TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DIPLOMARBEIT

Optimizing Interpreters with Superinstructions

ausgeführt am Institut für Computersprachen

der Technischen Universität Wien

unter Anleitung von Ao.Univ.Prof. Dr. Anton Ertl

durch

Helmut Eller

Schönburgstraße 7/3
1040 Wien

01. Februar 2005

Unterschrift

Abstract

Superinstructions can be used to make virtual machine (VM) interpreters faster. A superinstruction is a combination of simpler VM instructions which can be executed faster than the corresponding sequence of simpler VM instructions, because the interpretative overhead, like instruction dispatch and argument fetching, is reduced. This work discusses the following three topics related to superinstructions. First, I present some heuristics to choose superinstructions. I evaluated the heuristics for Forth and Java programs. If the number of allowed superinstructions was very large, > 1000, then the heuristic which chooses all possible subsequences up to length 4 achieved the best results. If the number of allowed superinstructions was more limited, then a heuristic which favors short sequences and sequences which occur in many different programs and many different basic blocks performed better than the others. Second, I compare a simple greedy algorithm and an optimal algorithm to cover a program with superinstructions. I found that the greedy algorithm achieves almost optimal results. Finally, I compare superinstructions with non-sequential patterns. In my experiments, superinstructions performed slightly better than non-sequential patterns.

Kurzfassung

Superinstruktionen können helfen um Interpreter für virtuelle Maschinen (VM) schneller zu machen. Eine Superinstruktion ist eine Kombination von einfacheren VM-Instruktionen die schneller ausgeführt werden kann als die äquivalente Sequenz von einfachen VM-Instruktion, weil einige Kosten der Interpretation, wie Kosten für Instruction-Dispatch und Argument-Fetching, verringert werden. Die vorliegende Arbeit diskutiert im Wesentlichen folgende drei Themen. Erstens beschreibe ich einige Heuristiken für die Auswahl von Superinstruktionen. Ich habe die Heuristiken für Forth und Java Programme evaluiert. Wenn die Anzahl der erlaubten Superinstruktionen sehr groß (> 1000) war, dann lieferte die Heuristik, welche alle möglichen Teilsequenzen bis zur Länge 4 auswählte, die besten Ergebnisse. War die erlaubte Anzahl jedoch geringer, schnitt eine Heuristik, die kurze Sequenzen bevorzugte oder Sequenzen, die in vielen verschiedenen Programmen und Basic-Blocks vorkommen, am besten ab. Zweitens vergleiche ich ein einfaches Greedy-Verfahren und einen optimalen Algorithmus zur Überdeckung eines Programs mit Superinstruktionen. Ich habe festgestellt, dass der Greedy-Algorithmus fast optimale Ergebnisse erzielt. Drittens vergleiche ich Superinstruktionen mit nicht-sequentiellen Mustern. In meinen Experimenten haben Superinstruktionen etwas besser abgeschnitten als die nicht-sequentiellen Muster.

Contents

1 Introduction **3**

2 Background and Related Work **5**
2.1 VM Interpreters 5
2.2 Related Work 7
2.2.1 Pattern Classes 7
2.2.2 Selection Strategies 10
2.2.3 Implementation Variants 13
2.2.4 Code Compression 15

3 Selecting Superinstructions **16**
3.1 Terminology 16
3.2 Method 17
3.3 Heuristics 20
3.4 Comparison 22

4 Parsing Strategies **29**
4.1 Parsing 29
4.2 Comparison 31

5 Superoperators **33**
5.1 Tree Patterns 33
5.2 Pattern Generation 34
5.3 Selection and Parsing 36
5.4 Comparison with Superinstructions 36

6 Conclusions **40**

Chapter 1

Introduction

Virtual machine (VM) interpreters are a popular implementation technique for programming languages because they offer various advantages over other techniques, like native code compilers or hardware implementations. The main advantage is portability. VM interpreters can be implemented in C and are therefore relatively easy to port to different architectures. The object code for VM interpreters (e.g., Java-bytecode) is often portable and often more compact than native code. Moreover, interpreters are also relatively simple programs, especially compared to a native code compiler. Although VM interpreters don't achieve the execution efficiency of native code, they are more efficient than tree or text based interpreters, which execute the source program more or less directly, without a compilation step. In summary, VM interpreters offer an interesting tradeoff between implementation simplicity, portability, and execution efficiency.

VM interpreters are not as efficient as native code, because maintaining and updating the state of the virtual machine introduces some overhead. The interpretation overhead for a virtual instruction includes things like VM instruction dispatch, argument access, instruction pointer updates, stack pointer updates and other things.

One way to reduce the interpretation overhead is to add *superinstructions* to the VM. Superinstructions are combinations of multiple simpler VM instructions. Combining several instructions reduces the interpretation overhead, because a superinstruction performs the same work as several simple instructions, but the interpretation overhead for the superinstruction is only about the same as for one simple instruction.

In this work I try to answer the following questions:

- *Which superinstruction should we add to a VM?* We can't add all possible combinations because there are too many possibilities. Therefore we have select a subset of the possible combinations and, of course, we'd like to add the best set.

- *How much effort should we spend for parsing?* When the set of superinstructions is fixed, we still need to decide which members of the set should be used for a particular sequence of (simple) instructions. The process of covering a sequence of simple instructions with superinstructions is called *parsing*. We can use a simple greedy heuristic or a more complicated optimal parsing algorithm.

- *Should we use non-sequential patterns?* Superinstructions correspond to simple sequential patterns. Non-sequential patterns are sequences with 'holes'. If our programs contain sequential patterns frequently, then it is reasonable to assume that there are non-sequential patterns too.

The main conclusions presented in this work are as follows. First, a selection strategy for general superinstructions should favor short sequences which occur in many different places. Second, for the purpose of superinstructions, a greedy parsing algorithm performs essentially as well as an optimal parsing algorithm. Finally, tree patterns (a form of non-sequential patterns) don't seem to offer substantially more optimization potential than sequential patterns.

The rest of this document is organized as follows. The basic techniques for implementing interpreters are explained at the beginning. Then follows a review of important ideas and tricks described in the literature. Different heuristics for choosing a set of superinstructions are considered in Chapter 3. This chapter also presents the method I used for the experiments. Chapter 4 discusses the tradeoffs between greedy and optimal parsing. Superoperators, i.e., non-sequential patterns, are compared with superinstructions in Chapter 5.

Chapter 2

Background and Related Work

The first section in this chapter explains the basic techniques needed to implement VM interpreters in C. The second section discusses related work.

2.1 VM Interpreters

A virtual machine interpreter emulates a virtual machines in software. A typical virtual machine (VM) defines some registers, some kind of memory, one or more stacks, and a set of datatypes with associated instructions. The details are often influenced by the programming language for which the virtual machine is designed. The programs for the virtual machine are usually generated by a compiler which translates source programs to programs for the virtual machine. Some programming systems with VM interpreters are: GForth, the Hotspot Java system, Squeak Smalltalk, or the Erlang BEAM emulator.

The canonical VM interpreter uses an instruction format with a one byte opcode followed by zero or more bytes for constant operands. The execution of such an instruction proceeds as follows. The interpreter loads the opcode (`fetch`) and selects the corresponding action (`dispatch`). The arguments for the instruction, e.g., arguments on the VM stack or in VM registers, are copied to local variables. The interpreter performs then the actual computation and writes the result back to the stack or register. Finally, the instruction pointer is incremented and the computation continues with the next VM instruction. In C pseudo code:

```
char* ip;                     /* instruction pointer */
for (;;) {
  char opcode = *ip;          /* fetch */
  switch (opcode) {           /* decode */
    case ADD:
      ... = ... + ...         /* execute */
      ++ip;                   /* next */
      continue;
```

```
      case MUL:
      ...
    }
  }
```

This is an elegant implementation, but has some shortcomings. One problem is that the C switch statement performs an unneeded range check. The check is not needed because we know that the bytecode contains only valid opcodes. E.g., `gcc` is not smart enough to remove this check by itself[Ert93] and has no option or declaration to suppress the check.

A popular workaround to this problem are *first-class labels*. First-class labels aren't available in ANSI C but `gcc` supports them. In GNU C it is possible to access the value of labels. The value is a pointer to a code location inside a function and can be stored in any pointer variable. A special *computed goto* operator can then be applied to such values. The syntax for accessing the value of a label is `&&`*label* and computed gotos are written as: `goto*` *pointer*. In GNU C we could write the interpreter like this:

```
  char* ip;                    /* instruction pointer */
  static void* optab[] = {
       &&add, &&mul, ...
  }
 next:
  char opcode = *ip;           /* fetch */
  goto* optab[opcode];         /* decode */
 add:
  ... = ... + ...              /* execute */
  ++ip;                        /* next */
  goto next;
 mul:
      ...
```

As before, the program is represented as a sequence of bytes (`ip`). But this time the opcode is used to lookup the value of the corresponding label in a jumptable (`optab`) and control is transferred to that label with a computed goto. This is a bit faster than the switch based version because the lookup doesn't involve any range check.

According to Anton Ertl's Threaded Code page[Ert], the former implementation variant is called *switch threading* and the latter *token threading*. Threaded code was apparently invented by Charles Moore in 1970 and the first publication was written by Bell[Bel73]. The reader should also note that "threaded code, in its original meaning [Bel73], is one of the techniques for implementing virtual machine interpreters. Nowadays, at least the Forth community uses the term threading for almost any technique used for implementing Forth's virtual machine."[Ert]

Another threading variant worth to mention is *direct threading*. For direct threading we replace the opcodes in the intermediate code with the values of the corresponding labels. This new intermediate code requires usually more space because a label takes a full machine word and not only a single byte, but since the value of the label is now directly available from the intermediate code, we save the lookup in the jumptable:

```
  void** ip;                  /* instruction pointer */
 next:
  void* op = *ip;             /* fetch */
  goto* op;                   /* decode */
 add:
  ... = ... + ...             /* execute */
  ++ip;                       /* next */
  goto next;
 mul:
      ...
```

2.2 Related Work

Combining patterns of simpler instructions to new, more complex instructions is by no means a new idea. This section lists some important ideas and methods described in the literature.

2.2.1 Pattern Classes

Several approaches to build complexer operators from simple operators have been proposed. One important property of an approach is the used *pattern class*. The pattern class describes the possible structure and possibly other constrains of the allowed patterns. E.g., are simple pattern class is the set of sequences of VM instructions. Another example for a pattern class is the set of tree fragments (if our language is tree oriented). The pattern class also describes whether patterns can contain wildcards or parameters. We now look at some publications describing various pattern classes.

Supercombinators

Hughes' paper about *supercombinators*[Hug82] is often cited as the first published work about the topic. A combinator is a λ-calculus expression which contains no free variables and no λ abstractions and can therefore be implemented efficiently in a low level language (such as C or assembly). One approach to implement λ-calculus expressions is to translate the expression into a set of combinator definitions and a single expression consisting only of constants and applications of the combinators. It is possible to use only the so called S, K, and I combinators, but those combinators are relatively small and the resulting program would be relatively large. Supercombinators on the other hand are derived from the *maximal free expressions*, i.e., maximal sub-expressions of the λ-expression which do not depend on the bound variable. Supercombinators are larger, match the source program better, and are better suited for an efficient implementation than the rather artificial S, K, I triple.

Superoperators

Proebsting[Pro95] describes *superoperators* for a C interpreter. The unoptimized interpreter executes C programs and uses a linearized form of `lcc`'s[FH95] intermediate representation (IR). One virtual instruction corresponds to one node in the IR-tree; the arguments are kept on a stack. The interpreter is not written in C, but it is generated by a modified version of `lcc`. The author describes a technique to find patterns in the IR-trees and to generate an optimized version of the interpreter with new instructions corresponding to the patterns. The patterns are tree based, e.g.:

```
ADDP(INDIRP(*),CNSTI)
```

`ADDP`, `INDIRP`, and `CNSTI` are the names of nodes in the IR and `*` is a wildcard for an argument on the stack. The selection heuristic described in the paper selects program-specific superoperators with the goal to reduce the space requirements for the bytecode program.

The heuristic examines all parent/child pairs occurring in the IR-trees of a program. The pairs are then weighted according to the (estimated) cost and the most promising pair is added to set of superoperators. The added operator can then be part of another superoperator. The process is iterated until the number of bytecodes is exhausted or no new superoperator can be found. The author reports a code compression ratio (i.e., reduction of text-segment size) of 78%-81% compared to the unoptimized interpreter; execution time was reduced by a factor of 2.2-3.1.

In a later (unpublished) paper[FP95] the authors present some improvements. First, they changed the base instruction set a bit to make it more suitable for superoperator generation. Specialized instructions to load 8, 16, and 32 bit constants were added and some instructions (which were actually `nop`s) were removed. Another improvement was the possibility to specialize instructions with small immediate arguments. E.g., `CNSTI[0]`, the instruction to push the constant 0, results in two patterns: `CONSTI[#]` and `CONSTI[0]`. The first variant reads an (arbitrary) constant from the bytecode, i.e `#` is a wildcard for a constant. The second variant is an instruction with a hardwired zero. The authors report a compression ratio of about 50%.

Maarten Wegdam[Weg96] describes some more improvements on Proebsting's and Fraser's work. He modified the selection heuristic to make it faster. (The unmodified version was apparently too slow. He reports compilation times in the order of a few days.) First he removes candidate patterns with too little space saving; a factor for 'too little' was determined empirically. Moreover, he reduces the number of iterations for the selection algorithm. In the original heuristic the weights of the candidate patterns were recomputed whenever a pattern was selected as superoperator because adding a superoperator can change the weights of the remaining candidates. In practice, this change is often neglectable. Wegdam selects the top 10% of the candidates instead of a single pattern before recomputing the weights. This way, the algorithm converges faster. To improve the quality of patterns, he allowed more general patterns. In particular, he allows multiple wildcards in arbitrary positions in a fragment. The original algorithm supported only a single '*' wildcard as the child of a root node. He also does some postprocessing to reuse unused opcodes. A pattern can become unused when it overlaps with another pattern. He reports code size reduction from 50%-70%. Execution speed of the compressed code was not a priority.

Hoogerbrugge et al.[HATW99] uses superoperators to reduce the space requirements of rarely executed code. (See also Subsection 2.2.2.)

Superinstructions

Gregg et al.[GEW01] present detailed information about instruction frequencies and other features of six large and four small Forth programs. An instrumented version of GForth was used to obtain the results. The authors report that Forth programs spend a significant amount of their execution time in user code, not in system code. Furthermore, Forth programs contain (statically) many call and return instructions. Basic blocks are very short: most of them only one instruction long, and relatively few (< 40%) are longer then three instructions. The instructions with the highest static frequency are `call` and `;s` (together about 25%). Also very common are `lit` and `@`. The dynamic and static frequencies are surprisingly similar: the dynamic frequency of calls and returns is a bit lower than their static frequency and some of the longer basic blocks are executed frequently, but most of the frequently occurring instructions have also a high dynamic frequency. The paper includes lists of the most common static and dynamic sequences. The authors conclude that it might be better to use the static frequencies than the dynamic frequencies as a basis for superinstructions selection, because the dynamic frequencies tend to be too program specific. Inlining is suggested to remove the overhead for the frequent calls and to increase the optimization potential for superinstructions.

Specialization

In many systems, VM instructions can have immediate arguments. Those arguments are typically constants. If a VM instruction occurs frequently with the same immediate argument, it is often beneficial to introduce a *specialized* version for that VM instruction/argument combination. Executing the specialized VM instruction is usually faster because no time is needed to decode the immediate argument. We have already seen one way to specialize operators in Fraser's work[FP95].

Nässén [NÖ1, NCS01] reports the result of optimizing the SICStus Prolog instruction set with combinations and specializations of existing instructions. The initial emulator was instrumented to record static and dynamic frequencies of instructions and instruction pairs in a training-set of programs. The author created various extended instruction sets based on the obtained profile information. (No automatic method for the selection of instruction or the generation of the extended emulators is presented. It is also not described how the profile results from different benchmark were combined.) A 10% speedup and 15% bytecode size reduction is reported. (The benchmark programs were the same as the training-set.) Specializing instructions was less beneficial on the SPARC than on the x86, apparently because the SPARC version of `gcc` performed a too aggressive space optimization.

Grammars

Evans and Fraser present a somewhat different approach in their 2001 paper[EF01]. The basic idea is to view bytecode programs as sentences generated by a grammar. If we use the grammar to parse the program, we get a tree (or a forest) of a certain size. The size decreases if we extend the grammar with new rules to allow shorter derivations.

Terminal symbols in the grammar correspond to bytecode instructions and constants. Non-terminal symbols are used to produce sequences of instructions and are used to group instructions by stack effect[1]. This grammar is then used to parse a training-set of programs. The result is a forest of parse trees. Each node in the parse trees corresponds to one rule in the grammar. Frequent rule pairs, i.e., frequent edges in the parse trees, are identified and used to create new rules. A pair of rules $A \rightarrow xBy$ and $B \rightarrow z$ can be combined to an new rule $A \rightarrow xzy$. The authors call this process *inlining*. The authors use a similar greedy algorithm as in their earlier work to select the set of new rules. The result is an extended grammar.

To compress a program it is parsed with the extended grammar and the shortest derivation is emitted. A compressed program is a sequence of rule numbers for the derivation.

The interpreter for compressed programs works a bit differently than the initial interpreter. The extended interpreter executes rules from the extended grammar. The interpreter uses the rule number to lookup the rule in a table and processes the symbols on the right-hand side of the rule sequentially. Terminal symbols correspond to instructions from the initial instruction set and the extended interpreter executes them the same way. Non-terminal symbols are executed by invoking the interpreter recursively, i.e., the extended interpreter requires an extra stack (or another mechanism) for recursive calls.

As the extended grammar is ambiguous, the authors used a version of Earley's algorithm. Earley's algorithm parses non-deterministic context free grammars in cubic time. An advantage of this grammar based approach compared to the superoperator approach is the ability to find patterns in non-adjacent code fragments, i.e., code fragments that are not near each other in the intermediate representation of the compiler.

2.2.2 Selection Strategies

In an implementation, the number of superinstructions is usually limited. Various *selection strategies* have been proposed to select good set of superinstructions.

[1] The exact grouping is probably not very important as long as all valid programs can be derived.

The problem of select an optimal set of (static) superinstructions is similar the problem of selecting an optimal dictionary for static dictionary based compression[BCW90].

As we have seen [Pro95, FP95, Weg96], a popular method is to create a set of pairs, sort the pairs in order their estimated cost (size or execution time), select the top N pairs, and repeat the process until the set of selected superinstructions no longer changes. The initial pairs are adjacent operators occurring in important programs. In the later steps, a pair can also contain superinstructions from the previous step.

Selection Strategies for Forth

Gregg and Waldron[GW02] present and evaluate various heuristics to select superinstructions for Forth programs. Again, GForth was modified to record the static and dynamic frequencies of basic blocks in five programs. The basic blocks together with the frequencies were then used to select superinstructions. The presented selection strategies are:

static Choose sequences in order of the static frequency.

dynamic Choose sequences in order of the dynamic frequency.

length Rank by the product of (static resp. dynamic) frequency and length of the sequence.

normalize Divide the static (dynamic) frequency by the number of static (dynamic) instructions in the program before ranking.

hybrid Rank by the sum of the static and dynamic frequency

rewrite Recompute the ranking each time a sequence is chosen.

These strategies (and some combinations) were evaluated for different numbers of superinstructions (8–2048). The number of saved dispatches was used as measure for the quality of the selected superinstructions. Superinstructions were chosen from four of five programs; the fitness was evaluated for the fifth. The authors report that static frequency is a usually a better utility measure than dynamic frequency; probably because dynamic frequencies are strongly influenced by a few loops and therefore not as good in other programs. The HYBRID heuristic performed best, but only sightly better than STATIC. Normalizing frequencies showed only minor benefits and only if the number of superinstructions was large. Weightening sequences by length did not result in better superinstructions. Shorter sequences seem to be preferable. The rewriting heuristic gave mixed results, but seems to favor longer sequences too much.

Selection Strategies for Java

Casey et.at.[CGEN03] used vmgen[EGKP02] and superinstructions to create an optimized version of CVM (Sun's VM for embedded processors). The effect of two superinstruction selection heuristics are reported. The first heuristic selects the sequences with the highest static frequencies in a set of programs. The second heuristic selects superinstructions with respect to a single program. Those sequences with the highest dynamic frequency in the program are chosen. The second variant is probably not a good way to choose general purpose superinstructions, but can be used as a indicator for the quality of the first method. The results are reported for VMs with different numbers of superinstructions (8–1024) when executing SPECjvm98 programs. The speedup for the static method ranged from -1%–60%. As expected, the dynamic method performed better: the speedups ranged from 5%–90%. The authors say that there is still potential for improvements, e.g., by implementing 'quickable' instructions with the trick described by Gagnon et al.[GH03] or by adding superinstructions which can cross basic block boundaries. They also compare their base interpreter (without superinstructions) to Sun's Hotspot. Hotspot in interpreter-mode is 24% faster and an order of magnitude faster in mixed-mode. The authors note that the difference may be due to Hotspots better tuned runtime system (GC etc.) and better optimized class libraries.

Updating Frequencies exactly

O'Donoghue and Power[OP04] formally present a sequence analysis algorithm and show the results of applying it to Java programs. Again, the frequencies for basic blocks in Caffeine benchmark were recorded and used as basis for the selection algorithm. The authors used the familiar iterative algorithm. They select the candidate based on the product of frequency and length of the basic block, recompute the frequencies of the remaining sequences, and reiterate until the number of superinstructions is exhausted. The paper presents an exact algorithm for updating the frequencies of overlapping and subsumed sequences. Various instruction sets with differing maximum length were evaluated. The machine with superinstructions of length up to four performed best.

Covering Matrix

Liao et.al[LDK99] describe two methods to reduce the code size in an embedded processor. Both methods use a dictionary of frequently used instruction sequences. The first method substitutes occurrences of those sequences with *mini-subroutine calls* (i.e., simplified calls which don't pass arguments) to entries in the dictionary. The instructions of a dictionary entry are executed like regular instructions but the entry is terminated with a `RET` instruction to pass the control back to the normal instruction flow. The second method uses some special hardware: a new instruction, `CALLD` i j. `CALLD` executes j instructions of the ith dictionary entry. This new instruction doesn't require an extra `RET` and it is also possible to execute only a subsequence of an entry. The candidate sequences for the dictionary are generated with a simple algorithm: all subsequences of basic blocks which occur in more than one place in the source are considered. The actual entries for the dictionary and the positions where the call instructions should be inserted are then computed in a single pass by solving a set covering problem. The authors represent the covering problem as a *covering matrix*. The entries in the matrix are boolean values and an entry (i, j) is marked 1 if column j (corresponding to a dictionary candidate) covers row i (corresponding to a position in the program). A heuristic covering algorithm is used to find an assignment for the columns and rows with minimal cost.

Optimal Tree Matching

Hoogerbrugge et al.[HATW99] use an VM interpreter with superoperators to reduce the size of programs for a VLIW processor. Time-critical parts of the application code are compiled to native VLIW instructions and non-time-critical parts to VM code. Their system uses a tree-based intermediate representation to find superinstructions: possible superinstructions are subtrees in the intermediate representation. The maximal size of the subtrees 8 to keep the selection algorithm efficient. Selection of superinstructions proceeds in two steps. The first step ranks all possible subtrees by the product of their static occurrence count and size. The first $2N$ candidates are used for the second step. N is the number of desired superinstructions. The second step performs optimal tree matching to cover the intermediate representation of the program. The patterns used consist of the base instructions and the $2N$ candidates from the first step. All patterns have the same cost. The N most frequently used patterns in the cover are the selected superinstructions. The motivation for this two step scheme is as follows. According to the authors, the occurrence count is not a good measure for selecting patterns, but a reasonable indicator for the initial selection. The second step ranks those patterns lower which are covered by larger patterns: If a tree T_1 is almost always a subtree of another tree T_2, then the second step can select T_2, but it is not necessary to include T_1.

2.2.3 Implementation Variants

This section describes some implementation tricks which are often used in combination with superinstructions.

Pipelined Interpreters

Hoogerbrugge et al.[HATW99] describe an interesting way to implement an interpreter: they use a software pipeline. Much like instructions in a pipelined processor, virtual instructions are executed in three stages FETCH, DECODE, and JUMP/EXEC. The various stages of consecutive virtual instructions are executed in parallel. This kind of pipelining gives the compiler more freedom to schedule loads early and generally more possibility to use the native VLIW instructions more efficiently. The authors report that this scheme reduces the critical path of a block from ten cycles to four cycles in most cases. Various complications regarding variable width instructions (introduced by superinstuctions) must be mastered, though. And like the pipeline in a processor, the pipeline for the interpreter must be flushed on jumps. According to the authors, a taken jump requires ten cycles more than a not taken jump.

Dynamic Superinstructions

A surprisingly simple and effective method to create superinstructions at runtime is described by Piumarta et al.[PR98] They start out with a direct threaded interpreter. The basic idea is to create superinstructions by copying the (native) machine code for the virtual instructions to a new memory area. The sequence of virtual instructions (represented as sequence of pointers) can then be replaced with a pointer to the new memory block. E.g., to create a superinstruction for the following add instruction

```
add_start:
 stack[1] = stack[0] + stack[1];
 --stack;
add_end:
 ++ip;
 goto next;
mul_start:
   ...
```

the code between `add_start` and `add_end` is copied to the new area. It is important that the code fragments to copy are relocatable. E.g., on IA32 this means that such a fragment should not contain calls to C functions because calls are typically implemented with jumps to absolute addresses. The authors describe a scheme to rewrite bytecode to direct threaded code with such superinstructions. Their scheme only concatenates the instructions in a basic block, i.e. the generated superinstructions begin with a branch target, end with a jump, and contain no other control flow operators. Superinstructions are cached and reused if the same sequence occurs in multiple locations. The reported speedups range from 1 to 2.

Branch Prediction

Ertl and Gregg[EG03] investigate two methods to improve branch prediction in interpreters. The first method, *instruction replication*, creates copies of the machine code for a VM instructions. This copying improves the prediction accuracy of branch target buffers (BTB) because the indirect branch at the end of a copied VM instruction jumps each time to the same target (the beginning of the next VM instruction). In contrast, if the code is not copied, i.e. shared, the indirect branch jumps often to different targets because the next VM instruction is not fixed. The copied branch instruction gets its own entry in the BTB and hence better prediction accuracy. The second method uses superinstructions. Superinstructions reduce the number of executed dispatches and native instruction, but also mispredictions. The reason is again the better utilization of the BTB. Consider a loop with 3 instructions `A B A` and the equivalent loop with a superinstruction `A_B A`. In the first case, the branch target for the `A` instruction alternates on each iteration and is therefore difficult to predict. In the loop with the superinstruction, the target is the same on each iteration and will be predicted correctly. Replicas and superinstruction can be created at interpreter generation time or at runtime. The authors evaluated various combinations of the different approaches and report speedups of up to 2.08 for a variant with 35 static superinstructions and 365 static replicas and a speedup of up to 3.17 for a variant with 400 static superinstructions and dynamic replication. Replication increases code size and also I-cache misses. However, the authors report that the improved prediction accuracy more than compensates the increased I-cache misses.

2.2.4 Code Compression

Related topics are also code compression and text compression. Superinstructions can help to reduce the size of programs but the most compact encodings can usually not be executed quickly, therefore I don't pursue this topic further. Árpád Beszédes et al.[ÁBFG+03] survey various code size compaction techniques. "Textcompression"[BCW90] is a classic book.

Chapter 3

Selecting Superinstructions

In this chapter I describe some heuristics I used to select a good set of superinstructions. The number of superinstructions we can add to a VM is limited in practice. On reason for the limit is that certain implementation techniques allow only a small number of operators, e.g., an interpreter with 8-bit opcodes allows only 256 operators. Another limiting factor is the time and memory needed to compile the VM. The main loop of the interpreter is usually implemented as one big function and is compiled at high optimization levels. Most compilers require a lot of time and memory to compile such functions, e.g., `gcc-2.95` needs more than 200 MB for an interpreter loop with 500 virtual instructions. Since the number is limited, we would like to add only the best superinstructions.

The following section defines some terminology and the problem more precisely. Then follows a description of the methodology I used to compare different selection heuristics. The actual heuristics are described afterwards and the last section in this chapter presents the results of the comparison.

3.1 Terminology

Finding a good set of superinstructions is an *optimization problem* with many parameters. We can optimize for speed by choosing the most frequently executed patterns or we can reduce code size by choosing the most frequently occurring patterns. The set of superinstructions can be selected for one particular program or for a range of applications. My primary goal was to find general purpose patterns to improve execution speed.

The general problem of finding good patterns can be formulated[BJEG95] as follows: given a *training-set* of programs and a *rating-function*, find a set of patterns with maximal rating. The set of patterns is often called the *dictionary*.

Proebsting[Pro95] notes that choosing an optimal set of superinstructions for space reduction is NP-complete and that optimizing for execution time has similar complexity. It is therefore reasonable to approach the problem with *heuristics*.

The training-set is usually a collection of 'representative' programs. A selection heuristic takes the training-set and selects a set of patterns found in those programs. The rating-function evaluates the patterns according to some metric. A common approach to evaluate the patterns is to measure the reduction of execution time some benchmark programs compared for a VM with superinstructions and for VM without superinstructions. The benchmark programs for the rating-function can be the same as the programs in the training-set. But if we are interested in generally useful patterns, it is probably better to use different programs for the training-set and the rating-function.

Approaches to find good patterns have often two steps[BJEG95]: the first step generates a set of *candidate patterns* and the second step uses a *heuristic* to select those candidates with high rating. Candidate patterns can be in found the training-set, e.g., all subsequences of basic blocks are candidates. Alternatively we could enumerate all valid patterns, e.g., all combinations of instructions up to a certain size. The former method is said to be *sequence-driven* and the latter *pattern-driven*. A pattern-driven method is theoretically able to find patterns not present in the trainings-set. But enumerating all patterns is usually very complex (often exponential in the maximal pattern size), and in our setting it is unlikely that a pattern not in the training-set has a high rating. Thus, I only considered sequence-driven methods.

Patterns can be categorized by *pattern class*. Two important classes for patterns in programs are sequence patterns, i.e. *superinstrunctions* and tree patterns, i.e., *superoperators*.

In a VM with (static) superinstructions we have usually several possibilities to replace sequences of simple instructions with superinstructions. This decision which of the possibilities should be chosen is called *parsing* in analogy to compression programs which parse the input with respect to a dictionary and to emit a shorter sequence. Also similar parsing algorithms as in compressors can be used.

3.2 Method

The approach I used to add superinstructions to a VM is as follows. First, generate an instrumented VM to record execution traces. Then execute the programs from the training-set to obtain actual trace files. The trace files serve as input to a selection heuristic which emits the set of superinstruction patterns. The superinstructions can then be added to the VM.

More concretely, I instrumented the GForth[Ert93] engine and also a version of GIJ. GForth is a product-quality Forth implementation and GIJ is the Java interpreter from the GNU Compiler Collection. GForth's engine is constructed with `vmgen`. As `vmgen` has already support to generate the needed instrumentation code, creating execution traces and an extended VM was relatively easy for GForth. Modifying GIJ required more manual work because the interpreter is not generated by a tool.

The Forth benchmark programs I used are listed in Table 3.1. The line count was determined with `egrep -Ehsv '^[ \\t]*(\\.*)?*$' | wc`, i.e., lines containing only whitespace or a comment were excluded. Many of these programs are, unfortunately, rather small and test only very specific properties.

The Java benchmark programs were taken from the SPECjvm98 suite. As the modified GIJ version did not support multithreading, only the single threaded programs were used: `check`, `compress`, `jess`, `db`, `javac`, `mpegaudio`, and `jack`.

Program	Lines	Instr. s.	BBL d.	BBL s.	Description
brainless	3088	11361	3.98	3.05	chess
brew	13695	6066	3.58	2.73	evol. programming
pentomino	184	7953	7.16	5.74	pentomino puzzle
siev	16	2327	3.98	3.10	prime numbers
bubble	48	2428	3.80	3.04	bubble sort
matrix	35	2285	16.41	3.17	matrix multiplication
fib	10	2080	2.63	3.08	fibonacci
regexmatch	92	4116	3.20	2.63	regex matching
lists	74	2914	3.34	2.97	list processing
spellcheck	24	2276	2.61	2.95	hashing, line I/O
heapsort	47	2963	5.73	3.06	in-place heap sort
check		2381	3.64	2.64	sanity check
compress		5088	3.39	3.75	text compression
jess		11689	2.35	2.56	expert system
db		4545	2.75	3.59	in memory DB
javac		10984	2.21	2.80	compiler
mpegaudio		35018	6.16	10.19	audio decompression
jack		18856	2.33	2.96	parser generator

Table 3.1: Forth and Java Benchmarks. Instr. s. is the number of static VM instructions, BBL d. the average dynamic basic block length, and BBL s. is the average static basic block length.

In Table 3.1, the static instruction count and static basic block length include only instructions which are executed at least once. I.e., the counts do not include the code in the image or libraries which is loaded but never executed.

The program to select superinstructions reads one or more trace files and emits the superinstructions in a format suitable for `vmgen`. I implemented some different heuristics which are described later.

Initially I tried to compare the different selection heuristics by running benchmarks with the optimized VM. But this turned out be a rather slow and difficult process. First, compiling the optimized VM, especially with large number of superinstructions, is very slow. Second, running the benchmarks takes also a long time. Even more so, if the benchmarks are executed multiple times to reduce noise factors like OS load. Another problem with running benchmarks was the parsing strategy. Current versions of GForth include an optimal parser, but at the time when I did my first experiments—it was before GForth-`0.6` was released—there was only a simple greedy parser. I had to implement my own optimal parser. I modified the compiler to use the optimal the parsing strategy. Eventually the compiler did what I wanted, but it was quite difficult as I ran into bootstrapping issues.

To avoid all these problems, I decided to run simulations instead of benchmarks on the real VM. The simulator takes as input a trace and an instruction set optionally with superinstructions. A trace is a list of basic blocks with the corresponding execution counts. The trace is then parsed, i.e., each original basic block is covered with instruction from the, possibly extended, instruction set. If we multiply the length of a basic block with its execution count, we get the number of executed virtual instructions. Since basic blocks with superinstructions are shorter than the non-optimized blocks, the number of executed VM instructions is smaller. I used the number of executed VM instructions as the primary measure for the quality of a selection heuristic.

Of course, a simulation like this ignores many details, e.g., cache issues. Moreover, the simulation assumes that each instruction takes the same amount of time to execute. However, I think the use of simulations is justified because compiling the VMs and running the benchmarks for Table3.4 requires more then 6 hours. Experimenting with slightly modified heuristics and parameter sets would take several days. Also after having waited a half day, I sometimes discovered that I had made a mistake at the beginning; the result was useless and the time was wasted. Compare this with the most complex simulation: Producing Table 5.3 took only 5 minutes. The simulation produces results much quicker and more variants and be evaluated.

To validate the simulation results, I selected some of the simulated instruction sets and compared the results of the simulation with the result of running the benchmarks on a real VM with the same instruction set.

If we would like to train a heuristic for a set of different programs, we need to combine the trace files. This poses a certain problem because the execution counts originating from different programs are not directly comparable. We need a way to normalize them. Let the simulated time $t_s(b,p)$ of a basic block b with respect to a program p be defined as the product of its execution count and its length, i.e.,

$$t_s(b,p) = count(b,p) \cdot length(b).$$

Intuitively this is the amount of time spent in this block if each instruction takes the same amount time to execute. Now define the relative time $t_r(b,p)$ of a basic block b in program p as the quotient of the time spent in the basic block and the time needed to complete the entire program. More formally:

$$t_r(b,p) = \frac{t_s(b,p)}{\sum_{b \in p} t_s(b,p)}.$$

A relative time value 0.3 means that the program spends 30% of the overall time in that basic block. The combined relative time $t_c(b,P)$ of a basic block b in a set of programs P is the quotient of the sum of all its relative times and the number of programs. Formally:

$$t_c(b,P) = \frac{1}{|P|} \sum_{p \in P} t_r(b,p).$$

Intuitively this is the percentage of time we would spend in the particular basic block if all programs would execute same number of instructions.

3.3 Heuristics

All heuristics presented in this section use sequences as pattern class. E.g., given the instructions `a`, `b`, and `c` then a possible pattern is `a_c_b`. All elements of in patterns are known, i.e., we don't allow wildcards in the middle of patterns.

I used the basic blocks from the trace data to generate candidate patterns. The set of candidate patterns consisted of all the subsequences of basic blocks. To keep the number of candidates down to a tractable number, I only considered subsequences shorter than 18 instructions. Each candidate pattern has an associated frequency. The frequency associated with a pattern is the frequency of its basic block or if the pattern occurs in different basic blocks, then the associated frequency is the sum of the frequencies of the corresponding basic blocks.

Full

This heuristic is relatively simple: The list of candidate patterns is sorted by decreasing execution and by increasing pattern length frequency, i.e., patterns with higher frequency are preferred and if two patterns have the same frequency, then the shorter one is chosen. The initial elements of the sorted list are the selected patterns.

This strategy is called FULL because all subsequences of a selected pattern are selected too. The above algorithm includes all subsequences because the list is sorted by frequency a subsequence of a pattern has equal or higher frequency than the full pattern.

A nice property of FULL is that a greedy parser will produce an optimal parse[BCW90].

A potential problem of this heuristic are underutilized subsequences. For instance, assume a program contains a certain basic block of length 5 very often and assume the subsequences of that basic block don't occur in other basic blocks. In this situation, the subsequences of length 4, 3, and 2 will be unused. However, the results of the simulation suggest that the problem occurs rarely in practice.

Static Dictionary

I adopted this strategy from a book about text compression[BCW90]. Certain compression schemes work with a so called 'static dictionary'. A dictionary for compression purposes contains frequently used sequences of symbols and the compressed text contains references to the dictionary entries. Choosing a set of superinstructions and creating a static dictionary are in many ways similar problems.

The STATICDICTIONARY starts like FULL, but then tries to fix the problem with underutilized subsequences. The candidate patterns are kept in a sorted list, the *candidate list*, and the selected patterns in a hash table of fixed size, the *dictionary*.

At the beginning, we move patterns of length 2 from the candidate list to the dictionary until we encounter a pattern of length 3. Before we add the pattern of length 3 to the dictionary, we reduce the frequencies of its subsequences by a certain factor. We also check whether the frequency of a pattern in the dictionary drops below the frequency of the first pattern in the candidate list. If so, we remove the pattern from the dictionary and re-insert it in the candidate list. This process is repeated for patterns of length 4 then 5 and so on until the dictionary is full. A pattern can move multiple times from the candidate list to the dictionary and vice versa.

The factor by which a the frequencies of subsequences should be reduced must be determined empirically.

This algorithm was able to find a very good set of superinstructions for a particular training-set, but performed poorly when the programs used by the rating function were disjoint from the training-set.

Static Count

This heuristic first sorts the list of candidate patterns by execution frequency and then applies some filters on the list. The filters remove candidates which don't seem to be valuable:

- Patterns which don't occur in different programs are removed. The number of different programs is a parameter to the heuristic. The idea is that a pattern is more general if it occurs in different benchmarks and will more likely be applicable in programs not present in the training-set.
- Patterns which don't occur in different basic blocks are removed. This is similar to the previous filter, but is also applicable if the training-set is a single program.
- If two sequences have almost the same frequency and one is a subsequence of the other, we only keep the longer. The idea is that two such patterns are probably derived from the same basic block and one of them is almost certainly not needed if the other is present. The range in which two frequencies are considered equal is a parameter to the heuristic.

Other Heuristics

The question was raised why I did not try other heuristics. The heuristics described above are the best I was able to find. I tried different parameter settings and variations of the described methods. For StaticCount I also used parameters which effectively disable the associated filter. E.g., setting the number of programs in which a pattern must occur to zero, disables the effect of the filter. For the filter which looks at patterns of about the same dynamic frequency and removes the shorter pattern, I tried to remove the longer instead of the shorter. None of the variations performed better.

Possible ideas which I haven't explored include: giving basic blocks originating from large programs more weight than basic blocks from small programs, in the hope that large programs contain more likely pattern which are useful for other programs.

3.4 Comparison

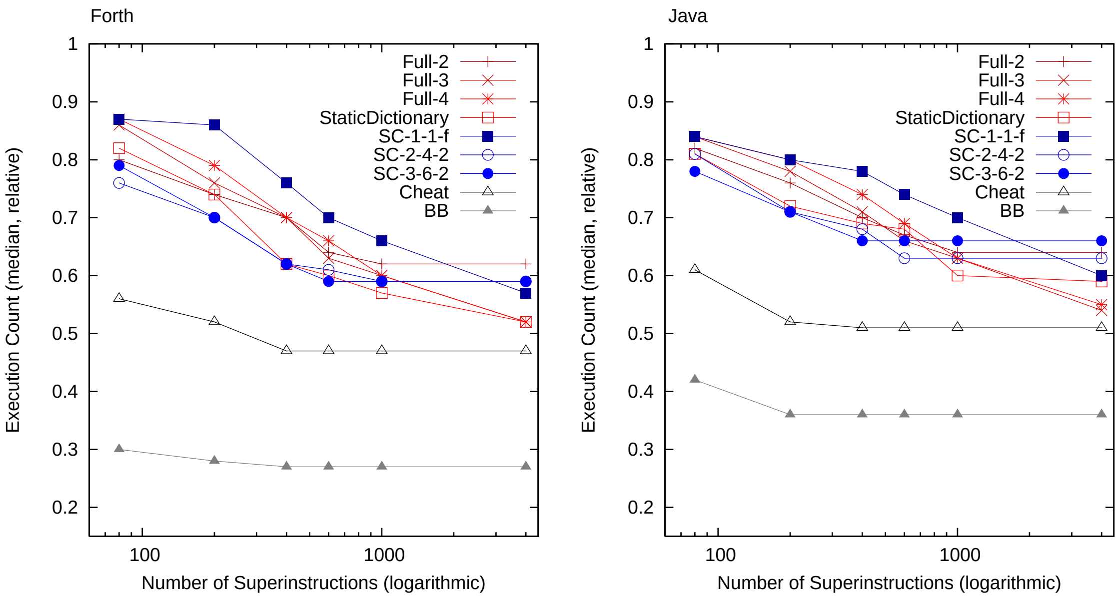

Figure 3.1: Selection Heuristics (Median)

The Figure 3.1 shows the results of comparing the different heuristics. The y-axis represents the median of the relative instruction counts (described below) and the y-axis represents the number of superinstructions in logarithmic scale. The data for Figure 3.1 is listed in Table 3.3. The first column in names the heuristic with parameters. The other columns contain the speedups for different limits on the number of superinstructions. The Figures 3.2, 3.3, and 3.4 show the results for the each benchmark separately.

The numbers are obtained as follows. First, one program is chosen as the *benchmark*. The remaining programs form the training-set. The traces of the programs in the training-set serve as input to the selection heuristic and the heuristic selects a set of patterns. The patterns are then evaluated for the other program, the benchmark. The evaluation function computes ratios of the number of VM instructions needed to execute the benchmark for an interpreter with superinstructions and for an interpreter without superinstructions. This process is repeated for all combinations of benchmark program and training-set and we get one ratio for benchmark/training-set combination. One entry in the table represents the median of those ratios (for a specific maximal number of superinstructions and selection heuristic). E.g., the entry 80 means, that the number of executed VM instructions in the optimized interpreter is the median

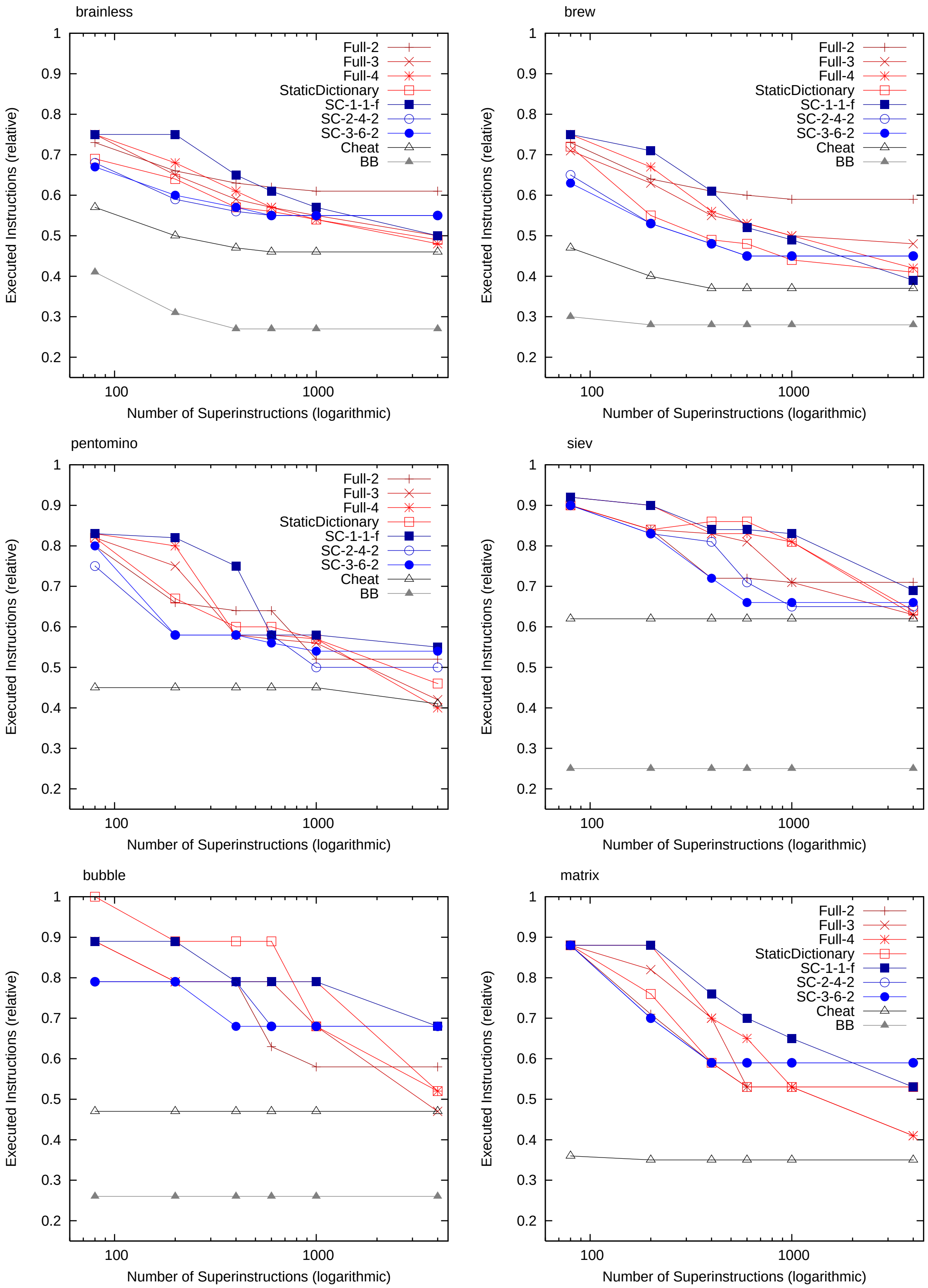

Figure 3.2: Graphs for individual Benchmarks

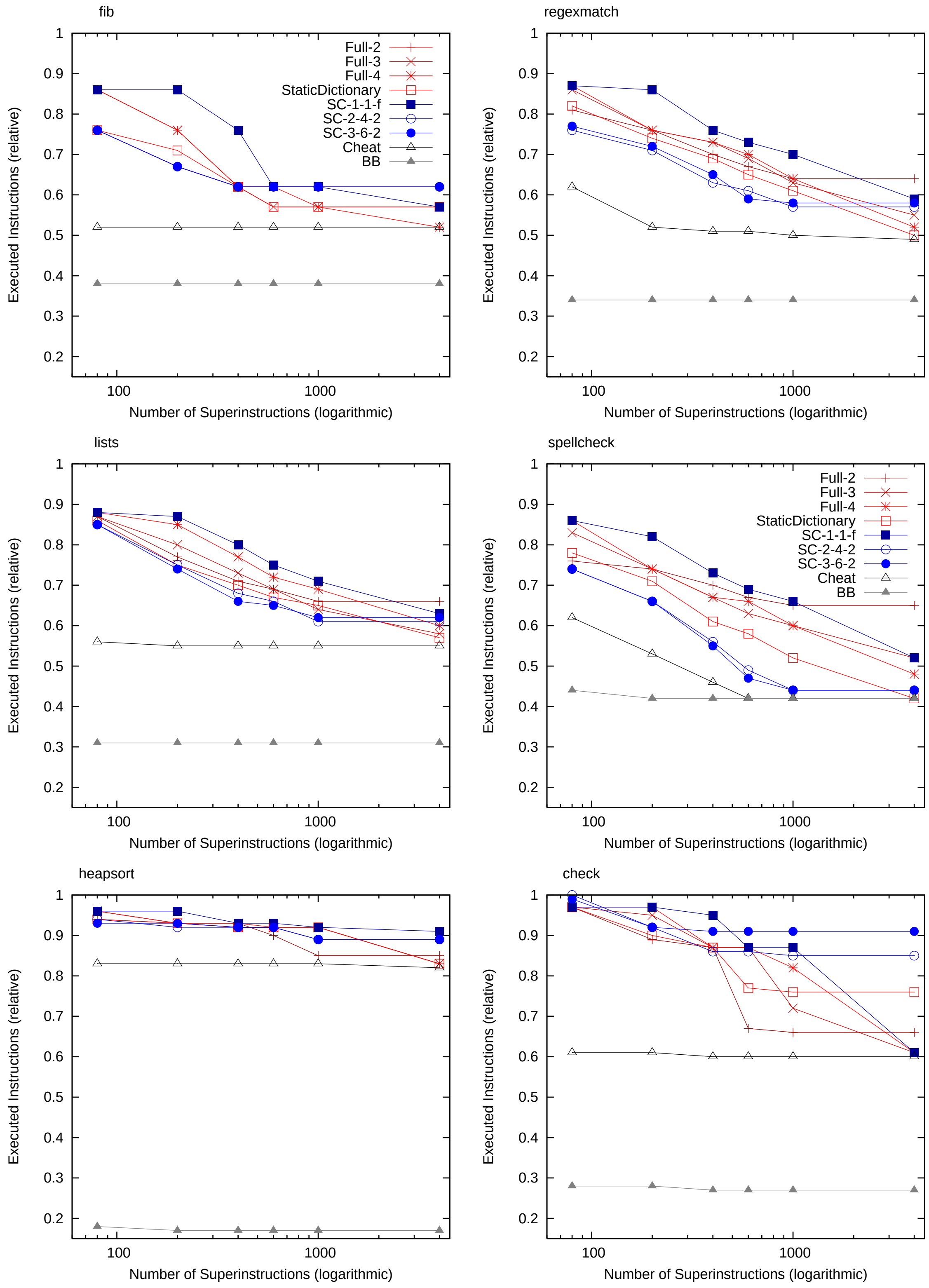

Figure 3.3: Graphs for individual Benchmarks

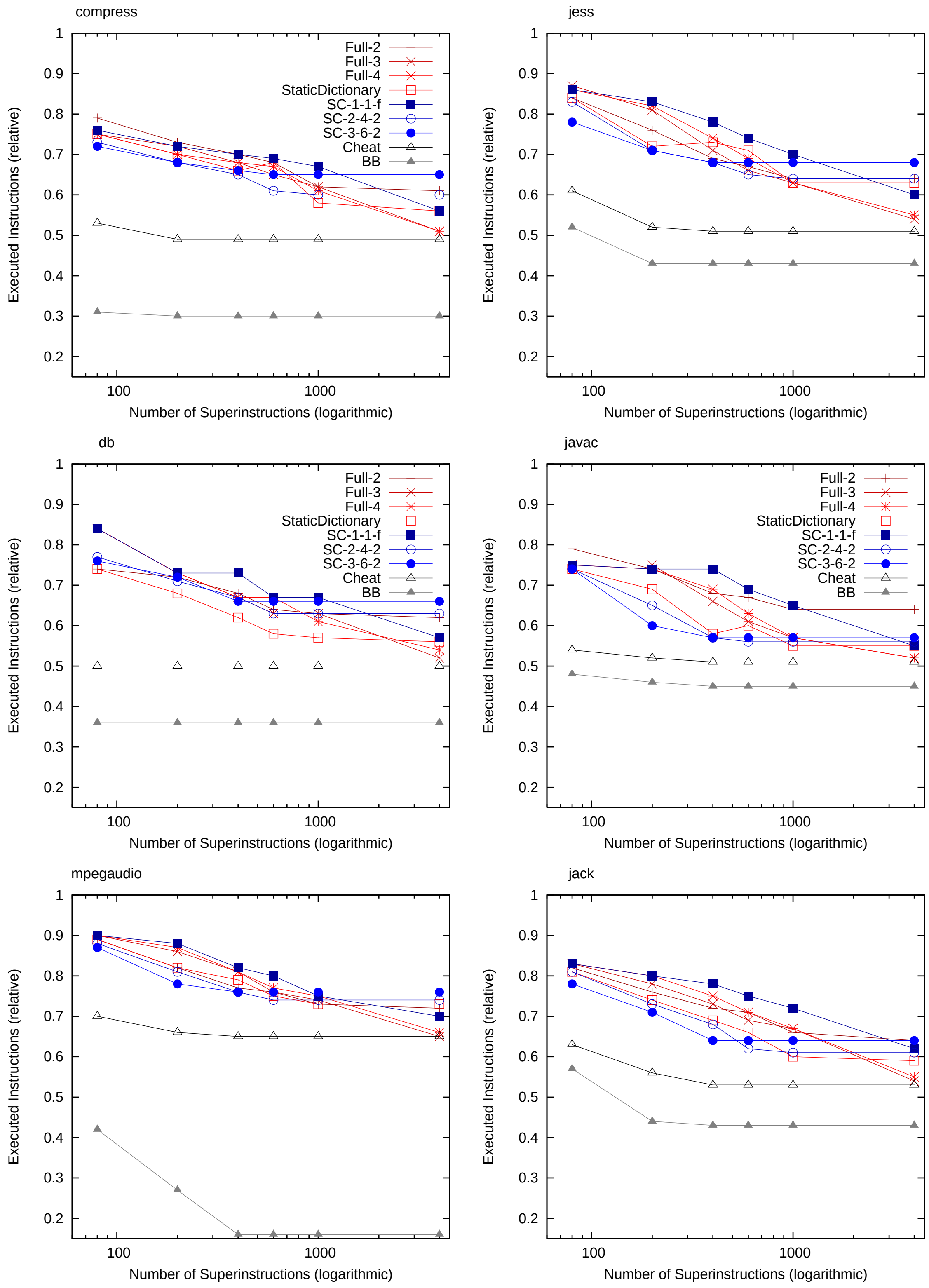

Figure 3.4: Graphs for Benchmarks

of the number needed by the unoptimized interpreter.

The labels for the heuristics in the table include parameters. Those parameters need some explanation:

FULL-n means the FULL heuristic, but the length of the patterns is limited to n.

STATICDICTIONARY needs no special parameters.

STATICCOUNT-x-y-z represents the STATICCOUNT algorithm with the following restrictions: each pattern must occur in at least x different benchmarks and in at least y different basic blocks. z specifies the frequencies which are considered equal. A number, say 2, means that two frequencies are considered equal if they are no more than 2% apart. If z is the special value F, then all frequencies are considered different, i.e., no pattern is removed based on (dynamic) frequency.

CHEATING is a not an actual heuristic, but is there for comparison purposes. The CHEATING algorithm ignores the distinction between training-set and benchmark: it picks those candidates from the training-set which achieve the best performance for the benchmark program. More precisely, CHEATING selects those patterns from the training-set with the highest frequency in the benchmark program. Of course, such a strategy is not very useful in practice, but it gives us an impression of what an optimal selection algorithm could achieve.

BASICBLOCKS chooses the most frequently executed basic blocks from the benchmark program as superinstructions. This is also not realistic, but we can view it as an approximation to a scheme which chooses superinstructions at runtime[PR98]. The numbers represent the reduction of dynamic dispatches.

Now let's look at the actual values in the tables. The average reduction of executed instructions varies from 50% to 87%. As expected, more superinstructions produce better speedups. Perhaps not-so-expected, the values for Java and Forth don't differ much.

STATICCOUNT-2-4-2 and STATICCOUNT-3-6-2 seem to work better than the others if the number of allowed superinstructions is less than 1000. FULL-4 and STATICDICTIONARY are the leaders in the column with 4000 possible patterns. My explanation for the better performance STATICCOUNT is as follows. STATICCOUNT is forced to choose relatively general patterns, but cannot find more than 1000. This explanation seems to be supported by the fact that STATICCOUNT basically stagnates after 600 instructions. FULL-4 on the other hand improves continuously as we increase the number of superinstructions and comes quite close to the performance of CHEATING.

Also worth to note, FULL-2, by far the simplest method, performs remarkedly well: the gap to the best competitor is rarely larger than 6 points, only in the column with 4000 superinstructions.

Heuristic	80	200	400	600	1000	4000
Full-2	80	73	68	64	62	62
Full-3	86	76	70	62	58	50
Full-4	87	79	70	65	59	50
StaticDictionary	82	73	62	64	59	52
StaticCount-1-1-F	87	86	76	70	65	57
StaticCount-2-4-2	79	70	62	60	59	59
StaticCount-3-6-2	79	70	62	59	59	59
Cheating	54	52	47	47	47	46
BasicBlocks	30	28	26	26	26	26

Table 3.2: Selection Heuristics for Forth Programs (See Fig. 3.1 left)

Heuristic	80	200	400	600	1000	4000
Full-2	82	76	70	67	64	64
Full-3	84	78	71	66	63	54
Full-4	84	80	74	69	63	55
StaticDictionary	81	72	69	68	60	59
StaticCount-1-1-F	84	80	78	74	70	60
StaticCount-2-4-2	81	71	68	63	63	63
StaticCount-3-6-2	78	71	66	66	66	66
Cheating	61	52	51	51	51	51
BasicBlocks	42	36	36	36	36	36

Table 3.3: Selection Heuristics for Java Programs (See Fig. 3.1 right)

	Relative Time [%]				
Heuristic	0	80	200	400	600
Full-4	100	89	85	77	76
StaticCount-2-4-2	100	84	76	71	70
Cheating	100	66	63	62	62

Table 3.4: Execution Times for various Selection Heuristics.

Table 3.4 shows the actual execution times for running benchmarks with GForth with different sets of superinstructions. One program formed the training set and the others served as benchmarks. This was repeated for all benchmark programs. The values in Table 3.4 represent the median of the relative times needed to execute the benchmarks, i.e., the ratio of the time needed with an unoptimized interpreter and the time needed for an unoptimized interpreter. The machine was a AMD Athlon with 756MHz, 64K/64K/256K L1-icache/L1-dcache/L2-cache, and 512MB RAM. The numbers for running actual benchmarks don't match the simulation results exactly, but the ranking between the selection heuristics is confirmed.

My conclusion of all these numbers is: a heuristic to select general patterns should primarily favor short patterns and then patterns which occur in many places.

When I did this work Gregg and Waldron's paper[GEW01] wasn't yet written. They present the following selection schemes:

static Sequences are chosen in order of static frequency.

dynamic Sequences are chosen in order of dynamic frequency.

length Sequences are weighted by length, i.e., frequencies are multiplied with the length during ranking.

normalize Frequencies are normalized across programs.

hybrid Sequences are ranked on the sum of their normalized static and dynamic frequencies.

rewrite After the highest ranking superinstructions is chosen, the program is rewritten using the current set of superinstructions. Rankings are recalculated, based on the new program.

The authors also present combinations of the different schemes. What I called FULL is roughly equivalent to a **dynamic normalize** scheme in their terminology. All my heuristics used normalization. Gregg and Waldron note that "normalizing the frequencies doesn't make very much difference to either static or dynamic schemes. When the number of superinstructions becomes very large, it appears to have some benefit, but the result is not consistent". My STATICDICTIONARY can be interpreted as a **dynamic rewrite** method. STATICCOUNT doesn't fit very well in their categorization. STATICCOUNT uses the dynamic frequency for ranking, but the number of static occurrences is used in a more ad-hoc manner. STATICCOUNT has also additional tweaks to remove similar sequences with about the same ranking. Their numbers seem to be a bit better than mine, but the difference is small ($< 5\%$) and we didn't use the same benchmarks.

Chapter 4

Parsing Strategies

In a VM with superinstructions we have usually multiple choices to replace certain sequences of non-superinstruction with superinstructions. In the following I will call an algorithm that performs this choice a *parsing algorithm* and the set of available superinstructions *dictionary*. A parsing algorithm takes as input a sequence of (simple) instructions and a dictionary. Its output is a cover of the input sequence with patterns from the dictionary. In this chapter I'll describe some general observations about parsing strategies, then I'll compare a greedy heuristic with an optimal parsing algorithm. We'll see that a greedy algorithm performs well enough for our purposes.

4.1 Parsing

Parsing covers a sequence of instructions with patterns from an extended instruction set. Of course, the cover should be minimal, i.e., the cover should contain as few instructions as possible. A parsing method that produces a, not necessarily unique, shortest cover is said to be *optimal*.

Parsing should also be efficient. How much time we can spend on parsing depends on some implementation choices. We have some degree of freedom about the stage when the parsing is actually performed:

- Compile time. If we know the superinstruction patterns at compile time, we could implement parsing as a pass in the compiler.
- Load time. Programs for VMs are often distributed in some portable format. At load time, the program is transformed from the portable format to the actual intermediate code which can be executed more efficiently. Parsing can be also be done during that translation phase.
- First time execution. It's also possible to delay the transformation from the portable format to the executable intermediate code until a particular function or basic block is executed the first time. This is sometimes desirable because it makes loading code faster, especially if large parts of the loaded code are never executed. Moreover, it's sometimes possible to combine the translation with other optimizations because at the time we

execute the code the first time, we know more about the callees of the functions than at load time or compile time. E.g., the exact address of a callee may not be known at load time, but must, of course, be available at the time we want to actually call it. We could use this knowledge to convert indirect calls to direct calls.

The efficiency of the parsing algorithm is not particularly important if we perform the parsing at compile time. But an efficient parsing algorithm is clearly desirable in the other two scenarios because there the parsing time is part of the execution time.

Now let's look at some concrete algorithms.

Greedy (without lookahead)

This algorithm scans the input one element at the time and tests whether the subsequence from the start position to the current scan position is present in the dictionary. If that subsequence is in the dictionary, then the algorithm skips to the next element. If the subsequence it is not in the dictionary, then the longest pattern so far is emitted, the start pointer is set to the current scan position, and the process continues.

This method is easy to implement and fast. But it is not always optimal. The algorithm always selects the longest possible pattern and ignores the possibility that choosing a shorter pattern at first could result in a shorter overall cover. E.g., if we have a dictionary with the sequences {`a_a`, `a_b`, `a_b_c`}, then the input `a a b c` will be parsed as `a_a b c` and not as `a a_b_c`

However, GREEDY *is* optimal if the dictionary is a *suffix dictionary*[CK96]. Suffix dictionary means that each suffix of a pattern in the dictionary is also in the dictionary. (If we parse from right-to-left, then we have to include all prefixes.)

First Longest Match

FIRSTLONGESTMATCH is an extension to GREEDY and often called "GREEDY with lookahead" or "lazy parsing".

This method tries to fix the following problem: Consider a dictionary consisting of the patterns `a_b` and `a_b_c_d`. Since the pattern `a_b_c` is not present in the dictionary, GREEDY without lookahead will never select the sequence `a_b_c_d`; not even if the input contains exactly `a b c d`. As we can see from this example, it doesn't make sense to include a pattern if we don't also include all its prefixes.

FIRSTLONGESTMATCH looks ahead to cope with this complication. To implement this idea we could organize the dictionary in a trie datastructure, i.e., a tree with a node for each dictionary entry and the edges from the root to a node are labeled with the corresponding instructions from the dictionary entry. Those internal nodes which don't correspond to a dictionary entry are marked as *empty*. To parse the input, we follow the corresponding path in the trie until we find no successor node for the input. At that point, we backtrack to the last non-empty node, emit the corresponding instruction, and then continue.

The time complexity of this algorithm is $O(n \times l_{max})$, where n is the length of the basic block and l_{max} is the length of the longest pattern. Like GREEDY, FIRSTLONGESTMATCH is optimal if the dictionary contains each suffix of each pattern.

Optimal Parsing

There exist algorithms to optimally parse a sequence with respect to a (general) dictionary. These algorithms use Dynamic Programming and run in $O(n \times l_{max})$ time, where n is the basic block length and l_{max} the maximum pattern size.

If the dictionary is fixed, we can use more sophisticated methods. E.g., BURG[FHP91] is a tool to generate parsers which run in linear time. However, I had some practical problems with BURG. The tool worked well for small grammars, but aborted with an internal error for larger grammars (> 200 superinstructions). Maybe my choice of grammar was just unfortunate or perhaps I hit some internal limit. I haven't investigated the exact cause of the problem and used Dynamic Programming[CLR90] instead.

For each position in the basic block we compute the optimal cover from the position to the end of the block and also the cost of the cover. To find the optimal cover at position i, we try all patterns and choose the pattern p that matches at position i and generates the minimal cost. The cost of the cover at position i is the cost of the matching pattern p added to the cost of the cover at position $i + length(p)$. If we assign each pattern the same cost, say 1, we will find the shortest cover. The computation starts at the last position of the basic block, so that the costs computed in previous steps can be reused.

4.2 Comparison

To compare the different optimal parsing algorithms, I created two different dictionaries. The first dictionary was created with the CHEATING heuristic and second with the STATICCOUNT-2-4-2 selection heuristic. Then I computed the reduction of executed virtual instruction like in Section 3.4.

Table 4.1 and 4.2 show the results. FLM in tables is the abbreviation for FIRSTLONGESTMATCH.

Parsing Method	Heuristic	80	200	400	600	1000	4000
GREEDY	STATICCOUNT	79	71	62	62	61	61
FLM	STATICCOUNT	79	70	62	60	59	59
OPTIMAL	STATICCOUNT	79	70	62	60	59	59
GREEDY	CHEATING	54	52	47	47	47	46
FLM	CHEATING	54	52	47	47	47	46
OPTIMAL	CHEATING	54	52	47	47	47	46

Table 4.1: Comparing Parsing Strategies for Forth

As we can see from these numbers, FIRSTLONGESTMATCH performs essentially as well as an optimal parsing algorithm. GREEDY is a little worse, but not

Parsing Method	Heuristic	80	200	400	600	1000	4000
Greedy	StaticCount	81	72	69	65	64	64
FLM	StaticCount	81	71	68	63	63	63
Optimal	StaticCount	81	71	68	63	63	63
Greedy	Cheating	61	52	51	51	51	51
FLM	Cheating	61	52	51	51	51	51
Optimal	Cheating	61	52	51	51	51	51

Table 4.2: Comparing Parsing Strategies for Java

much. Note that all parsing algorithms use the same dictionary and some of the patterns may be useless for Greedy because some prefix of those patterns may not be present in the dictionary.

One possible reason for the good performance of the non-optimal algorithms is the length of the basic blocks. Most basic blocks are short, 3–5 instruction long. FirstLongestMatch is optimal for basic blocks of length 3 and the penalty for making an non-optimal choice for blocks of length 4 or 5 is apparently small.

The Cheating heuristic selects a suffix dictionary (and also prefix dictionary) because the heuristic chooses shorter sequences before longer sequences. This means, for the Cheating selection strategy the Greedy parser is optimal. As Full also selects a suffix dictionary, Greedy is also optimal for Full (not listed in the table).

I didn't verify the numbers in an actual implementation because the existing implementation uses already an optimal algorithm. The main advantage of Greedy is the simpler implementation, but it wasn't clear at the beginning that the difference would be so small and the optimal algorithm was the safe choice.

However, the numbers suggest that a little implementation and maintenance effort could have been saved by using Greedy.

Chapter 5

Superoperators

Like superinstructions, superoperators are a method to combine multiple operators to larger, more efficient, operators. While superinstructions are sequence oriented, superoperators are tree oriented. In this chapter we'll look at ways to discover superoperators in instruction sequences and compare superoperators with superinstructions.

5.1 Tree Patterns

Superoperators and superinstructions are similar ideas, but superoperators are easier to explain in the context of trees and tree patterns. So let's assume our programs are represented with trees and further assume that tree patterns are written as `(a (b *) *)`, where `a` and `b` are names for operators and '`*`' is a wildcard. A superoperator combines an operator with some of its operands to a larger operator, e.g., the superoperator for the above pattern is `(ab * *)`.

Of course, a program represented as a tree can easily be transformed to a sequence of instructions, e.g., with a post-order left-to-right traversal. E.g., if we transform the tree pattern `(a (b 2) *)`, we get the sequence pattern `2 b * a`. Note that the wildcard `*` could be a tree, resp. sequence, of arbitrary size. Also note that the operators `a` and `b` are adjacent in the tree representation, but they are not adjacent in the sequence representation.

As we have seen, superoperators use a different pattern class than superinstructions. An interesting question is whether such tree patterns, i.e., non-sequential patterns, occur frequently in practice and whether we should exploit them.

To answer these questions, I simulated and compared a VM with support for superoperators with a VM with superinstructions. The method I used to simulate and evaluate the superoperator VM is the same method as for the superinstruction VM in Section 3.2. I generated candidate patterns from the training programs, used a heuristic to select the fittest candidates and computed the speedups compared to an unoptimized VM. The details are, of course, different for tree patterns and are explained in the following sections.

5.2 Pattern Generation

The first problem we need to solve is pattern generation. We receive a set training programs and our task is to return a set of tree patterns from those programs. This task is complicated by the fact that our programs are not represented as trees, but as sequences. The approach I used was as follows: First convert the basic blocks to a tree representation and then take all subtrees of the tree representation as patterns.

I used the stack effects of the instructions to guide the conversion from sequence representation to tree representation. The stack effect describes how many items an instruction takes from the stack and the number of items it leaves on the stack. E.g., the stack effects for some common instructions in GForth are:

```
lit :     -- x
  + : x y -- z
  @ : x   -- y
dup : x   -- x x
```

`lit` takes zero stack items, `+` takes two items, and `@` takes one item from the stack; all three instruction leave one result item behind. `dup` takes one item and produces two results. The stack effects of the instructions are part of a `vmgen` description and I extracted the effects from there. The stack effects for the Java instructions are the same as the stack effects described in the Java VM specification[LY99] with some additions for instructions which are only used by the GCJ interpreter.

The conversion function performs a simple abstract interpretation of the input sequence according to the stack effects. Instead of concrete values, the generated trees are pushed and popped from the stack. E.g., consider the input sequence

```
lit @ lit +
```

then the stack after each abstract interpretation step looks as follows:

```
                          (lit)
1: (lit)  2: (@ (lit))  3: (@ (lit))  4: (+ (@ (lit)) (lit)) .
```

The result tree is `(+ (@ (lit)) (lit))`. Because it is possible that we have multiple items on the stack at the end, we return a list of trees and not a single tree. In other words, the result of the conversion is a forest.

This conversion scheme fails for stack effects with multiple or zero output values. To fix this problem, we flush the stack whenever we encounter such a problematic stack effect and begin a new tree. At the end, we add the flushed stack items to the result forest. E.g., the sequence `lit dup +` is converted to the 2 element forest `[(dup (lit)) (+ * *)]`, i.e., the stack is flushed after `dup`. This example also introduces wildcards: a wildcard is created when the operator needs to take a value from the stack, but the stack is empty. Note that creating a forest is not the only possible solution to the problem caused by multiple return values, e.g., we could use DAGs instead of trees. I used forests because that's simple. Algorithms for DAGs are often more complicated than algorithms the trees, e.g., Least-Cost Instruction Selection in DAGs is NP-Complete[Pro]. It's also unclear whether DAGs have any advantages over forests for pattern discovery.

Another complication are multiple stacks. E.g., GForth has a data stack, a return stack, and a floating point stack. The converter uses only a single stack, but marks each stack item with a tag for the type of the item, i.e., items for the data stack receive a different tag as the items for the return stack. If an operator needs to take a value from the stack, it also tests whether the value has the correct tag. If the tag doesn't match, the stack is again flushed and a new tree is started.

So in summary, we build a tree form the items on the stack if the stack effect for the operator has exactly one result and if all argument tags match. In the other cases we flush the stack and begin a new tree.

			Distribution (static)					
Program	dyn. Avg.	stat. Avg.	1	2	3	4	5	> 5
brainless	1.97	1.97	0.60	0.15	0.11	0.07	0.03	0.04
brew	1.90	1.69	0.66	0.15	0.10	0.05	0.02	0.02
pentomino	2.71	1.69	0.65	0.16	0.12	0.04	0.01	0.02
siev	1.22	1.59	0.67	0.16	0.10	0.05	0.01	0.01
bubble	1.46	1.61	0.67	0.16	0.11	0.05	0.01	0.01
matrix	1.71	1.59	0.68	0.16	0.10	0.04	0.01	0.01
fib	1.31	1.60	0.67	0.16	0.11	0.04	0.01	0.01
regexmatch	1.64	1.61	0.66	0.16	0.11	0.05	0.01	0.01
lists	1.29	1.60	0.67	0.17	0.10	0.05	0.01	0.01
spellcheck	1.42	1.60	0.67	0.15	0.11	0.04	0.01	0.01
heapsort	1.90	1.64	0.66	0.16	0.11	0.05	0.01	0.01
check	2.85	1.52	0.70	0.17	0.08	0.03	0.01	0.01
compress	2.01	1.76	0.59	0.20	0.13	0.05	0.01	0.02
jess	1.60	1.41	0.75	0.16	0.07	0.01	0.01	0.01
db	1.77	1.47	0.69	0.20	0.07	0.03	0.01	0.00
javac	1.55	1.46	0.72	0.13	0.14	0.01	0.00	0.00
mpegaudio	2.84	1.93	0.56	0.05	0.36	0.02	0.00	0.01
jack	1.43	1.42	0.72	0.16	0.11	0.00	0.01	0.00

Table 5.1: Tree Sizes in the Benchmarks

Table 5.1 shows the average dynamic size, the static size and the size distribution of the trees. Most trees contain only one operator and few trees have more than four. `mpegaudio` is the only program with more trees of size three than trees of size two.

It think one reason for relatively small Java trees was the unknown stack effect for `invokevirtual`. I used an unknown stack effect because `invokevirtual` can take an arbitrary number of items from the stack and can push 0, 1, or 2 items. Since the stack effect was unknown, `invokevirtual` was never combined to a tree, but there seem to be many occurrences of sequences like `aload invokevirtual`. A similar problem occurred with the `getfield` operator which can push 1 or 2 stack items. (2 items are pushed if the type is `long` or `double`.)

When we have a set of forests, we can begin to generate the actual patterns. I generated patterns by enumerating all subtrees with at least two but at most six operators. E.g., the generated patterns for the tree `(+ (@ (lit)) (lit))` are:

```
(+ * (lit))
(+ (@ *) *)
(+ (@ *) (lit))
(+ (@ (lit)) *)
(+ (@ (lit)) (lit))
(@ (lit)).
```

I used two variations of the above scheme. The first variant computes for each basic block the set of trees and uses all subtrees as patterns.

The second variant converts basic blocks to a sequence of trees and generates patterns for sequences of trees. E.g., the patterns for the sequence of two trees `[(dup (lit)) (+ (lit) *)]` include all subtrees of `(dup (lit))` and `(+ (lit) *)`, but also sequences of trees like `[(dup *) (+ * *)]`. Note however that wildcards are never introduced at root nodes of trees, e.g., patterns like `[(dup *) * (+ * *)]` are not allowed. Each element in such a sequence pattern must be a tree and the operators in the root notes are the same as the corresponding operator in the original tree.

5.3 Selection and Parsing

Pattern selection for tree patterns proceeds very similar as for sequence patterns. As for sequences, we need to normalize the execution frequencies when we combine patterns from different training programs. The normalization formula is the same as for sequences, but we use the size of the tree instead of the length of the sequence. With minor adaptions we can then use the same selection heuristics as for sequence patterns.

To parse the benchmark programs with respect to a dictionary of tree patterns, I first converted the basic blocks in the program to trees and covered the trees with the patterns from the dictionary. I used an optimal parsing algorithm.

5.4 Comparison with Superinstructions

Figure 5.1 shows the simulation results for superoperators and for superinstructions. The corresponding data is listed in Table 5.2 and 5.3. The labels in the with /Tree in their names correspond to a VM with superoperators for pure tree patterns. Labels with /Forest suffixes are for VMs with superoperators

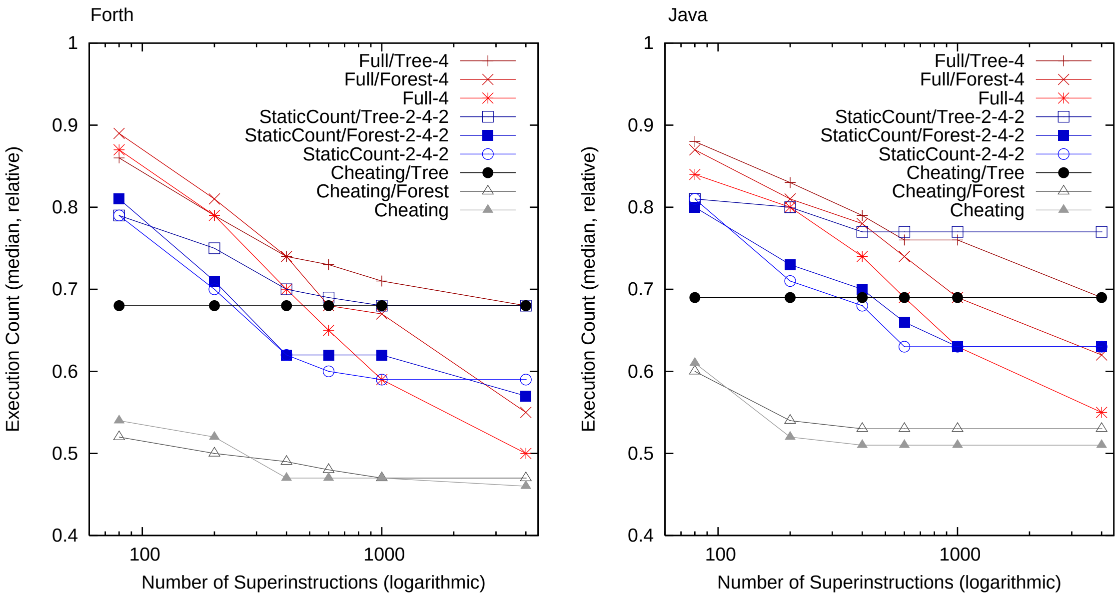

Figure 5.1: Comparision of superoperators and superinstructions

for sequences of tree patterns. The remaining labels name VMs with superinstructions and are the same as in Section 3.4.

FULL/TREE and FULL/FOREST are the tree oriented pendants to the FULL heuristic for sequences. The rows for STATICCOUNT and CHEATING correspond in a similar manner.

The values in the tables are computed in the same way as the values in Section 3.4. One value in the tables represents the median of the relative VM execution counts for the different benchmarks. I.e., the median of dividing the number of VM instructions needed by an interpreter with superoperators by the number of VM instructions needed by an interpreter without superoperators.

The numbers in the table indicate that the tree variant performs worst. The forest variant is almost as good as the superinstruction variant, but rarely better. Like for sequence variant, STATICCOUNT for trees or forests doesn't improve much if we allow more than 400 patterns.

It is a bit surprising that the tree version performs better than the forest version with 80 allowed patterns for the Forth benchmarks. The forest variant should perform at least as well as the tree variant, because the forest version can use the same patterns as tree variant in a addition to sequences of trees. I think that the non-optimal selection strategy is the cause of this unexpected result.

Another point to note is that forest patterns are not necessarily more general than pure sequence patterns. Consider a basic block `lit lit 1+ +`. The block corresponds to the forest `[(+ (lit) (1+ (lit)))]`. But while there will be a sequence pattern for `lit_lit` there will be no equivalent forest pattern. In principle it is possible to include a pattern for, say `[(lit) (lit)]`, but the parser, as it is implemented now, would not be able to find it. With an

Heuristic	80	200	400	600	1000	4000
Full/Tree-4	86	79	74	73	71	68
Full/Forest-4	89	81	74	68	67	55
Full-4	87	79	70	65	59	50
StaticCount/Tree-2-4-2	79	75	70	69	68	68
StaticCount/Forest-2-4-2	81	71	62	62	62	57
StaticCount-2-4-2	79	70	62	60	59	59
Cheating/Tree	68	68	68	68	68	68
Cheating/Forest	52	50	49	48	47	47
Cheating	54	52	47	47	47	46

Table 5.2: Comparison of superoperators with superinstructions for Forth programs. The numbers are in %. See Figure 5.1 left.

Heuristic	80	200	400	600	1000	4000
Full/Tree-4	88	83	79	76	76	69
Full/Forest-4	87	81	78	74	69	62
Full-4	84	80	74	69	63	55
StaticCount/Tree-2-4-2	81	80	77	77	77	77
StaticCount/Forest-2-4-2	80	73	70	66	63	63
StaticCount-2-4-2	81	71	68	63	63	63
Cheating/Tree	69	69	69	69	69	69
Cheating/Forest	60	54	53	53	53	53
Cheating	61	52	51	51	51	51

Table 5.3: Comparison of superoperators with superinstructions for Java Programs. The numbers are in %. See Figure 5.1 right.

appropriate parser (and a good selection heuristic) it could be possible to use the union of pure sequence pattern and forest patterns as candidates to achieve better results than with sequence patterns alone. But implementing such a parser may not be easy.

Implementing Tree Patterns

Note that the presented numbers are only the result of a simulation and I didn't actually implement superoperators in a real VM. It is not quite obvious, at least not to me, how superoperators or other non-sequential combinations can be implemented efficiently. Consider a pattern like `a * b`. How should we implement this in a single virtual instruction? Proebsting[Pro95] rewrites the pattern to `* a b`. The rewritten version is easy to implement by introducing a new operator for `a_b`. But such a reordering is in general not semantics preserving, e.g., if `a` has a side effect on which `*` depends. One possibility is to constrain the trees such that side effects can only occur in root nodes. If non-root operators have no side-effects they can be reordered safely. Alternatively, we could perform data-flow analysis to decide whether a reordering is allowed. But such an analysis would be quite complicated.

Chapter 6

Conclusions

Superinstructions are an effective way to reduce the interpretation overhead of VM interpreters. A superinstruction combines a sequence of ordinary instructions to a larger instruction. Since the interpretation overhead for a superinstruction is comparable to the overhead for an ordinary instruction, the overall overhead for executing the superinstruction is smaller than for executing the corresponding sequence of individual instructions.

In this work I tried to answer the following questions:

- *Which superinstruction should we add to a VM?* Because the problem of choosing an optimal set of superinstructions is NP-complete, we should use a heuristic. I compared some different selection heuristics. If the number of allowed superinstructions was very large, > 1000, then the heuristic which chooses all possible subsequences up to length 4 achieved the best results. If the number of allowed superinstructions was more limited, then a heuristic which favors short sequences and sequences which occur in many different programs and many different basic blocks performed better than the others.

- *How much effort should we spend for parsing?* I compared optimal parsing with a greedy algorithm and found that the greedy algorithm achieves almost optimal results.

- *Should we use non-sequential patterns?* I evaluated the optimization potential of superoperators, i.e., combinations of instructions which correspond to tree patterns. I found that superoperators don't offer significantly more optimization possibilities than superinstructions. In my experiments, superinstructions performed slightly better than superoperators. I think that non-sequential patterns are too complicated to implement and simple sequential patterns should be used instead.

References

[BCW90] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice-Hall, Inc., 1990.

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[BJEG95] A. Brazma, I. Jonassen, I. Eidhammer, and D.R. Gilbert. Approaches to the automatic discovery of patterns in biosequences. Technical report, 1995.

[CGEN03] Kevin Casey, David Gregg, M. Anton Ertl, and Andrew Nisbet. Towards superinstructions for Java interpreters. In A. Krall, editor, *Software and Compilers for Embedded Systems*, volume 2826 of *LNCS*, pages 329–343. Springer-Verlag Heidelberg, 2003.

[CK96] Martin Cohn and Roger Khazan. Parsing with suffix and prefix dictionaries. In *Proc. Data Compression Conference (DCC'96)*. IEEE Computer Society Press, 1996.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[EF01] William S. Evans and Christopher W. Fraser. Bytecode compression via profiled grammar rewriting. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 148–155, 2001.

[EG03] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN '03 Conference on Programming Language Design and Implementation*, 2003.

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

[Ert] M. Anton Ertl. Threaded code. `http://www.complang.tuwien.ac.at/forth/threaded-code.html`.

[Ert93] M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Láznè (Marienbad), 1993.

[FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995. ISBN 0-8053-1670-1.

[FHP91] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG–fast optimal instruction selection and tree parsing. Technical Report CS-TR-1991-1066, 1991.

[FP95] Christopher W. Fraser and Todd A. Proebsting. Custom instructions sets for code compression. Unpublished, 1995. `http://www.cs.arizona.edu/people/todd/papers/pldi2.ps`.

[GEW01] David Gregg, M. Anton Ertl, and John Waldron. The common case in Forth programs. In *EuroForth 2001 Conference Proceedings*, pages 63–70, 2001.

[GH03] Etienne Gagnon and Laurie Hendren. Effective inline-threaded interpretation of Java bytecode using preparation sequences. In *Proceedings of the 12th Internetional Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 170–184, April 2003.

[GW02] David Gregg and John Waldron. Primitive sequences in general purpose Forth programs. In *Proceedings of the 18th EuroForth European Conference on Forth*, pages 24–32, Vienna, Austria, September 2002.

[HATW99] Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik Van De Wiel. A code compression system based on pipelined interpreters. *Software–Practice & Experience*, 29(11):1005–1023, September 1999.

[Hug82] R. J. M. Hughes. Super-combinators a new implementation method for applicative languages. In *Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 1–10, 1982.

[LDK99] Stan Liao, Srinivas Devadas, and Kurt Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 4(1):12–38, January 1999.

[LY99] Tim Lindholm and Frank Yellin. The Java virtual machine specification. Addison-Wesley Pub Co; 2 edition, 1999.

[N01] Henrik Nässén. Optimizing the SICStus Prolog virtual machine instruction set. Technical Report T2001-01, Computer Science Department, Uppsala University, March 2001.

[NCS01] Henrik Nässén, Mats Carlsson, and Konstantinos Sagonas. Instruction merging and specialization in the SICStus Prolog virtual machine. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and Practice of Declarative Programming*, pages 49–60. ACM Press, 2001.

[OP04] Diarmuid O'Donoghue and James F. Power. Identifying and evaluating a generic set of superinstructions for embedded Java programs. Technical Report NUIM-CS-TR-2004-04, National University of Ireland, Department of Computer Science, February 2004.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct-threaded code by selective inlining. *ACM SIGPLAN Notices*, 33(5):291–300, May 1998.

[Pro] Todd Proebsting. Least-cost instruction selection in DAGs is NP-complete. `http://www.research.microsoft.com/~toddpro/papers/proof.htm`.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332. ACM Press, 1995.

[Weg96] Maarten Wegdam. Compact code generation through custom instruction sets. Technical Report TN 417/96, Philips Research, Eindhoven, The Netherlands, December 1996.

[ÁBFG+03] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. Survey of code-size reduction methods. *ACM Comput. Surv.*, 35(3):223–267, 2003.