Pattern and Approximate-Pattern Matching for Program Compaction

Neil Johnson[⋆], Alan Mycroft

University of Cambridge Computer Laboratory
New Museums Site, Cambridge, CB2 3QG, United Kingdom
{nej22,am}@cl.cam.ac.uk

Abstract. The application of pattern and approximate-pattern matching to program compaction offers considerable benefits in restructuring code as a means of reducing its size. We survey the field of code compaction and compression, considering code size, the effect on execution time, and additional resources, and present a literature survey identifying representative papers.

Concentrating on compaction, we discuss phase-ordering and the two main code compaction techniques—code factoring and procedural abstraction. Relating to common subsequence computation we consider compact suffix trees, suffix arrays and suffix cacti. After introducing classical pattern matching, we review parameterized pattern matching, based on a modified suffix tree (the *parameterized suffix tree*).

We consider four supporting technologies: static single assignment (SSA) form simplifies data flow analysis; equivalence considers the issue of whether two sections of code are equivalent; normalization, which considers code structuring techniques to aid pattern matching; and register allocation, which we argue should be done after code compaction.

1 Introduction

Program space compaction has become an important and active area of research, primarily in the field of embedded systems. In this paper we survey recent developments in code compaction and compression techniques relevant to code size reduction, and conclude by identifying promising research directions.

A typical deeply-embedded system will consist of a compact code processor (e.g. ARM Thumb [31]), limited runtime storage (RAM) and limited code storage (ROM), together with I/O interfaces specified for a particular application. In the majority of cases, such systems are mass produced in large volumes. Clearly, at these quantities material costs account for a major part of the total product design cost. As the size of ROM required to store the program is proportional to the size of the program, a worthwhile cost reduction can be achieved through reducing the size of the program code.

[⋆] Supported by a grant from Advanced RISC Machines Ltd, Fulbourn Road, Cambridge, UK.

The field of code size reduction is becoming increasingly important as the size of embedded software grows. A survey of some of the key techniques—common subsequence computation, pattern matching and approximate-pattern matching—on which code size reduction transformations are built is appropriate, highlighting the synergies between a number of active research areas related to the two main size reduction strategies—compaction and compression.

A new generation of transformations are based on *procedural abstraction*, where common basic blocks are abstracted out into a single compiler-generated parameterized abstract procedure, with call-sites replacing the original code blocks, and *code factoring*, where instructions common to several blocks are merged together.

In both procedural abstraction and code factoring, the techniques of suffix tree construction, pattern and approximate-pattern matching techniques play a pivotal role: code-factoring utilises pattern matching to locate common sequences of instructions, and procedural abstraction can additionally increase the number of common sequences through approximate-matching and parameterization located at multiple usage sites.

Section 2 compares code compaction and compression, highlighting the similarities and differences between the two methods, how and when they are applied, and reviewing some of the more interesting work done in this area. Section 3 gives a summary of code factoring and procedural abstraction. Section 4 surveys developments in common subsequence computation and suffix trees, while sections 5 and 6 survey pattern and approximate-pattern matching respectively. Section 7 discusses a number of related topics that have a bearing on the performance and behaviour of the algorithms discussed in the previous sections. Section 8 concludes, with a view on future directions of profitable research for pattern-matching-based code compaction.

2 Code Compaction vs. Compression

Before describing in detail the operation of code compaction, we present a comparison of code compaction and code compression. We consider three characteristics of each method: code size, the effect on execution time, and additional resources required to support the method.

2.1 Code Size

Classical sequential data compression [59, 63, 64] offers a potentially high compression ratio because it does not include the data dictionary as part of the compressed data, instead relying on rebuilding the data dictionary during decompression. The widely-used Lempel-Ziv-Welch (LZW) algorithm [59] preloads the data dictionary with the data alphabet prior to compression/decompression, removing the need to include these in the compressed form. In a typical implementation for compressing text, each input symbol is eight bits wide, and each output token (dictionary pointer) is up to twelve bits wide, allowing for a 4,096-entry dictionary.

Two considerable disadvantages of these compression algorithms are (a) that the compression gain is achieved only over large blocks of data, and (b) there is no linear relation between a position in the original data and a position in the compressed form. The non-linear nature of program execution—with the presence of branches and procedure calls—means that single-block compressed data cannot be executed directly.

A different approach to data compression, the Burrows-Wheeler algorithm [13], analyses the data as a whole, reordering data blocks to achieve significantly better compression than the LZW approach. However, the high compression ratios are only achieved for large block sizes; for smaller blocks the performance degrades to that comparable with LZW. This algorithm would be suitable for whole-program compression, but not for procedure-based compression.

Code compression, in particular, transforms a block of executable code into a highly-compressed, non-executable block of data. Compression is applied either to the binary image for the entire application [37, 40] or on a procedure-by-procedure basis [34]. A variation on the compression scheme is instruction-level compression, wherein the instruction set of the target microprocessor is compressed, and an inline decompresser is placed between the program store and the microprocessor [31, 35].

On the other hand, *code compaction* can be achieved through two separate methods. Traditional code-space optimizations remove common subexpressions, dead and unreachable code, simplify expressions, together with code factoring and cross-jumping transformations [61], where common blocks of code on parallel branches within the flow-graph [3] are moved to a single block placed in a (pre/post-)dominant node, together with target-dependent peephole optimization (see [45] for a general introduction to these topics).

The second method is procedural abstraction, whereby common instruction sequences are replaced by one abstract procedure body and multiple call-sites. The task of finding common subsequences of instructions within the whole program is complicated by the presence of subtle variations in the instruction stream: register numbering, instruction ordering, sub-expressions etc. These can be ameliorated through parameterization and abstraction prior to register allocation.

Procedural abstraction is similar to Storer and Szymanski's [51] *external pointer macro*, where the pointers into the dictionary are implemented as calls to abstracted procedures. A similar scheme [39], using illegal processor instructions to indicate tokens, has shown compression ratios of between 52-70% over a range of large applications. For the same set of programs, the Unix utility *compress*—using LZW coding—achieved a compression ratio about 5% better. However, it could be argued that the encoding scheme may not be optimal because of the differing statistical distributions of opcodes and operands [7, 47].

2.2 Execution Time

Code *compressed* with the classical methods (see above) requires the data to be decompressed before it can executed. This decompression can either be applied to the whole program prior to execution, or on a block-by-block basis. In either case, this decompression time, t_d, is required before the program itself can begin execution. For small programs, t_d will be insignificant compared to the running time of the program, but for larger applications (web browsers, image viewers, etc) then t_d will become significant, leading to unacceptable delays while the program is decompressed.

Compaction does not suffer from this effect, as it does not need extra time to allow for the decompression of the program segment. However, the overall execution time of the program may be increased due to the overhead introduced by extra procedure calls.

2.3 Additional Resources

The primary advantage compaction has over compression is that it does not require any extra physical resources at runtime, such as a RAM buffer in which to decompress executable code or extra hardware to decompress the instruction stream on-the-fly. The cost implications should be self-evident.

Storer and Szymanski's [51] text compression algorithm places the data dictionary with the compressed and tokenized instruction stream—common code sequences are factored out to a dictionary, and replaced with special tokens that point to the relevant entry in the dictionary. This method is very similar to procedural abstraction, but without the capability of parameterization.

A considerable advantage compaction has over compression is that compaction can be applied to the program at a higher level than object or assembly code during the build process. We consider that by applying the transformations at a higher intermediate-code level, we benefit by (a) having access to the original flow graph and data structures generated within the compiler, (b) not having to tackle the problems associated with post-register allocation and post-code generation transformations, and (c) returning an equivalent degree of code compaction through fewer operations on higher-level code symbols.

2.4 A Combined Approach

While both methods operate independently of each other, it is entirely possible to apply *both* methods to a program in order to achieve a very high degree of code size reduction—first the code is compacted to factor out common sequences of code, then compressed to produce the final binary image for installing in the embedded system.

Both compaction and compression rely, to a varying degree, on finding repetition (redundancy) within a given block of data. In program size reduction, compression operates on the binary executable file, replacing repeating long sequences of bits with shorter token bit-patterns. Compaction, through code factoring and procedural abstraction, operates at a much higher, symbolic, level where it can be more flexible in its approach to pattern matching (§5.1).

It could be argued that compaction, by factoring out repeating code, reduces the number of opportunities for a compressor to find repeating patterns to compress. However, because of the effects of target-dependent code generation (system libraries, standard function prologue and epilogue code, instruction bit patterns, etc) we believe there still to be sufficient repetition within a program's binary file to warrant further investigation of the combined effect of compaction and compression.

Some possible gains include:

- By reducing the uncompressed binary data, both decompression time and buffer space are also reduced.
- A procedure-based block compression system could operate with a smaller runtime cache.
- Either compaction or compression may be applied as and when needed, rather than an all-or-nothing approach (e.g. use compression during development, and reserve compaction if the target hardware specification imposes such constraints as to warrant compaction, or vice versa).
- A (possibly marginal) gain in code size reduction compared to either compaction or compression alone.
- Less aggressive, and so possibly faster, compaction and/or compression required to achieve the target code size, balancing the runtime penalties associated with compaction against product cost and development time.

However, we argue that sufficient code size reduction should be achievable from code compaction alone, without incurring the greater overheads associated with runtime decompression.

3 Code Compaction Methods

Code compaction is a transformation applied to program code during compilation in order to reduce its size, yet maintaining the property of direct execution. This section describes the phase-ordering issue with respect to other compiler stages (e.g. register allocation (§7.4), common subexpression elimination, strength reduction, etc) and the two main code compaction techniques—code factoring and procedural abstraction.

3.1 Phase Ordering

The order in which transformation phases occur during compilation is called the *phase order*. Early compilers applied optimization phases without consideration of the effects each phase had on subsequent optimization phases. One approach to this phase-ordering issue [17] applies genetic algorithms to search for the best sequence of optimizations to apply to a given program, where the cost function is the size of the program code.

However, compaction through code factoring and procedural abstraction is sensitive to the order of previous code transformations. For example, common subexpression elimination (CSE) may remove duplicate subexpressions from a block of code, but may well also reduce the degree of matching between the block and a similar block; had the CSE *not* removed the duplicate subexpression, the entire block may well be abstracted out to a procedure, reducing the code size by a greater amount than that achieved by the CSE optimization alone. Register allocation has similar effects: increasing the degree of variation between equivalent code blocks reduces the opportunity for size reduction through code compaction.

The majority of previous work concerning procedural abstraction has applied transformations to code after the link stage (so-called *post-link optimizations*) [16, 22, 46, 62]. Early work in the field of procedural abstraction at the intermediate code level *before* register allocation and linking looks promising—Runeson [49] achieves a 21% reduction in the number of intermediate code instructions

3.2 Code Factoring

Code factoring is the process of moving common code elements from two or more basic blocks into dominator or post-dominator nodes. For example, two equivalent instructions I_1 and I_2 that appear in two blocks dominated by node N, and where I_1 is not dependent on any instruction after N (and similarly for I_2), can be moved up into N and merged into one instruction, thus saving one instruction. Conversely, where two or more procedures share a common tail section, one tail section can be left in its parent procedure, and cross-jumps inserted in place of the other tail sections.

Code factoring is susceptible to variations (noise) caused by, for example, register allocation (§7.4) and other optimizations like common sub-expression elimination, strength reduction and loop-invariant code motion. Clearly, an instruction (or block of instructions) can only be moved if (a) they use the same registers, and (b) the instruction(s) to be moved do not depend or anti-depend on the instructions over which they will be moved. In some cases it is possible to insert register-register copies to fix up register assignment variations between blocks. Ideally, this process should be performed before register allocation, removing the need for explicit register-register copies.

At its simplest, code factoring is a procedure-oriented transformation—instructions are moved between basic blocks within the flow-graph of a procedure (see Figure 1). Cross-jumping extends this to operate inter-procedurally, but with the restriction that only procedure suffixes[1] can be merged.

Generally, the size of patterns for code factoring are small, limited to the size of basic blocks. Cross-jumping can extend this further, but still requires a perfect

[1] I.e. sections of code between an exit-dominator node and the exit node.

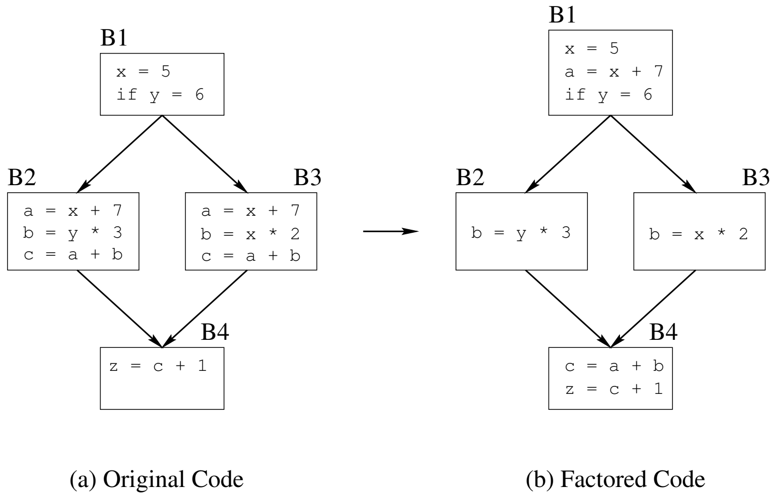

Fig. 1. Code factoring example. (a) The original code prior to code factoring. (b) Shows the results of code factoring—the first instruction in blocks B2 and B3 is moved up to B1 (saving one instruction) while the last instruction, also common to both blocks, is moved down into B4, saving another instruction.

match between the multiple code sequences. Traditional code factoring requires a perfect match, but parameterization ideas from our procedural abstraction approach can be used to factor near matches too.

3.3 Procedural Abstraction

Procedural abstraction takes a whole-program view of code compaction. Common sequences of instructions from any of the procedures (including library code) within the program can be extracted, and replaced by one abstract procedure and a number of calls to that abstract procedure in place of the original code sequences. The potential costs of procedural abstraction are register shuffling before and after the procedure call, and the extra processing overhead associated with the procedure call and return. The former can be ameliorated through post-compaction register allocation (§7.4), while the latter can be accommodated either through software support [38] or hardware support, e.g. passing arguments and results through registers, and using multiple return link registers or implicit push-on-call.

Because these abstract procedures are generated internally within the compiler, there is no requirement for standard argument-passing and result-returning conventions to be obeyed. This is particularly advantageous for abstract procedures that may take multiple arguments and return values—these can be mapped to processor registers during register allocation, and treating each call to the abstract procedure as being an instruction that needs arguments in certain registers, and returns results in certain registers.

An early form of procedural abstraction, due to Marks [42], is related to table-based compression, where common instruction sequences were placed in a table, and pseudo-instructions inserted into the instruction stream in place of the original instruction sequences (*tailored interpretation*). Fraser *et al.* [25] developed an algorithm which does not need runtime support from the operating system, instead inserting `call` instructions to procedures that are generated during the transformation process.

A related issue is the placement of the return address—if placed on the stack then stack adjustment operations must be factored into the cost function; if placed in a register then either a free register must be found, or a register made available through some other mechanism.

As for any optimization transformation, there is a cost associated with procedural abstraction: if the cost is negative (ie. there is *reduction* in code size) then the given block should be abstracted; if zero then there is no improvement to be had from the transformation; and if positive then code size would *increase* if the transformation were to be applied.

A simplistic upper-bound cost function, C^u, for procedural abstraction is suggested below for N instances of abstracted block b, where *frame* is the abstract procedure prologue/epilogue frame, and *args(b)* and *results(b)* are the sets of arguments and results of b respectively:

$$C^u(b) = |b|(1-N) + |\mathit{frame}| + N|\mathit{call}| + (N-1)(|\mathit{args}(b)| + |\mathit{results}(b)|) \quad (1)$$

For example, for a block of ten instructions, which appears fives times in a program, and takes three arguments and returns one result, and assuming that an abstract procedure frame takes two instructions, and each call takes one instruction, then the cost for abstracting out this block is -17, ie. we *save* seventeen instructions if we abstract out this block.

The power of procedural abstraction is enhanced yet further by parameterizing the abstract procedures [62]. This allows a degree of approximate-matching between the code blocks; the differences are parameterized, and each call site specifies through additional parameters the required behaviour of the abstract procedure.

Previous work has considered procedural abstraction both pre- [49] and post- [21, 62] register allocation. It can be argued that the latter is less efficient than the former since the work of the register allocator has to be undone to some extent, either through register renaming [21] or through parameterization [16]. Ideally, procedural abstraction should be applied at the whole-program *post-link* stage, wherein the optimizer has access to the entire body of code from which to discover and extract abstract procedures. Previous post-register allocation research has been at this post-link stage [46], and thus must also tackle both target-specific disassembly and register variation.

The area of parameterized procedural abstraction has concentrated mainly on the use of suffix trees (§4.1) to highlight similar sections of parameterized code. An alternative approach is *program slicing* [58][2]. A program slice is a reduced

[2] The interested reader wish to refer to [54] for a survey of program slicing techniques.

subset of a program that is guaranteed to represent the original program within the domain of the specified subset of behaviour. The *statement-minimal slice* is one such that no other behaviour-equivalent slice of the program has fewer statements.

At a procedural level, one could abstract out a common slice from a number of donor procedures, replacing each slice with a call to the abstract procedure, and creating one instance of the slice within an abstract procedure frame. This approach is clearly demonstrated by Komondoor and Horwitz [**?**] when applied to finding duplicate code sections within legacy C source code.

Pattern matching to identify abstract procedures is the greater challenge: not only can the common sequences be different lengths, but the search region is much larger (the whole program) and a degree of mismatch between similar patterns is allowed. We perceive that finding the space-optimal set of abstract procedures is crucial to aggressive code compaction. However, we must not ignore the inherent complexity of this task—finding the statement-minimal slice, for example, is equivalent to solving the halting problem[3].

4 Common Subsequence Computation

Both code factoring and procedural abstraction are based on finding and removing repeating sequences of instructions within a program. For procedural abstraction this is a two-stage process: extracting all the subsequences within procedure bodies (subsequence computation), and then matching subsequences both within a procedure and across multiple procedures (pattern and approximate-pattern matching).

4.1 Compact Suffix Trees

The standard method for subsequence (or substring) extraction is Weiner's suffix tree [57], enhanced by McCreight [43], and refined further by Ukkonen [55] into an on-line (left-to-right) construction algorithm; all three algorithms operate in space and time of $O(n)$. Giegerich and Kurtz [27] summarise the algorithms, highlighting the similarities and differences between them. Throughout this paper we assume the Ukkonen algorithm for suffix tree construction.

As an example, consider the short string S = *xyxyz\$*, where *\$* denotes the string terminal symbol. The suffix tree for S is shown in Figure 2. For each non-leaf edge (i.e. an edge not ending in a leaf node) a weight is calculated as the product of the number of leaf nodes reachable from any path from the start node though the edge, and the length of the substring represented by that edge.

Choosing which substring that would result in the greatest code space saving is simplified to finding the non-leaf edge with the greatest weight. In the example given, edge *xy* has the greatest weight ($w = 4$), resulting in a potential code

[3] This complexity can be managed by finding near-minimal slices using dataflow techniques.

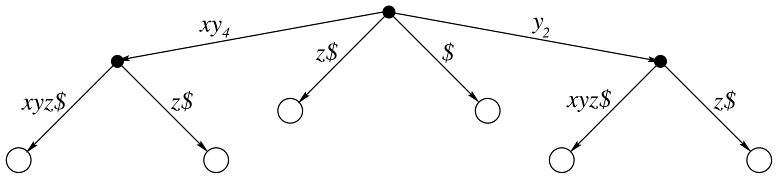

Fig. 2. The Compact Suffix Tree for $S = xyxyz\$$. Hollow nodes indicate leaf nodes, and branch labels denote the substrings within S. Weights w for non-leaf edges are shown as subscripts.

saving of two instructions[4]. Where multiple non-leaf edges exist in sequence then a *super-edge* exists between the start of the first edge and the end of the second edge. Its weight is the product of the sum of the lengths of the component substrings and the number of leaves of the super-edge. This is illustrated in Figure 3.

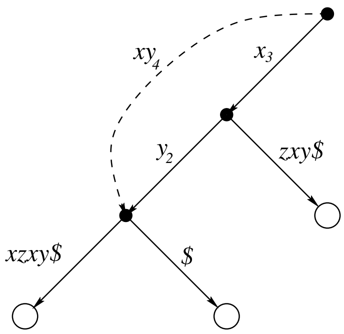

Fig. 3. Part of the suffix tree for $S = xyxzxy\$$. The (dashed) super-edge for substring xy has a weight greater than either of its two component edges.

Their construction finds all exact matches but suffix trees also provide candidates for searching for near matches elsewhere (*c.f.* p-suffix tree, §6.1). For this purpose, the performance of both simple and complex searching operations (e.g. regular expression matching) on suffix trees is good. However, a typical implementation might use fifteen bytes per text symbol. As Kärkkäinen [32] points out, for large text strings, the ability to fit the entire data structure in physical memory (as opposed to virtual memory) is important for computation within acceptable time.

[4] We ignore for the moment the cost of the additional `call/return` instructions. For procedural abstraction a simplistic space saving σ is given by $\sigma = w - |S|$, accounting for the size of the abstract procedure.

4.2 Alternatives to the Suffix Tree

Suffix Array [41] is also constructed in linear time and space, but consumes about half the space of an equivalent suffix tree (typically six bytes per symbol). The performance of simple searching operations is also comparable, but more complex searches are slower on suffix arrays.

Suffix Cactus [32] is a hybrid of the suffix tree and the suffix array, offering space and time performance somewhere between the two: faster than the suffix array for complex searches, more space-economical than the suffix tree.

5 Pattern Matching Algorithms

Pattern matching is a general form of string matching, where one or more given patterns are searched for within a block of data. In the context of code compaction, pattern matching is used to find repeating sequences of instructions that are suitable for either code factoring (§3.2) or procedural abstraction (§3.3).

5.1 Classical Algorithms

Efficient pattern matching has been derived from the original Knuth-Morris-Pratt (KMP) [36] and the Boyer-Moore (BM) [12] substring searching algorithms. The BM algorithm is generally the faster of the two algorithms, although its worst-case behaviour is $O(mn)$ (where n is the length of the string, and m is the length of the search pattern). Sunday's modified BM algorithm [52] runs approximately 20% faster for normal English text, dropping down to about 10% faster for longer strings.

Both the KMP and BM algorithms are tailored towards searching text strings. For code compaction we are interested in searching a potentially larger alphabet consisting of all the instructions present in the code stream at the desired level of compaction[5].

The basic mechanism for pattern matching is the finite state machine (FSM) [2]. The candidate pattern is generated from the suffix tree (§4.1) from which the FSM state table is created. The FSM is then applied to the candidate string, from which the algorithm either returns a pointer to the first occurrence of the pattern, or nothing if the pattern does not occur within the string.

5.2 Application of Pattern Matching

The code factoring transformation moves matching sections of code out from donor blocks and into a target block. Pure procedural abstraction (i.e. without

[5] As stated earlier, code compaction can be applied to intermediate codes, not just processor instructions.

parameterization) operates in a similar fashion, except the matching code sections are removed from the donor blocks and replaced with calls to the abstract procedure.

Pattern matching for code factoring requires knowledge not only of the instructions themselves, but also of the structure of the flow-graph—code can only be factored where there is a common predecessor or successor node in the flow-graph (see Figure 1).

At the intermediate code level, code factoring is not affected by register mismatches between patterns, as at this stage register allocation has not been performed so we do not need to undo the work of the register allocator, either through register-renaming or inserting additional instructions for register-register copying.

Pattern matching can also be applied at the post-link stage as a peephole optimization [50]. Optimizations applied at this level are simple control-flow optimizations—jump-to-jump elimination, loop restructuring. Patterns match basic blocks and flow-control instructions, with the execution order of statements preserved.

As well as linear sequences of instruction codes, another approach is to build a tree of the code structure and apply pattern matching to the tree [29]. An advantage of this approach is that it can be applied much earlier in the compilation process[6] than classical linear substring searching. Application of the algorithm at this earlier phase can avoid wasted effort during later code generation phases. It has been shown [15] that tree pattern matching can be performed in $O(n \log^3 m)$ time.

6 Approximate-Pattern Matching

The effectiveness of parameterized procedural abstraction is limited by the flexibility of the pattern matching algorithm thus employed—clearly, if more blocks of near-equivalent code can be discovered, so the resulting reduction in code size will be greater. The most important development in this area is *parameterized pattern matching.*

6.1 Parameterized Pattern Matching

Originally developed by Baker [8] for software maintenance, parameterized pattern matching replaces names within a body of code with generic parameter symbols. Subsequence computation and matching are then performed on the parameterized strings.

With these *parameterized strings* (p-strings), Baker defines a new data structure—the *parameterized suffix tree* (p-suffix tree)—suitable for parameterized pattern matching. Further developments in this area have generalised the BM algorithm

[6] In the majority of compilers, some tree structure (e.g. an abstract syntax tree) is generated prior to code generation.

to parameterized pattern matching [10]; the algorithm is shown to execute in $O(n \log min(m,p))$ time (where n is the length of the text, m is the length of the pattern, and p is the number of distinct parameter symbols) and preprocessing time of $O(m \log min(m,p))$.

Baker's method for parameterized pattern matching is sensitive to the size of the alphabet of the candidate search string, where the worst-case running time is $O(n\,|\Pi| + \log(|\Sigma| + |\Pi|))$ for input of length n constructed from an alphabet Σ and parameter symbol alphabet Π. Amir *et al.* [5] have subsequently shown that this can be achieved in $O(n \log min(m, |\Pi|))$ time.

From Baker [9], the p-suffix tree is a suffix tree constructed from p-strings. P-strings are constructed from $(\Sigma \bigcup \Pi)^*$, where Σ is the finite alphabet of ordinary symbols and Π is the finite alphabet of parameter symbols, and both are disjoint from each other and from the set of non-negative integers. Then, two strings S and T are said to be a *parameterized match* (p-match), if S can be transformed into T through a simple parameter mapping.

Let S be a string, then S_o, its *occurrence-chained* string (o-string), is such that each occurrence of a parameter symbol is replaced by an integer—a *parameter pointer*—representing the difference in position compared to the previous occurrence of that parameter symbol. The p-match test is applied to o-strings, which is reduced to a simple linear-time string compare. For example, if $\Sigma = \{a,b,c\}$, $\Pi = \{x,y\}$, $S = axbxcxc$ and $T = aybycyc$, then $S_o = a0b2c2c$ and $T_o = a0b2c2c$. Clearly, $S_o = T_o$.

In the case of multiple parameters then multiple occurance chains are used to represent the different parameter chains. For example, if $\Sigma = \{a,b\}$, $\Pi_1 = \{x,y\}$, $\Pi_2 = \{c,d\}$, $S = axbxcxc$ and $T = aybydyd$, then $S_o = a0_1b2_10_22_12_2$ and $T_o = a0_1b2_10_22_12_2$. Again, $S_o = T_o$.

Construction of the p-suffix tree is similar in principle to construction of the compact suffix tree (§4.1). The additional effort is in recomputing the parameter pointers within the suffices, which is achievable in linear time. The re-computation function, *f*, is given by:

$$f(b,j) = \begin{cases} 0 & \text{if } b \geq j \\ \\ b & \text{otherwise} \end{cases} \tag{2}$$

for the jth symbol of $psuffix(S,i)$ and corresponding symbol $b = S_{j+i-1}$ of S_o, and where $psuffix(S,i)$ is the ith p-suffix of a p-string of length S. From the p-string and f we can generate the p-suffix tree. The p-suffix tree of $S =$ *axbxcxc\$* is shown in Figure 4.

Parameterized pattern matching has several advantages for code compaction when compared to classical pattern matching: it provides an elegant mechanism for identifying equivalent sections of code that differ by specified parameters (i.e. instruction sequences at specified locations); it allows many more candidate code blocks to be mapped onto an abstract procedure than would otherwise be obtained through classical pattern matching; it offers a solution that operates in near-linear time (of the order of $O(n + \log m)$).

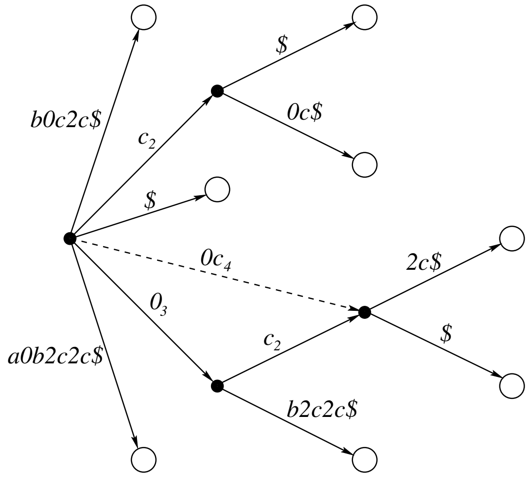

Fig. 4. P-suffix tree for p-string $S_o = a0b2c2c\$$. Weights are shown attached to non-leaf edges, and branch labels denote the parameterized substrings within S_o. The substring with greatest weight is $0c$, with $w = 4$.

The additional overheads associated with parameterized pattern matching are two-fold. At compile-time we pay the penalty of subsequence computation—through the p-suffix tree—to identify sections of code suitable for abstraction. At runtime, passing parameterized values to (abstract) procedures places extra burden on the stack, or extra pressure on the register allocator (§7.4). However, given the demand for more compact code, we argue that these overheads are acceptable given the potential gains in reduced code size.

7 Related Topics

Pattern matching forms part of the general solution to the code compaction challenge. Supporting pattern matching are a number of other key techniques, which are described in this section.

7.1 Static Single Assignment

A program rewritten in Static Single Assignment (SSA) form [20], in which each variable assignment is replaced by an assignment to a new unique variable, has properties which aid data-flow analysis of the program. The majority of classical optimizations are simplified by SSA-form, together with newer optimizations (described above) which make heavier use of information derivable from the flow-graph. Figure 5 gives a block of code and its SSA-form equivalent.

Two important points of SSA-form are shown in Figure 5. In order to maintain the static single assignment property of SSA-form we insert ϕ-functions [19] into the program at points where two or more paths in the flow-graph meet—in

```
c = 10;              c1 = 10;
x = 0;               x1 = 0;
y = 1;               y1 = 1;
do {                 do {
                       c2 = φ(c1, c3);
                       x2 = φ(x1, x3);
                       y2 = φ(y1, y3);
  c = c − 1;           c3 = c2 − 1;
  x = x + 1;           x3 = x2 + 1;
  y = y << 1;          y3 = y2 << 1;
} while(c);          } while(c3);
print(c, x, y);      print(c3, x3, y3);

(a)                  (b)
```

Fig. 5. (a) Original and (b) SSA-form code for discussion. The ϕ-functions maintain the single assignment property of SSA-form. The suffices in (b) make each variable uniquely assigned while maintaining a relationship with the original variable name.

Figure 5 this is at the top of the `do...while` loop. The ϕ-function returns the argument that corresponds to the edge that was taken to reach the ϕ-function; in our example, for the variable `c` the first edge corresponds to $\mathtt{c}_1$ and the second edge to $\mathtt{c}_3$.

The second point is that while an assignment to a variable is instantiated only once, that assignment can be executed one or more times at runtime, i.e. *dynamically*. For example, for variable $\mathtt{c}_3$ there is only one statement that assigns to it, but that statement is executed ten times within the loop.

Once in SSA-form data-flow analysis becomes trivial. Generation of def-use chains is straightforward—for each SSA variable there is only one definition and zero or more uses. Consider $\mathtt{c}_3$: it has one definition and two uses (once in the loop test and once as an argument to `print`).

Transformations based on def-use chains become equally simple: dead-code elimination [48], common-subexpression elimination [4], loop-invariant code motion [18], and so on. Consequently, analysis is faster, safer and potentially more powerful given the defined constraints imposed by SSA-form.

7.2 Equivalence

Equivalence—deciding whether two or more distinct and separate sections of program are equivalent—is very important for parameterized pattern matching (§6.1). Multiple code blocks that are equivalent can be replaced by multiple calls to a single abstract procedure, so reducing the size of the program.

Finding a degree of equivalence between two code blocks allows parameterization of the mismatch between blocks. For instance, block $\mathtt{B}_1$ may differ from block $\mathtt{B}_2$ by one instruction; this difference can be parameterized with an `if`...`then`...`else` placed around both instructions, and a selection parameter passed to the abstract procedure to control which of the two instructions is to be executed. Other near-equivalences can be merged into abstract procedures in a similar way, the limit being when there is no further code size reduction benefit in mapping any more code sequences onto an abstract procedure.

An equivalence relation can be expressed as bisimulation equivalence [44], which is applied to a labelled transition system describing the program and its properties. There has been considerable interest in bisimulation equivalence, including program verification and abstraction [28], program derivation [33], and equivalence checking [24]. Supporting work has concentrated on efficient implementation [23] and minimisation [11].

7.3 Normalization

Both classical pattern matching (§5.1) and parameterized pattern matching (§6.1) attempt to find multiple occurrences of patterns within a body of code. Clearly, if we restrict the number of ways in which code can be written then we stand a better chance of finding more instances of given patterns.

Rewriting code sequences into normal forms (*normalization*) is a powerful aid to subsequence computation, and thus to procedural abstraction, code factoring and, ultimately, code compaction. The use of high-level languages already offers a degree of normalization, in that typical software structures (loops, selection, expression evaluation, etc) are generally represented by similar internal data structures for each code structure, and so would exhibit tendencies to generate intermediate code of near-normal form. For identifying near matches it helps if pieces of code become the same, but it is a problem if two similar bits of code can be rewritten to be far apart.

Control-Flow Normalization [6] consists chiefly of translating `goto`s into `while`s (while-conversion) and/or `if`s (if-conversion). Average case complexity is linear in time, with the worst-case complexity of the order of $O(n^2)$.

Control-flow normalization is also used in reverse engineering of legacy systems [56]. Here, the concern is to rewrite legacy applications into a structure that aids both understanding of the program and subsequent maintenance of that program.

In both cases the process involves constructing the control flow-graph of the source program on which while-conversion and if-conversion transformations are applied. Within the context of a compiler, we already potentially have access to a control flow-graph, so adding control-flow normalization to an existing compiler would not require any extra data structures to be built; rather, the transformations could be applied to the compiler's control flow-graph prior to intermediate code generation.

Expression Normalization It is not clear that normalization can be used to simplify all matching or approximate-matching problems. For example, suppose $C[e]$ is a command containing expression e, to be nested within various loops for which $a + b$ is an invariant expression and t a variable not otherwise occurring. Consider the functions:

$$\begin{array}{l}\mathtt{f1()\{while(...)\ C[a+b];\}}\\ \mathtt{f2()\{t=a+b;while(...)\ C[t];\}}\end{array}$$

These differ solely by loop-invariant lifting of the expression $a + b$ and would hence like to be considered as identical code. However, on a textual level they differ significantly. We might take the view that $\mathtt{f1}()$ should be normalized into $\mathtt{f2}()$ or perhaps vice-versa.

Now suppose we normalise $\mathtt{f2}()$ to $\mathtt{f1}()$, by expanding all loop invariants. Then 'normalising' the following two functions would increase their textual differences:

$$\begin{array}{lcl}\mathtt{g1()\{t=a+b;while(...)\ C[t];\}} & \mapsto & \mathtt{g1'()\{while(...)\ C[a+b];\}}\\ \mathtt{g2()\{t=a+c;while(...)\ C[t];\}} & \mapsto & \mathtt{g2'()\{while(...)\ C[a+c];\}}\end{array}$$

Clearly, from a textual viewpoint the commonality is reduced. However, parameterized matching would match these two function, identifying b and c as parameterized symbols.

On the other hand suppose we normalise $\mathtt{f1}()$ to $\mathtt{f2}()$, by lifting all loop invariants. Then 'normalising' the following two functions would again increase their textual differences:

$$\begin{array}{lcl}\mathtt{h1()\{while(...)\ \{while(...)\ C[a+b];\}\}} & \mapsto & \mathtt{h1'()\{t=a+b;while(...)\ \{while(...)\ C[t];\}\}}\\ \mathtt{h2()\{while(...)\ C[a+b];\}} & \mapsto & \mathtt{h2'()\{t=a+b;while(...)\ C[t];\}}\end{array}$$

In this case, however, while abstraction could be applied twice (for "$\mathtt{t = a + b}$" and "$\mathtt{while(...)\ C[t]}$") this is potentially not as space-saving as abstracting out "$\mathtt{while(...)\ C[a + b]}$" from the pre-normalized case[7].

Even avoiding the issue of computability of algebraic identity (by considering only schematic, or Herbrand-valid, transformations) we find that there does not appear to be a single notion of normalization— hence we may require to search for matches or approximate-matches subject to a notion of equivalence.

7.4 Register Allocation

Register allocation is applied to intermediate code to lower it closer to the final target code. The intermediate code tree is analysed, typically through graph

[7] This would not be the case, however, if the two expressions for `t` were significantly different.

colouring [14], and registers assigned to nodes, as well as spills and loads inserted to accommodate the graph colouring algorithm. Graph colouring is an NP-complete problem [1]; all solutions to the graph-coloring problem are only near-optimal [26].

We argue that code compaction should be performed *before* register allocation for three reasons. Firstly, register allocation increases the variation within code sequences on which pattern matching will be subsequently applied. The expected effect of this is to reduce the ability of a given pattern-matching algorithm to find equivalent and near-equivalent sections of code for factoring or abstraction. Secondly, the effort expended in allocating registers to instructions which are then removed by code compaction transformations is wasted. Thirdly, the code transformations must undo the work of the register allocator through a combination of re-allocating registers and inserting register-register copy instructions.

Sweany and Beaty [53] implement post-compaction register allocation in the ROCKET retargetable C compiler for VLIW architectures. Runeson [49] also applies register allocation after procedural abstraction, achieving significant savings in intermediate instructions, averaging $\approx 21\%$ size reduction.

7.5 Parallels in Signal Processing

A common task in signal processing is to look for an approximate match of a signal within another noisy signal. One example might be to attempt to identify a clock-face in the video image by finding all candidate circle using a Hough transform [30] in which the convolution of the image with all circles is taken (there are three-dimensions worth—two for the position of the centre and a further one for the radius value).

Another example might be that of ‘periodogram’ for a time series [60] where we wish to find a pattern of length T of a signal $f(x)$ which near-repeats by computing

$$g(x) = \int_0^T f(t).f(x+t)dt \qquad (3)$$

One advantage is that such systems cope with noisy matches quite well; a disadvantage for our purposes is that there is no clear map of our problem into such a framework and the implicit linearity assumptions (note in the periodogram the assumption that the repeated pattern occurs at the same frequency as the original).

8 Promising Directions

Program compaction is an important and active area of research. The application of pattern and approximate-pattern matching to this problem offers considerable benefits, not only in finding common subsequences, but also in restructuring code as a means of preprocessing the program to aid the pattern matching algorithms.

We have illustrated the issues surrounding code compaction and compression, the trade-offs associated with each method, and described the two main code compaction techniques: code factoring and procedural abstraction.

An important aspect of pattern matching is common subsequence computation—finding the longest common subsequnce that appears multiple times within a block of code. Ukkonen's on-line suffix tree algorithm builds a tree structure describing all the substrings within a string of symbols, and has the benefit of $O(n)$ complexity. We have also briefly considered the suffix array and suffix cactus as alternatives to the suffix tree.

The application of classical pattern matching is restricted to finding exact matches between substrings. We have shown that the more powerful approximate-pattern matching algorithms, based on suffix trees, offer a more powerful tool for code compaction. Indeed, the application of Baker's p-suffix tree and parameterized pattern matching offers a powerful means of finding and mapping more program blocks onto abstract procedures than traditional pattern matching algorithms.

We believe that the development of normalization prior to pattern matching, together with the application of equivalence matching, offers additional opportunities for matching and, ultimately, code size reduction.

Equally, we believe that SSA form, when combined with normalization and equivalence, provides a powerful framework in which to develop optimization strategies aimed at code size reduction via procedural abstraction. It is this framework that will be the subject of further research.

References

1. Aho, A., Hopcroft, J., and Ullman, J. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.
2. Aho, A. V., and Corasick, M. J. Efficient string matching: An aid to bibliographic search. *Comm. ACM 18*, 6 (June 1975), 333–340.
3. Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.
4. Alpern, B., Wegman, M. N., and Zadeck, F. K. Detecting equality of variables in programs. In *Proc. Conf. Principles of Programming Languages* (January 1988), vol. 10, ACM, pp. 1–11.
5. Amir, A., Farach, M., and Muthukrishnan, S. Alphabet dependence in parameterized matching. *Information Processing Letters 49*, 3 (1994), 111–115.
6. Ammarguellat, Z. A control-flow normalization algorithm and its complexity. *IEEE Trans. on Software Engineering 18*, 3 (March 1992), 237–251.
7. Araujo, G., Centoducatte, P., Cortes, M., and Pannain, R. Code compression based on operand factorization. In *Proc. of 31st Annual International Symposium on Microarchitecture (MICRO'31)* (December 1998), pp. 194–201.
8. Baker, B. S. On finding duplication in strings and software. Tech. rep., AT & T Bell Laboratories, Murray Hill, New Jersey 07974, 1993.
9. Baker, B. S. A theory of parameterized pattern matching: Algorithms and applications (extended abstract). In *Proc. of the 25th Annual Symp. on Theory of Computing* (San Diego, California, 1993), ACM, pp. 71–80.

10. Baker, B. S. Parameterized pattern matching by Boyer-Moore-type algorithms. In *Proc. of the 6th ACM-SIAM Annual Symp. on Discrete Algorithms* (San Francisco, California, 1995), ACM, pp. 541–550.

11. Bouali, A., and de Simone, R. Symbolic bisimulation minimzation. In *Proc. of the Intl. Conf. on Computer-Aided Verification (CAV'92)* (Montreal, June 1992), G. V. Bochmann and D. K. Probst, Eds., vol. 663 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 96–108.

12. Boyer, R. S., and Moore, J. S. A fast string searching algorithm. *Comm. ACM 20*, 10 (October 1977), 762–772.

13. Burrows, M., and Wheeler, D. J. A block-sorting lossless data compression algorithm. Tech. rep., DEC Syetems Research Center, Palo Alto, CA, May 1994.

14. Chaitin, G. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices 17*, 6 (June 1982), 98–105.

15. Cole, R., and Hariharan, R. Tree pattern maching and subset matching in randomized $\mathrm{O}(n \log^3 m)$ time. In *Proc. 29th Annual ACM Symp. on Theory of Computing* (1997), ACM, pp. 66–75.

16. Cooper, K. D., and McIntosh, N. Enhanced code compression for embedded RISC processors. In *Proc. ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI'99)* (May 1999), vol. 34, ACM, pp. 139–149.

17. Cooper, K. D., Scheilke, P. J., and Subramanian, D. Optimizing for reduced code space using genetic algorithms. In *ACM SIGPLAN Proc. Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)* (July 1999), vol. 34, pp. 1–9.

18. Cytron, R., Lowry, A., and Zadeck, K. Code motion of control structures in high-level languages. In *Proc. ACM Symp. on Principles of Programming Languages* (St. Petersburg Beach, Florida, January 1986), ACM, pp. 70–85.

19. Cytron, R. K., and Ferrante, J. Efficiently computing nodes ϕ-nodes on-the-fly. *ACM Trans. Programming Languages and Systems 17*, 3 (May 1995), 487–506.

20. Cytron, R. K., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing the static single assignment form and the control dependence graph. *ACM Trans. Programming Languages and Systems 12*, 4 (October 1991), 451–490.

21. Debray, S. K., Evans, W., and Muth, R. Compiler techniques for code compression. Tech. Rep. TR99-07, University of Arizona, April 1999.

22. Debray, S. K., Evans, W., Muth, R., and Sutter, B. D. Compiler techniques for code compaction. *ACM Trans. Programming Languages and Systems 22*, 2 (March 2000), 378–415.

23. Fernandez, J.-C. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming 13* (1989), 219–236.

24. Fernandez, J.-C., Kerbrat, A., and Mounier, L. Symbolic equivalence checking. In *Proc. 5th Intl. Conf. on Computer Aided Verificartion (CAV'93)* (Heraklion, Greece, June 1993), C. Courcoubetis, Ed., vol. 697 of *Lecture Notes in Computer Science*, Springer-Verlag.

25. Fraser, C. W., Myers, E. W., and Wendt, A. L. Analyzing and compressing assembly code. In *Proc. ACM SIGPLAN Symp. Compiler Construction* (June 1984), vol. 19, ACM, pp. 117–121.

26. Garey, M., and Johnson, D. The complexity of near-optimal graph coloring. *J. ACM 23*, 1 (January 1976), 43–49.

27. Giegerich, R., and Kurtz, S. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica 19* (1997), 331–353.

28. Graf, S., and Loiseaux, C. A tool for symbolic program verification and abstraction. In *Proc. 5th Intl. Conf. on Computer Aided Verificartion (CAV'93)* (Heraklion, Greece, June 1993), C. Courcoubetis, Ed., vol. 697 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 71–84.

29. Hoffmann, C. M., and O'Donnell, M. J. Pattern matching in trees. *J. ACM 29*, 1 (January 1982), 68–95.

30. Illingworth, J., and Kittler, J. A survey of the Hough transform. *Computer Vision, Graphics and Image Processing 44* (1988), 87–116.

31. Jaggar, D. *ARM Architecture Reference Manual.* Prentice Hall, Cambridge, UK, 1996.

32. Kärkkäinen, J. Suffix cactus: A cross between suffix tree and suffix array. In *Proc. Annual Symp. on Combinatorial Pattern Matching (CPM'95)* (1995), vol. 937 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 191–204.

33. Keller, J., and Paige, R. Program derivation with verified transformations—a case study. *Comm. Pure App. Math. 48*, 9–10 (1995), 1053–1113.

34. Kirovski, D., Kin, J., and Mangione-Smith, W. H. Procedure based program compression. In *Proc. 30th Intl. Symp. on Microarchitecture (MICRO'30)* (December 1997).

35. Kissell, K. MIPS16: High-density MIPS for the embedded market. In *Proc. Conf. Real Time Systems (RTS'97)* (1997).

36. Knuth, D., Morris, J., and Pratt, V. Fast pattern matching in strings. *SIAM Jour. Computing 6* (June 1977), 323–350.

37. Kozuch, M., and Wolfe, A. Compression of embedded system programs. In *Proc. Intl. Conf. on Computer Design: VLSI in Computers and Processors* (October 1994), IEEE.

38. Lampson, B. W. Fast procedure calls. In *Proc. Symp. ACM SIGARCH Architectural Support for Prog. Langs and Operating Syst.* (March 1982), vol. 10, ACM, pp. 66–76.

39. Lefurgy, C., Bird, P., Chen, I.-C., and Mudge, T. Improving code density using compression techniques. Tech. Rep. CSE-TR-342-97, EECS Department, University of Michigan, Ann Arbor, MI, 1997.

40. Lekatsas, H., and Wolf, W. Code compression for embedded systems. In *Proc. ACM/IEEE Design Automation Conference (DAC'98)* (San Francisco, CA, 1998), ACM, pp. 516–521.

41. Manber, U., and Myers, G. Suffix arrays: A new method for on-line string searches. *SIAM Jour. Computing 22*, 5 (October 1993), 935–948.

42. Marks, B. Compilation to compact code. *IBM Jour. of Research and Development 22*, 6 (November 1980), 684–691.

43. McCreight, E. A space-economical suffix tree construction algorithm. *Jour. ACM 23*, 2 (April 1976), 262–272.

44. Milner, R. *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

45. Muchnick, S. S. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco, CA, 1997.

46. Muth, R. *ALTO: A Platform for Object Code Modification.* PhD thesis, Dept. Computer Science, University of Arizona, 1999.

47. Richards, M. Compact target code design for interpretation and just-in-time compilation.

48. Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Global value numbers and redundant computations. In *Proc. ACM SIGPLAN Conf. Principles of Prog. Langs (POPL)* (January 1988), vol. 10, ACM, pp. 12–27.

49. Runeson, J. Code compression through procedural abstraction before register allocation. Master's thesis, Computer Science Department, Uppsala University, 2000.
50. Spinellis, D. Declarative peephole optimization using string pattern matching. *ACM SIGPLAN Notices 34*, 2 (February 1999), 47–51.
51. Storer, J., and Szymanski, T. Data compression via textual substitution. *Jour. ACM 49*, 4 (October 1982), 928–951.
52. Sunday, D. A very fast substring search algorithm. *Comm. ACM 33*, 8 (August 1990), 132–142.
53. Sweany, P., and Beaty, S. Post-compaction register assignment in a retargetable compiler. In *Proc. 23rd Annual Workshop on Microprogramming and Microarchitecture* (November 1990), pp. 107–116.
54. Tip, F. A survey of program slicing techniques. *Jour. Programming Languages 2*, 3 (March–December 1995), 121–189.
55. Ukkonen, E. Constructing suffix trees on-line in linear time. In *Algorithms, Software, Architecture*, J. van Leeuwen, Ed., vol. 1. Elsevier, 1992, pp. 484–492.
56. van den Brand, M., Sellink, M., and Verhoef, C. Control flow normalization for COBOL/CICS legacy systems. Tech. Rep. P9714, University of Amsterdam, Programming Research Group, 1997.
57. Weiner, P. Linear pattern matching algorithms. In *Proc. 14th IEEE Annual Symp. on Switching and Automata Theory* (1973), IEEE, pp. 1–11.
58. Weiser, M. Program slicing. *IEEE Trans. Software Engineering SE-10*, 4 (July 1984), 352–357.
59. Welch, T. A. A technique for high-performance data compression. *IEEE Computer 17*, 6 (June 1984), 8–19.
60. Wiener, N. *Extrapolation, Interpolation, and Smoothing of Stationary Time Series.* MIT Press, 1949.
61. Wulf, W., Johnsson, R. K., Weinstock, C. B., Hobbs, S. O., and Geschke, C. M. *The Design of an Optimizing Compiler.* Elsevier Computer Science Library, 1975.
62. Zastre, M. J. Compacting object code via parameterized procedural abstraction. Master's thesis, Department of Computer Science, University of Victoria, 1995.
63. Ziv, J., and Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory IT-23*, 3 (May 1977), 337–343.
64. Ziv, J., and Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory IT-24*, 5 (September 1978), 530–536.