a component equal to U has appeared for this Ω-array, then U is added to the Γ-array in the next open position (line 10). The column index j is added to G_z, indicating that the component described by Γ_z appears in the vertical partitioning for this column. All

components for the column are found before proceeding to the next column (lines 3, 5, 12). After processing all columns, the two arrays of sets are returned to the caller.

Running this algorithm on the Ω-array of Figure 4.3 produces the components and component appearances shown in Table 4.1. By definition, all Γ_z values for an operand

Table 4.1 Vertical components for candidate in Figure 4.3

z	Γ_z (components)	G_z (component appearances)	
		before closure	after closure
1	i1	1,2,3,6,7,8,9,11,12	1,2,3,6,7,8,9,11,12,4,10,5
2	i2,i3,i4,i5,i6,i7	1,6,7,8	1,6,7,8
3	i2,i3	2,3,5,9,11	2,3,5,9,11,7,8,4,10,1,6,12
4	i4,i5,i6,i7	2	2,1,6,7,8
5	i4	3,9,11	3,9,11,1,2,4,5,6,7,8,10,12
6	i5	3,5,9,11	3,5,9,11,1,2,4,6,7,8,10,12
7	i6	3,5,9,11	3,5,9,11,1,2,4,6,7,8,10,12
8	i7	3,5,9,11,12	3,5,9,11,1,2,4,6,7,8,10,12
9	i1,i4	4,5,10	4,5,10
10	i2,i3,i5,i6,i7	4,10	4,10,1,6,7,8
11	i2,i3,i4,i5,i6	12	12,1,6,7,8

column are disjoint. Γ array entries may be subsets of other components in the Γ-array. For instance, Γ_2 is `{i2, i3, i4, i5, i6, i7}`, and this component can be found in the partitioning of columns 1, 6, 7 and 8. Therefore, we expect that instances from any Γ-array entry which is a subset of Γ_4 will have the same value in these columns. After performing a closure operation, the G-array entries corresponding to `{i2, i3}`, `{i4, i5, i6, i7}`, `{i4}`, `{i5}`, `{i6}`, `{i7}`, `{i2, i3, i5, i6, i7}`, and `{i2, i3, i4, i5, i6}` will contain columns 1, 6, 7 and 8 after closure. The closure operation can be applied to the G-array elements either during or after partition component construction.

4.3.3 Constructing Horizontal Partitions

Vertical partitions suggest candidate groupings through components, as each component is a possible grouping of instances to a procedure. Since the instances of the component Γ_z

have the same operand values in the columns contained in G_z, the instances may be covered with a procedure that does not need parameters for the columns in G_z. The remaining columns ($\{1, \ldots, numColumns\} - G_z$) *do* require parameters, and the values to be passed in could be tied together, and this results in a requirement for fewer parameters.

For each instance (Ω -array row), operand positions (Ω -array columns) with the same value are partitioned together. Components with a single member are operand values that appear only once within the instance (singletons), and are ignored when reporting ties. Table 4.2 contains the horizontal partitionings for the Ω -array of Figure 4.2. Their construction is similar to the process for vertical partitions.

Table 4.2 Horizontal partitions for example in Figure 4.2

instance	horizontal partition
i1	{1} {2} {3, 9} {4, 10} {5} {6,8,12} {7}{11}
i2	{1, 6, 7, 12} {2, 4, 8, 10} {3} {5} {9} {11}
i3	{1, 6, 7, 12} {2, 4, 8, 10} {3} {5} {9} {11}
i4	{1, 6, 7, 12} {2, 4, 8, 10} {3} {5} {9} {11}
i5	{1, 6, 7, 12} {2, 8} {4, 10} {3} {5} {9} {11}
i6	{1, 6, 7, 12} {2, 8} {4, 10} {3} {5} {9} {11}
i7	{1, 6, 7} {2, 8} {4, 10} {3} {5} {9} {10} {11} {12}

Given the horizontal partitions for each instance, we wish to find the ties that are valid for all instances of a vertical partition component. In our notation, capital letters are horizontal partitions, and lower case letters are their components. Given two partitions satisfying our definition in Section 4.1:

$$Q = \{q_1, q_2, \ldots, q_n\} \qquad R = \{r_1, r_2, \ldots, r_m\}$$

where each of the components q and r are sets of column indices, a partition that is valid for both of them is called T, and defined as:

$$T = \{q \cap r | q \in Q, r \in R\}$$

We write the operation implementing this relation as:

$$T = Q \cap R$$

As an example, we take one possible grouping of instances from Figure 4.2 — {i3, i5, i7}. The following horizontal sets are taken from Table 4.2. We wish to know what ties will be valid for a procedure covering all instances of this partition. Given:

```
i3:   {1, 6, 7, 12} {2, 4,  8, 10} {3} {5} {9} {11}
i5:   {1, 6, 7, 12} {2, 8} {4, 10} {3} {5} {9} {11}
i7:   {1, 6, 7}{12} {2, 8} {4, 10} {3} {5} {9} {11}
```

then the computation of T (where T happens to coincide with the value for i7) produces:

```
T:  {1, 6, 7} {12} {2, 8} {4, 10} {3} {5} {9} {11}
```

Using Venn diagrams, one can see the "∩" operation applied to the different horizontal partition components in the computation of $i3 \cap i7$ (Figure 4.5). Each of the intersections becomes a component of the tied partition. All columns in an intersection by themselves are operand positions that cannot safely share a parameter with any another position. For instance, i3 has the same operand values in columns 1, 6, 7, and 12; i7 does not include column 12 as part of the set. Therefore any tieing applied to both i3 and i7 must give column 12 its own parameter, otherwise the tie will copy in an incorrect value into operand 12 for instance i7.

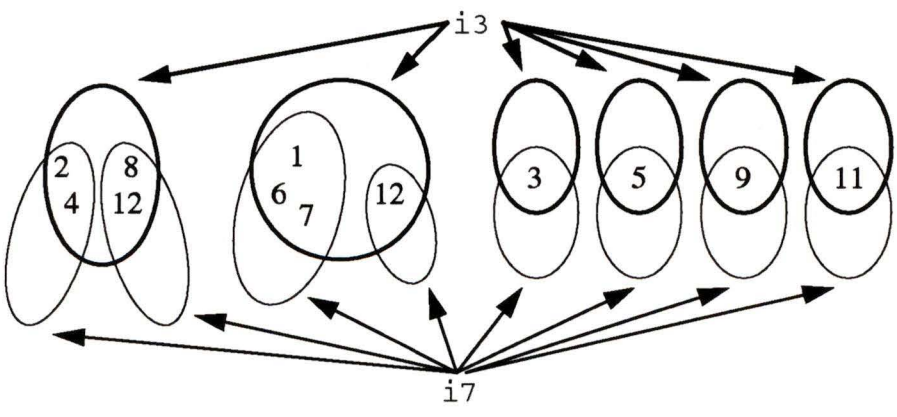

Figure 4.5 Finding a valid tieing partition

Horizontal partitions are needed to find parameter ties, but parameter ties provide no clue as to which candidate groupings save the most space: parameter tieing makes a good

situation better. However, even a good tie partition will not reduce the number of parameters if the operand columns it covers do not need parameters.

```
/* The horizontal partitions H for the candidate and the
 * candidate partitioning component κ, also known as a
 * candidate grouping, are passed in.
 * Valid ties for all instances are returned in T.
 */
procedure findValidTies( H, κ, VAR T)

select any i ∈ κ;
κ' := κ - i;
T = H_i
for each i such that i ∈ κ' do
    T = T ∩ H
return T
```

Figure 4.6 Algorithm 3 — Find tieing valid for partition instances

With parameter ties valid for all component instances, we can calculate the cost of any mapping of candidate instances to procedures. Finding the components leading to the best space savings is the difficult part, and we look at this next.

4.4 Heuristics

The main motivation for using vertical and horizontal partitions is to provide a starting point for exploration of the procedure/parameter solution space. A brute force approach which examines each possible partitioning is expensive because it is combinatorially explosive. Suppose that the candidate has been converted into an Ω-array. If the number of rows in the array is m, and the number of partitions over the candidate is n and allowed to vary over $1 \le n \le m$), then the number of unordered partitionings on m is:

$$\sum_{n=1}^{m} S(m, n) \qquad \textbf{(EQ 4.1)}$$

where $S(m, n)$ is a Stirling number of the second kind [26]:

$$S(m, n) = \frac{\sum_{k=0}^{n-1} (-1)^k C(n, k)\,(n-k)^m}{n!} \qquad \text{where} \qquad C(n, k) = \frac{n!}{k!\,(n-k)!}$$

Even reasonably sized candidates can produce large numbers of combinations, and our heuristics regularly meet up with 10 to 15 instances in a candidate (see Table 4.3). Casting the question "how should the candidate instances be partitioned" into a known optimization problem also appears to be difficult.

Table 4.3 Possible number of partitionings

number of instances (m)	number of unordered partitionings (Eq. 1)	number of instances (m)	number of unordered partitionings (Eq. 1)
1	1	8	4140
2	2	9	21147
3	4	10	115,875
4	15	15	1,382,958,545
5	52	20	~5.174 x 10^{13}
6	203	25	~4.638 x 10^{18}
7	877	30	~8.467 x 10^{23}

In this section we do not aim for optimal compaction. Instead we attempt to find space saving opportunities that other compaction algorithms (Marks, FMW) pass over. For this more modest goal, we may appeal to heuristics. Our basic rule of thumb is that a grouping of instances to a procedure should (a) cover as many instances as possible while (b) resulting in as few parameters as possible. Requirement (a) maximizes the ratio of released space to procedure body space, and (b) improves the space savings at the original instance positions.

4.4.1 Calculating Space Savings

Each heuristic is evaluated based on the space savings potential of its generated groupings. Computing space savings on architectures with variably sized instructions (e.g. Intel 80x86, Motorola 680x0) must take into account the size of the operands. However, SPARC instructions have a fixed size, which simplifies the space savings calculation. Parameter costs are also slightly reduced on the SPARC machines because all branch and call instructions are paired together with an extra instruction called the *delay slot*. This slot forces a compile-time decision as to which instruction is fed into the processor pipeline

when there is a change in control flow during program execution. We use this slot to copy a value into a parameter variable during a procedure invocation. If there are no parameters, then the first instruction of the procedure may be placed in the slot.

There are five parts to the calculation:

S_u — *savings upper bound*: No scheme can do better than that which removes all instances from the code. All of the following costs are subtracted from this upper bound.

C_1 — *parameters copied at procedure entry*: A procedure call (closed subroutine) or branch (open routine) may use the delay slot to copy a parameter, so there is no cost for the first parameter. After the first, all other parameters cost one instruction.

C_2 — *parameters copied at procedure exit*: One instruction per parameter is needed. These copy the results from parameter registers to the original variable locations.

C_3 — *space needed for procedure body*: Procedures are almost the same size as a candidate instance. Two extra instructions are needed if the candidate is in closed form (subroutine) for the "return" instruction and its delay slot. No extra instructions are needed if the procedure is in open form.

C_4 — *branch or call instruction*: As mentioned in the description of C_1, one instruction is needed for the transfer of control to the procedure. A second is for the delay slot.

Each component of the candidate partition Π (except κ_ε) requires one procedure, and therefore contributes a list of costs. Different parameterizations for different partitions mean different values for C_1, C_2 and C_3. As instances of a candidate use the same procedure invocation mechanism, C_4 remains fixed. In the equation, $C_{i,j}$ is the cost C_i associated with parameterization of κ_j. For n components, the space saving is:

$$S_{\text{total}} = S_u - \sum_{j=1}^{n} (C_{1,j} + C_{2,j} + C_{3,j} + C_4) \qquad \textbf{(EQ 4.2)}$$

Equation 4.2 is the basis for evaluation of the partitioning heuristics presented in the rest of this chapter. After finding all the space savings for a binary given the advice of a specific

heuristic, the total space savings for the whole file are divided by the original binary size, multiplied by 100, and reported as the "% space savings". There are four categories based on the number of lines that are disassembled as code: (a) <2K instructions; (b) 2K — 5K instructions; (c) 5K — 10K instructions; and (d) >10K instructions. Since we are investigating the performance of the heuristics, it is important to focus on how well they work, as distinct from how well the disassembler separates code from data. Improving disassembler code coverage is important, but as mentioned earlier in the thesis, if we are only given the binary file and its symbol table, 100% separation of code from data is expensive.

For simplicity of analysis, there is no nesting or overlapping of procedures. Our results were compared against the FMW results (in Section 4.4.6) if the number of instructions saved was greater than 50 and the space savings at least 1%. It *does* appear that the order in which candidates are evaluated by the heuristics affects the final result. We tried three different orderings: (a) by maximum potential space savings; (b) by the number of instances in the candidate; and (c) by length of an instance in the candidate (number of instructions). The best space savings of the three orderings are used for our reports. Out of 200 UNIX utilities, 174 produced a space savings greater than 1 instruction, and 67 produced a space savings greater than 50 instructions.

4.4.2 FMW Compaction (Base Line)

A big surprise was the poor performance of the FMW algorithm. Although the 1984 paper [8] reported space savings ranging from 0 to 39% with an average of 7%, our implementation of FWM on the SPARC produced far poorer results. This could be due to several factors: the SPARC register set's richness makes identical matches less frequent. The paper applied compaction to assembly code where, by definition, all the code and data is separated. FMW may be affected by presence of string constants within the code. This may explain the order of magnitude difference in the paper's results and our own implementation's results. However, this does not rule out comparison with our heuristics

— the substrings FMW finds will always be found by our algorithm. Results for the FMW scheme are in Table 4.4.

Table 4.4 FMW compaction — % savings

instructions	average	min	max
0 — 2K	1.5	1.1	1.7
2K — 5K	2.3	1.0	5.4
5K — 10K	2.0	1.0	3.9
10K +	2.3	1.1	6.1

4.4.3 Heuristic 1

This scheme attempts to find one procedure by maximizing the number of instances in a partition component $|\Gamma_z|$ *and* maximizing the number of operand columns not needing a parameter. Larger operand column sets ($|G_z|$) indicate that fewer parameters are needed for a partitioning based on Γ_z. All instances not included in the procedure are left as found in the original binary. A pseudocode algorithm appears in Figure 4.6.

```
procedure heuristic1( Γ, G, VAR κ)

max := 0;
maxSet := -1;
∀z do
    a := |Γz| ;        /* number of instances */
    b := |Gz| ;        /* number of operand positions */
    if ( a * b) > max then
        κ := Γz ;
        max := a * b;
return κ;
```

Figure 4.7 Algorithm 4 — partitioning heuristic 1

The algorithm is greedy as it finds the partition that covers the largest area of the Ω-array. If the disassembled binary contains substrings that heavily favor some operand values over others, then the space savings suggested by the heuristic are high. Results are poor if the number of instances making up the returned component is large while the number of operands making up each instance is small. That is, we find better space savings

if we have candidates with long instances (many instructions), but fewer of them in the component. One major drawback of heuristic 1 is that after making a good compaction choice, all excluded instances are completely ignored (i.e. placed in κ_ε) and are not re-examined for further compaction opportunities. Experimental results are in Table 4.5.

Table 4.5 Heuristic 1 compaction — % savings

instructions	average	min	max
0 — 2K	2.0	1.1	3.2
2K — 5K	2.2	1.1	4.8
5K — 10K	2.2	1.0	5.3
10K +	2.6	1.0	8.5

4.4.4 Heuristic 2

The drawbacks of the previous heuristic suggest another method. Partitionings of the candidates should contain more than two components (in heuristic 1, we have only $\{\kappa_\varepsilon, \kappa_1\}$). As we are interested in all those components (groups of instances) leading to procedures with potential for space savings, only those containing three or more instances are examined. All vertical components are sorted by decreasing number of columns ($|G_z|$) — components with the greatest numbers of parameter free columns are examined first. The secondary sort field is the length of the component itself ($|\Gamma_z|$) — the number of instances. Components κ are added to the partitioning Π if they are *instance disjoint* with those already in Π (i.e. an instance may not be replaced by two procedures). After examining the sorted components, all instances excluded from the selected partitions will remain as found in the original executable. Note that the excluded instances can support no other space saving partitions, and this is an improvement from heuristic 1. The algorithm is found in Figure 4.8.

This algorithm is also greedy, but it is allowed to progress further than heuristic 1. It is not significantly better (see Table 4.6). As the size of the program grows, it begins to perform worse than heuristic 1, and this goes against our intuition. After all, heuristic 2 should contain all the space savings of the previous scheme. This is not the case, though,

```
procedure heuristic2( G, Γ, VAR Π)

for each z = 1...|Γ| do
    if |Γ_z| < 3 then
        Γ_z, G_z := ∅ ;          /* discard set */

G' := sort ( G, primary key decreasing |G_i|,
                    secondary key decreasing |Γ_i| );
Γ' := Γ sorted in the same order as G';

Π := ∅ ;                        /* start with empty partitioning    */
for each z = 1...|G'| do        /* look at each set in sorted order */
    s := Γ_z' ;                 /* s is a set of instances (rows)   */
    if (∃t ∈ s and ∃κ ∈ Π) and (t ∩ κ = t) then
        ignore s ;              /* set contains instance already in P */
    else
        Π = Π ∪ s ;             /* add set to partitioning          */

return Π ;                      /* return the partitioning          */
```

Figure 4.8 Algorithm 5 — partitioning heuristic 2

as 2 places the highest priority on components with the fewest parameters, as opposed to heuristic 1 which tries to find the largest possible space savings in one procedure. Even though a heuristic 1 procedure may have several parameters, if it has a sufficiently large number of instances (and each instance is made up of many instructions), then better space savings would result with this rejected partition than with many smaller heuristic 2 procedures that use fewer parameters.

Table 4.6 Heuristic 2 compaction — % savings

instructions	average	min	max
0 — 2K	2.0	1.2	3.0
2K — 5K	2.3	1.0	5.7
5K — 10K	2.3	1.0	4.9
10K +	2.1	1.0	7.3

4.4.5 Heuristic 3

One point of view holds that parameters are too expensive: we are willing to pay at most for one or two. Heuristic 3 limits partitions to a maximum number of parameters. The method for this scheme (algorithm 6) may be found in Figure 4.9. It is difficult to detect an

```
procedure heuristic3( maxParms, G, Γ, VAR Π)

c := 1;
for each z do
    n := numParms( G_z, Γ_z );
    if ( n ≤ maxParms ) then
        N_c := n;
        G_c' := G_z;
        Γ_c' := Γ_z;
        c := c+1;

G'' := sort ( G', primary key |Γ_i'|, secondary key decreasing |N_i| );
Γ'' := Γ' sorted in the same order as G'';

/* Algorithm is now similar to latter half of Heuristic #2 */

Π := ∅;                          /* start with empty partitioning */
for each c = 1...|G''| do        /* look at each set in sorted order */
    s := Γ_c'';                  /* s is a set of instances (rows) */
    if (∃t ∈ s and ∃κ ∈ Π) and (t ∩ κ = t) then
        ignore s;                /* set contains instance already in P */
    else
        Π = Π ∪ s;               /* add set to partitioning */

return Π;                        /* return the partitioning */
```

Figure 4.9 Algorithm 6 — partitioning heuristic 3

improvement over the previous heuristics for small files, and some people may find the numbers insignificant. We have limited the number of parameters to one, and compaction does not appear to have suffered. The results of this heuristic are in Table 4.7.

A flexible algorithm could tune the maximum number of allowable parameters to the size of the candidates — shorter candidates would not be allowed to have any parameters,

Table 4.7 Heuristic 3 compaction — % savings

instructions	average	min	max
0 — 2K	2.0	1.2	3.0
2K — 5K	1.7	1.0	3.9
5K — 10K	2.0	1.1	4.6
10K +	2.2	1.1	7.0

while candidates spanning many instructions could produce large enough space savings to sustain a bigger parameter cost. The longer the candidate, the greater the probability that there will be mismatches between operand values in the same column.

4.4.6 "Best of heuristic", and percentage improvement over FMW

The "best of heuristic" is the result of running each heuristic on a candidate, and then choosing the partitioning/procedures with the best space savings suggested by any heuristic. In many cases the best space savings for a binary are suggested by only one of the heuristics, but frequently the "best of heuristic" results for a binary are better than the results of any one of the three (see Table 4.8). This suggests that better results will come from compaction schemes that are tuned to several candidates, i.e. use one heuristic for candidates with many short instances, use another for candidates with fewer but longer instances, etc.

Table 4.8 Best of heuristic — % savings

instructions	average	min	max
0 — 2K	2.0	1.1	3.2
2K — 5K	2.4	1.1	6.4
5K — 10K	2.5	1.1	6.6
10K +	2.7	1.1	9.3

Our final results appear worse if compared with the reported space savings in [8] (4% average versus our 2.7% average). However, as mentioned at the start of this section, our implementation of the FMW algorithm revealed that its performance was quite poor when

working with disassembled SPARC code. A more interesting comparison is made on a binary-by-binary basis, and the FMW results are compared with our own "Best of Heuristic". We have calculated the percentage space savings improvement of our scheme over FMW and averaged out these results for each of the program size categories. As programs become longer, our algorithm outperforms FMW even more. The large variances point out that occasionally a program is encountered with plenty of redundancy. Overall, however, these results show that better space savings may be achieved if we use parameters to change similar strings of instructions into one procedure.

Table 4.9 "Best of heuristic" improvement over FMW

instructions	average (%)	min (%)	max (%)
2K — 5K	49.1	13.9	115.4
5K — 10K	60.8	6.8	173.6
10K +	84.4	17.1	566.7

4.5 Further transformations that could be pursued

The heuristics given above do not transform the candidates themselves, but only pick and choose among candidates. More aggressive schemes should seek to transform the candidate instances such that compaction opportunities are improved.

4.5.1 Dropping instructions from instances

If the parameters are necessary because of differences in the instructions at the beginning or end of candidate instances, then these instruction could be dropped. An increase in the potential space savings results since the cost of adding parameters for the offending instruction is significantly higher than the space saved by including the instruction in the procedure.

4.5.2 Wasp waisting

This is a variation of the previous technique, but the instruction causing the parameter is now in the middle of the candidate. To remedy this, candidates are split in two, and covered

by two subroutines: one for the part before the rejected instruction, and a second for that coming after. This technique is used by parallellizing optimizations, where two otherwise similar sequences of code, proceeding in parallel in the control flow graph, differ by only one instruction in the middle of the parallel sections [9]. The control flow graph looks like a wasp as the two parallel flows are pinched together at the offending instruction.

4.5.3 Parameterizing for instructions

In all of our discussions, we have required instances to be candidate matches — instruction operations must match between instances. This could be relaxed, as mentioned in passing earlier in this chapter, by parameterizing a procedure for differences in operations. For instance, instead of wasp waisting, which means creating two or more separate procedure bodies, a value is passed into the procedure as a parameter. This value is the selector in a case statement, with the instruction chosen depending on the original instance invoking the procedure. Some rearrangement of the body of the procedure would be necessary to make room for the case statement and the jump table.

4.5.4 Reduction of parameter cost

Which parameters are input only? Which are output only? Which are input and output? At present, all are considered to be input and output. The procedure abstraction does support "call by value" and "call by reference." Observing which parameters are "read-only" can reduce the number of copy instructions performed during procedure exit. Similarly, "write-only" parameters need not be assigned values at procedure entry, further reducing the number of statements needed for parameter handling. Data flow analysis can be applied to the candidates to obtain information on parameter usage.

4.6 Summary

Previous compaction techniques do not cope well in cases where procedures covering many instruction substrings require parameters. One scheme (FMW) accepts only exact matches of operand values between instructions. Another (Marks) uses at most one parameter, but only in the context of tailored interpretation, which is admitted to be an extreme form of compaction that trades off run-time performance for space savings.

Compilers of a decade ago may have generated code containing more exact matches between candidates than today's compilers, but in SPARC object code exact matches tend to be frequent but too short. Therefore, any improvement in compaction demands the parameterization of procedures so that similar, inexact, matches may be replaced with one procedure.

Our heuristics for selecting procedures outperforms the FMW scheme by 49.1% — 84.4%. This is an encouraging result, and justifies the effort spent developing the heuristics. The average overall space savings are at 2.0% — 2.7%.

5 Summary and Conclusions

5.1 Summary

We have observed that current research in compiler optimization focuses on program transformations to increase run-time speed, and to a lesser extent on reducing the size of object code. The latter goal deserves more attention. There are several reasons why "smallness" is desirable: smaller programs require less storage space, whether on disk or in main memory; in a distributed environment, network utilization and efficiency is improved if less bandwidth is required to transmit each binary; and a larger proportion of the object code fits into the instruction cache, thereby improving hit-miss ratios during instruction fetches. Several optimizations for speed also result in smaller binaries, and this indicates that the two goals (higher speed and less memory) do not necessarily conflict.

In addition to standard compiler transformations, we have reviewed several approaches to reducing the size of a program's machine representation. These approaches may be separated into two groups:

- those that create custom opcode and operand bit representations such that the new representation is smaller than that using the original machine code representation;
- techniques which identify repeating sequences of code and replace these sequences by a call to a single procedure.

We have studied and extended the latter group with an approach called *procedural abstraction.* There are several implementations of the technique which achieve various rates of compaction. The highest rates require run-time support in the form of an interpreter to decode the procedure calls. Others produce smaller space savings but are are easier to implement, needing only the rearrangement of the existing code and the addition of extra procedure calls and branches. Procedural abstraction leading to space savings without extra run-time support is called *compaction.* This distinguishes these algorithms from *compression,* which may produce smaller files which are not executable until they are processed by the complementary decompressor.

Repeating sequences of code may be constrained to be exactly alike, in which case there are no differences between instances in instruction operations or opcodes. A looser definition allows operand values to differ between instances, but operations must still match. In this latter case, procedures which cover all instances are *parameterized* for the differences. Previous researchers have declined to use parameters because of their high cost, and have assumed that any space savings attained through the removal of redundant code would be lost by the extra parameter copying instructions. We have shown this assumption to be incorrect. Our investigation into parameterized procedures has found that there may be many possible procedure covers for a given set of sequences. All instances need not be replaced by the same procedure, and this means that the maximal space savings are achieved when the number of parameters and and number of procedures are minimized. Finding this minimum requires a search of the parameterization/procedure possibilities, but given the large number of repeating sequences, the search cost would be prohibitive. Instead, we have enumerated the possibilities that lead to positive space savings, and then developed heuristics to choose the best of these.

We have implemented a prototype that accepts binaries as input, which then disassembles and passes them through a substring identification algorithm that has been modified to find similar section of code. This data is analyzed using the heuristic-based compactor. Our basis for comparison were the results of our implementation of the FMW algorithm applied to 200 UNIX utilities. When processing large binaries, our prototype produces 2.7% compaction on the average, ranging from 1.1% to 9.3%. This takes into account the parameter cost incurred by our scheme and is an 84.4% *improvement* over the FMW algorithm results.

5.2 Future Work

We have not examined the effect on run-time performance of our compaction algorithm. It remains to be seen whether acceptable compaction rates can be achieved while simultaneously limiting the effect of extra branches, calls, and parameter copying instructions in increasing run times. *Execution profiles* [17] may be used to identify

infrequently executed code, which is then compacted to a greater degree than frequently executed code. As we are working at the level of assembly language, the granularity of the regular UNIX profilers such as `gprof` [11] are far too coarse. Recent advances in profiling technology yield exact machine instruction usage frequencies while adding a respectable overhead [4], and this data can be put to use by the compactor.

Similar sections of code is another concept which should be probed further. There are two related directions which may yield positive results:

- As mentioned earlier, similar sections could differ in instruction operations as well as operands, with all differences parameterized. While it is easy to speculate on the costs of such a scheme, it is more difficult to determine the space savings. A similar prototype to that used in our research could be constructed, but with the understanding that parameters for a differing instruction will incur a high cost than those for differing operands.
- Identical sections of code do not differ in instruction operations and operands. A related concept is *idempotent sections of code*. These sections have the same instructions and produce the same results but the instructions have different orderings in the object file. These orderings are produced by optimizing compilers for modern pipelined processors as a way to increase total throughput given several functional units. Data flow analysis could identify these idempotent sections. The concept of similar sections of code could be further extended to include subgraphs of the object's control flow graph, where multiple copies of the subgraph can be removed given some constraints on how the subgraphs are entered and exited.

5.3 Conclusion

Code compaction can achieve space savings even if procedural abstraction is used on sections of code with operand differences. The technique can be applied during compilation, in which case it works in combination with other space-savings transformations, or after compilation as a system tuning tool. Finding good space savings

need not require an expensive search through possible procedure/sections combinations, as we have shown through our use of heuristics.

References

[1] Alfred V. Aho, Ravi S. Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1985.

[2] Frances E. Allen and John Cocke, "A Catalogue of Optimizing Transformations," in Rustin, P. (ed) *Design and Optimization of Compilers*, Prentice-Hall, 1973, 1-30.

[3] David F. Bacon, Susan L. Graham, and Oliver J. Sharp, "Compiler Transformations for High-Performance Computing," Technical Report UCB/CSD-93-781, Computer Science Division, University of California, Berkeley, 1993.

[4] Thomas Ball and James R. Larus, "Optimally Profiling and Tracing Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 4, July 1994, 1319-1360.

[5] Thomas Ball and James R. Larus, "Rewriting Executable Files to Measure Program Behavior," Computer Science Technical Report 1083, University of Wisconsin, March 1992.

[6] Timothy C. Bell, John G. Cleary, and Ian H. Witten, *Text Compression*, Prentice-Hall, 1990.

[7] John Cocke, "Global Common Subexpression Elimination," *Proceedings of the ACM Symposium on Compiler Optimization, SIGPLAN Notices,* Vol. 5, No. 7, 20 - 24, 1970.

[8] Chris W. Fraser, Eugene W. Myers, and Alan L. Wendt, "Analyzing and Compressing Assembly Code," *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, 117-121, 1984.

[9] Charles M. Geschke, "Global Program Optimizations," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1972.

[10] David Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, Vol. 23, No. 1, March 1991.

[11] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick, "gprof: a Call Graph Execution Profiler," *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, 120-126, 1982.

[12] Matthew S. Hecht, *Flow Analysis of Computer Programs*, North-Holland Elsevier, New York, 1977, 16-18.

[13] Eric C. R. Hehner, "Matching Program and Data Representations to a Computing Environment," Technical Report CSRG-44, Computer Systems Research Group, University of Toronto, November 1974.

[14] R. Nigel Horspool and N. Marovac, "An Approach to the Problem of Detranslation of Computer Programs," *The Computer Journal*, Vol. 23, No. 3, 1979, 223-229.

[15] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the Institute of Electrical and Radio Engineers*, Vol. 40, No. 9, September 1952, 1098-1101.

[16] David Keppel and Stephen Russell, "Faster Dynamic Linking for SPARC V8 and System V.4," CS&E Department Technical report 93-12-08, University of Washington, 1993.

[17] James R. Larus, "Efficient Program Tracing," *IEEE Computer*, Vol. 26, No. 5, May 1993, 52-61.

[18] James R. Larus and Eric Schnarr, "EEL: Machine-Independent Executable Editing," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 291-300, June 1995.

[19] Edward M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," *Journal of the ACM*, Vol. 23, No. 2, April 1976, 262-272.

[20] W. M. McKeeman, "Peephole Optimization," *Communications of the ACM*, Vol. 8, No. 7, July 1965, 443-444.

[21] Brian Marks, "Compilation to Compact Code," *IBM Journal of Research and Development*, Vol. 24, No. 6., November 1980, 684-691.

[22] Richard P. Paul, *SPARC Architecture, Assembly Language Programming, and C*, Prentice-Hall, 1994.

[23] Michael Rodeh, Vaughan R. Pratt, and Shimon Even, "Linear Algorithm for Data Compression via String Matching," *Journal of the ACM*, Vol. 28, No.1, January 1981, 16-24.

[24] Richard H. Stallman and Roland H. Pesch, "`gdb`: GNU's Source-Level Debugger," version 4.13, Free Software Foundation, January 1994.

[25] G. A. Stephen, "String Search," Technical Report TR-92-gas-01, School of Electronic Engineering Science, University College of North Wales, October 1992.

[26] H. Joseph Straight, *Combinatorics: An Invitation*, Brooks/Cole Publish Company, Pacific Grove, California, 1993.

[27] Thomas G. Szymanski, "Assembling Code for Machines with Span-Dependent Instructions," *Communications of the ACM*, Vol. 21, No. 4, 300-308, 1978.

[28] Mark Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, July 1984, 352-357.

[29] Niklaus Wirth, "A Plea for Lean Software," *IEEE Computer*, Vol. 28, No. 2, February 1995, 64-68.

[30] William Wulf, Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke, *The Design of an Optimizing Compiler*, Elsevier Computer Science Library, 1975, 11-43.

Glossary

compaction

In this thesis, compaction refers to the process of rewriting an already compiled program such that it is both smaller and immediately executable.

delay slot instruction

Found in RISC architectures, it appears after any instruction that could change the flow of control. These pipelined architectures execute one instruction while simultaneously fetching the next. However, fetching the instruction following a conditional branch may be incorrect, so the compiler is given the freedom to insert a suitable instruction in the delay slot. Complications for compaction appear when a delay slot instruction is also the destination of another branch.

horizontal partition

Instruction sequences are represented in this thesis as a string of instructions (i.e. from left to right on the page, or *horizontal*). Operands from these instructions are grouped into subsets where no two subsets share an operand with the same value. The collection of these subsets is a *partitioning*. A procedure call covering the sequence will need at most one parameter for operands in the same subset.

instance

One sequence of instructions that could be replaced by a procedure.

instance grouping

A collection of instruction sequences, each sequence of the same length; all instruction operations match between sequences, but the instruction operands may differ. Depending on the resulting space savings, these groupings may be mapped to one or many procedures.

intermediate representation (IR)

A format used by the compiler while translating the source language into the target language; this representation is easier to manipulate when performing many compiler tasks, including optimization.

procedural abstraction

The process of turning repeating instruction sequences into procedures.

suffix tree

A data structure providing a dictionary of substrings within a string.

tieing

If two or more parameters share the same values in each procedure call, then those parameters may be combined, or tied together and passed as one. Ties therefore help to reduce the number of parameters required for passing data into procedures.

vertical partition

A collection of instruction sequences, all having the same length and the same operations, may be represented by a two-dimensional array: sequences are the rows, operations and operands the columns. Operand values in each column (*vertical*) may be grouped into sub-sets of row indices with no two subsets sharing operands of the same value (*partitioning*). Therefore each column has a vertical partition.

Vita

Surname: Zastre Given Names: Michael Joseph

Place of Birth: Prince George, British Columbia, Canada

Education Institutions Attended:

University of Victoria	1993 —1995
Simon Fraser University	1989 —1993
Kwantlen College	1988 —1989

Degrees Awarded:

B.Sc. (Honours, First Class, Co-op)	Simon Fraser University	1993

Honors and Awards:

Natural Sciences and Engineering Research Council	
Summer Scholarship	1993
Simon Fraser University	
Open Undergraduate Scholarship	1993
Open Undergraduate Scholarship	1992
Open Undergraduate Scholarship	1991

Partial Copyright Licence

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library from any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: Compacting Object Code via Parameterized Procedural Abstraction

Author [redacted]

Michael Joseph Zastre

Date 1 Nov. ~~1996~~ 1995 mz