

Procedural Abstraction with Reverse Prefix Trees

Stefan Schaeckeler
Department of Computer Engineering
Santa Clara University
Santa Clara, CA 95053
sschaeck@engr.scu.edu

Weijia Shang
Department of Computer Engineering
Santa Clara University
Santa Clara, CA 95053
wshang@scu.edu

Abstract—For memory constrained environments like embedded systems, optimization for size is often as important as, if not more important than, optimization for execution speed. A common technique for compacting code is procedural abstraction. Equivalent code fragments are identified and abstracted into a procedure. The standard algorithm for identifying these fragments is based on suffix trees. We propose in this paper the calculation of suffix trees over the program text not in the common top-down fashion, but reversed, i.e. bottom-up. With this simple modification, not only equivalent fragments can be identified, but also fragments equivalent to (possibly often differently long) suffixes of the longest fragments. A longest fragment is then abstracted, and all fragments are replaced by procedure calls to their corresponding start instruction *somewhere* in the abstracted procedure. This allows us to harvest more and longer fragments than with standard suffix trees, improving code size reductions on average by 8.277% over standard suffix trees.

Keywords—embedded systems; code compaction; code size reduction; post-pass optimization; procedural abstraction; suffix tree; reverse prefix tree; program visualization

I. INTRODUCTION

For memory constrained environments like embedded systems, optimization for size is often as important as, if not more important than, optimization for execution speed. Large portions of the die are devoted to ROM, RAM and cache memories and optimizing a program for size may either lead to smaller dies, or alternatively to more functionality on the original die. The dominant factor in the cost of an embedded system is the total die size and reducing die memory will result in cheaper manufacturing costs. Mobile devices like mp3 players, cell phones or personal digital assistants (PDAs) are gaining in popularity and as smaller memories consume less power; this may also translate into longer running times of such devices.

A common technique for compacting code is procedural abstraction [1]. Equivalent code fragments¹ are identified, abstracted in a new procedure, and then replaced by procedure calls. This saves all but one occurrence of the fragments and adds a small overhead of one procedure

¹Procedural abstraction can be implemented on top of assembler/machine code, intermediate code, or source code. Due to its fine granularity, best results are achieved on top of assembler/machine code.

call per fragment and one return instruction per abstracted procedure.

The standard algorithm for identifying equivalent code fragments is based on suffix trees. Although suffix trees fail to identify overlapping fragments, they are used for their fast generation.

We propose in this paper the calculation of suffix trees over the program text not in the common top-down fashion, but reversed, i.e. bottom-up. With this simple modification, not only equivalent fragments can be identified, but also fragments equivalent to (possibly often differently long) suffixes of the longest fragments. A longest fragment is then abstracted into a procedure, and all fragments are replaced by procedure calls to their corresponding start instruction *somewhere* in the abstracted procedure. *Reverse prefix trees*, as we call our bottom-up generated suffix trees, allow us to harvest more and longer fragments, improving code size reductions up to 13.387% over standard suffix trees with an average improvement of 8.277%.

The contribution of this paper is twofold. First, we introduce reverse prefix trees as an efficient tool for abstracting more and longer fragments than possible with standard suffix trees while the computational complexity remains $O(n \cdot \log(n))$. Second, we give detailed analyses, measurements and comparisons of standard suffix tree and reverse prefix tree procedural abstraction.

The rest of the paper is organized as follows. The next section goes over related work in the field of procedural abstraction. Section III reviews standard suffix tree procedural abstraction and introduces reverse prefix tree procedural abstraction. Section IV describes our implementations of two post pass compactors for suffix tree and reverse prefix tree procedural abstraction. Section V presents and analyzes benchmark results. Section VI outlines ideas for future work and Section VII concludes the paper.

II. RELATED WORK

Procedural abstraction, or *abstraction* for short, is based in its simplest form on syntactic equivalence of instructions, e.g. on identical fragments, but often on semantic equivalence of instructions. We call such fragments *equivalent fragments*.

Fraser et al. observed that some fragments are equivalent except for different register names [1]. Cooper and McIntosh recolored in [2] the register conflict graph to render such fragments equivalent and proposed to abstract different constants in a similar way. They further proposed to reorder instructions to generate additional duplicate code. Dreweke et al. solve this with graph mining algorithms on the data flow graph [3].

Liao et al. showed how to identify fragments equivalent to suffixes of the longest fragment. They give in [4] first an $O(n^2)$ time algorithm to identify all equivalent fragments and describe then a set-covering problem for choosing the fragments for the actual abstraction. Gyimóthy et al. give in [5] also such an algorithm in $O(n^2)$ time. In this paper, we introduce an algorithm based on reverse prefix trees to efficiently identify such fragments in $O(n \cdot \log(n))$ time. Both, Liao et al. and Gyimóthy et al. do not provide any comparison with standard procedural abstraction and it remained open whether there is an actual improvement for real programs. In this paper, we give also detailed measurements and comparison of both abstractions.

III. SUFFIX TREES AND REVERSE PREFIX TREES

Suffix trees are used for a variety of pattern matching purposes [6]. Fraser et al. first observed their use for procedural abstraction [1]. Suffix trees were first introduced by Weiner [7] as position trees and subsequently simplified by McCreight [8] and Ukkonen [9]. In this paper, we develop a suffix tree variant, reverse prefix trees, that is better suited for procedural abstraction. In Section III-A we review suffix trees and their use for procedural abstraction and in Section III-B we introduce reverse prefix trees and their use for procedural abstraction.

A. Suffix Trees and their Use for Procedural Abstraction

Let S be a string over an alphabet Σ and let s_i and $S[i]$ denote the i^{th} symbol.² S has exactly $n = |S|$ non-empty suffixes $\{s_m s_{m+1} \dots s_n \mid m \in \{n, n-1, \dots, 1\}\}$, e.g. the string *possessor* has the twelve non-empty suffixes $s, ss, ess, \dots, possessor$.

A tree $\mathfrak{T}(S) = (\mathfrak{V}, \mathfrak{E})$ is the suffix tree for a string $S \in \Sigma^*$ if each internal vertex, except perhaps the root τ , has at least two child vertices and if all edges $e \in \mathfrak{E}$ are labeled with non-empty strings $l(e) \in \Sigma^+$ so that a) concatenation of labels along all paths from τ to leaves are exactly all suffixes of S and b) labels of edges between vertices and their child vertices start with different symbols, i.e. $\forall v \in \mathfrak{V}: \forall u \rightarrow v \in \mathfrak{E}: \forall w \rightarrow v \in \mathfrak{E}: u = w \Leftrightarrow l(v \rightarrow u)[1] = l(v \rightarrow w)[1]$. The suffix tree exists for a string S iff S ends in a symbol not seen earlier. This can be enforced by including in Σ a further symbol $\$$ and appending it to S .

²In our examples we use for the sake of brevity the Latin alphabet $\Sigma = \{a, b, c, \dots, z\}$. For compacting code, the alphabet Σ is really the instruction set of an architecture.

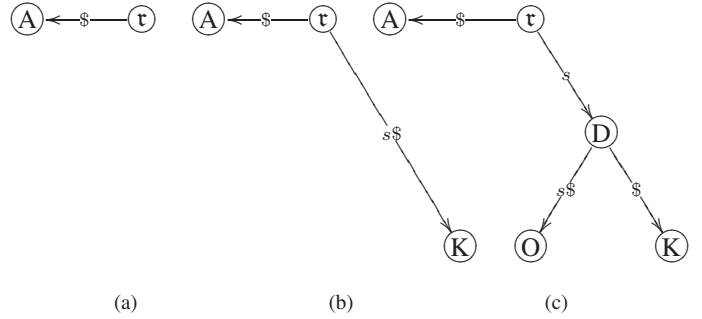


Figure 1. Suffix Tree Generation for *possessor*\$

We define a label $\tilde{l}(e_1 e_2 \dots e_n) \in \Sigma^*$ for a path $e_1 e_2 \dots e_n$ as the concatenation of the labels along its edges e_1, e_2, \dots, e_n , i.e.

$$\tilde{l}(\epsilon) = \epsilon \quad (1)$$

$$\tilde{l}(e_1 e_2 \dots e_n) = l(e_1) \tilde{l}(e_2 \dots e_n). \quad (2)$$

For a clean notation, we redefine \tilde{l} as l . The suffix tree for a string S can be constructed in $O(n^2)$ time by letting all suffixes of S sink in an initially empty tree. A suffix $s_m s_{m+1} \dots s_n$ of S sinks in the tree by creating a new branch at string position $k = \max(\{p \mid S[i+m-1] = l(\tau = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots)[i], i = 1 \dots p\} \cup \{0\})$ for paths $\tau = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots$. The new branch is then labeled $s_{k+1} s_{k+2} \dots s_n$. We refer for faster suffix tree generation algorithms in $O(n \cdot \log|\Sigma|)$ time to the literature [7]–[9].

Fig. 1 shows the first three intermediate steps of constructing the suffix tree for the string *possessor*\$. We let all non-empty suffixes sink in the empty tree starting with the shortest suffix. This suffix, $\$$, creates instantly a new branch, i.e. at string position 0 on an empty path (Fig. 1a). The next suffix, $s\$$, creates instantly a further new branch, i.e. at string position 0 again on an empty path (Fig. 1b). The next suffix, $ss\$$, creates a new branch at string position 1 on the path $\tau \rightarrow K$, because $ss\$$ shares with this path the prefix s of length 1. The new branch $D \rightarrow O$ is labeled $s\$$ (Fig. 1c). Fig. 2 shows then the complete suffix tree for *possessor*\$.

By construction of the suffix tree, identical sub-strings can be identified from the suffix tree. An internal vertex $v_i \neq \tau$ describes identical fragments. The leaves $c_{i,k}$ in the sub-tree \mathfrak{T}_{v_i} of v_i correspond to suffixes sharing a same prefix. This prefix $l(\tau \rightarrow^+ v_i)$ along the path from root to v_i occurs within S ending in positions $\mathfrak{P}_{v_i} = \{|S| - |l(v_i \rightarrow^+ c_{i,k})| \mid c_{i,k} \text{ is a leaf}\}$ for paths from v_i to leaves $c_{i,k}$. The size of \mathfrak{P}_{v_i} is exactly the number of leaves in \mathfrak{T}_{v_i} .

An example is given in Fig. 2. At vertex J hangs a sub-tree with three leaves $O, P,$ and Q . Each of these leaves corresponds to a suffix with the prefix $l(\tau \rightarrow^+ J) = ss$. The end positions of ss can be derived from $|l(J \rightarrow O)| = 1$, $|l(J \rightarrow P)| = 6$, and $|l(J \rightarrow Q)| = 9$ and are $13 - 1 = 12$, $13 - 6 = 7$, and $13 - 9 = 4$. For vertex E , as a further

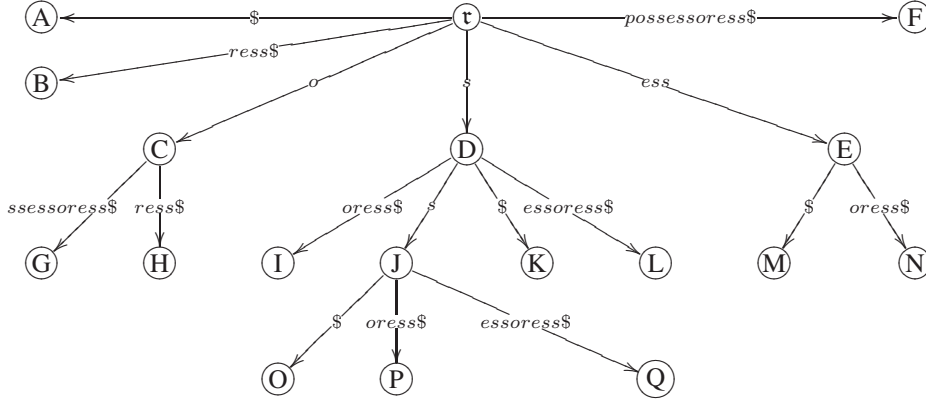


Figure 2. Suffix Tree for $possessoress\$$

example, the sub-string ess can be found ending in positions 7 and 12.

For procedural abstraction, we are using the term *code fragment* or in short, *fragment*, for *sub-string*. We call identical or equivalent fragments a *set of fragments*, or in short, a *set*. Sets are candidates for *abstraction*, e.g. a fragment can be abstracted in a separate procedure and each fragment is then replaced by a procedure call. The procedures are "lightweight", neither with procedure prologues or epilogues, and no arguments are passed or returned. In the example, ss can be abstracted in a separate procedure p so that all three occurrences of ss ending in positions 4, 7, and 12 are replaced by calls to p . Alternatively, ess can be abstracted in a separate procedure p so that all two occurrences of ess ending in positions 7 and 12 can be replaced by calls to p .

Sets can overlap, either internally or with other sets. Such *conflicts* cannot be determined from suffix trees. We will come back to this in Section IV-B.

B. Reverse Prefix Trees and their Use for Procedural Abstraction

The reverse prefix tree for a string S is the suffix tree for the reversed string $s_n s_{n-1} \dots s_1$. It can be constructed in $O(n^2)$ time by letting all reversed non-empty prefixes $\{s_m s_{m-1} \dots s_1 \mid m \in \{1, 2, \dots, n\}\}$ sink in an initially empty tree—thus, we have chosen the name *reverse prefix tree*. The fast suffix tree algorithms [7]–[9] can be also used for constructing the reverse prefix tree in $O(n * \log |\Sigma|)$ time by constructing the suffix tree over the reversed string.

The reverse prefix tree exists for a string S iff S begins with a symbol not found later, again. This can be enforced by including in Σ a further symbol $\hat{}$ and preceding it to S .

As with suffix trees, identical sub-strings can be identified from reverse prefix trees. The leafs $c_{i,k}$ for the sub-tree \mathfrak{T}_{v_i} of an internal vertex $v_i \neq \tau$ correspond to reversed prefixes sharing a same prefix. This prefix $l(\tau \rightarrow^+ v_i)$, when reversed, e.g. read from v_i to τ , occurs within S starting in positions $\mathfrak{P}_{v_i} = \{|l(v_i \rightarrow^+ c_{i,k})| + 1 \mid c_{i,k} \text{ is a leaf}\}$ for

paths from v_i to leafs $c_{i,k}$. As before, the size of \mathfrak{P}_{v_i} is exactly the number of leafs in \mathfrak{T}_{v_i} .

To see how reverse prefix trees can give a finer control over the shapes of abstracted sets, we continue with the example from Section III-A. Fig. 3 gives the reverse prefix tree for the string $S = possessoress\$$. As S already starts with a symbol not found later, preceding with $\hat{}$ is not necessary.

Consider vertex H together with vertex O . From vertex O , we can conclude that there are two reversed prefixes with the same suffix sse . One for leaf R , and one for leaf S . From vertex H , we can conclude that there are three reversed prefixes with the same suffix ss . They are the two reversed prefixes with the suffix sse and another reversed prefix for leaf N . Vertices H and O thus describe together the three sub-strings sse , sse , and ss . When reversed, they appear within S starting in positions 5, 10, and 3.

For procedural abstraction, a longest fragment can be, when reversed, abstracted into a separate procedure, and each fragment is then replaced by a call into this procedure, i.e. an abstracted procedure can be entered at different points, but is always left at the bottom. When abstracting ess in a separate procedure p , the two occurrences of ess at positions 5 and 10 can be replaced by calls to the first instruction of p and the single occurrence of ss at position 3 can be replaced by a call to the second instruction of p .

More formally: a set of fragments is identified by a pair of two internal vertices ($v_i \neq \tau, v_j$) on a path $\tau = v_0 \rightarrow^+ v_i \rightarrow^* v_j \rightarrow^+ v_{j,k} = c_{j,k}$ from τ to leafs $c_{j,k}$ in the sub-tree \mathfrak{T}_{v_j} . The sub-string for abstraction is the reversed sub-string $l(\tau \rightarrow^+ v_j)$ along the path τ to v_j .

By construction of the reverse prefix tree, the leafs $c_{j,k}$ in the sub-tree \mathfrak{T}_{v_j} describe for an internal vertex v_j sub-strings equivalent to the abstracted sub-string. They can be found within S at positions starting $\mathfrak{P}_{v_j} = \{|l(v_j \rightarrow^+ c_{j,k})| + 1 \mid c_{j,k} \text{ is a leaf}\}$. These sub-strings can be replaced for abstraction with calls to the beginning of the abstracted sub-string.

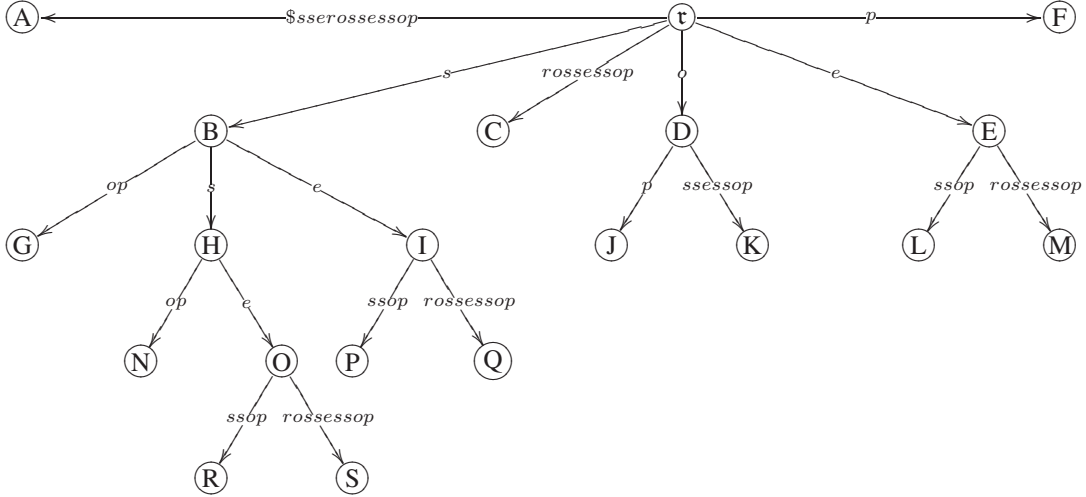


Figure 3. Reverse Prefix Tree for $possessors\$$

The leafs $c_{q,k}$ in the sub-trees $\{\mathcal{T}_{v_p}|v_q \rightarrow v_p\} \setminus \{\mathcal{T}_{v_{q+1}}\}$ describe for any vertex $v_q, i \leq q < j$ on the path $v_i \rightarrow v_{i+1} \rightarrow \dots \rightarrow v_{j-1}$ from v_i to just before v_j , sub-strings equivalent to suffixes of the abstracted sub-string. They can be found within S at positions starting $\mathcal{P}_{v_p} = \{|l(v_q \rightarrow^+ c_{q,k})| + 1 \mid c_{q,k} \text{ is a leaf in the sub-trees } \{\mathcal{T}_{v_p}|v_q \rightarrow v_p\} \setminus \{\mathcal{T}_{v_{q+1}}\}\}$ and can be replaced for abstraction with calls to the $|l(v_q \rightarrow^+ v_j)|^{\text{th}}$ position of the abstracted sub-string.

If $v_i = v_j$ is chosen for each abstraction ($v_i \neq r, v_j$), then the abstractions are the same as for the corresponding suffix tree.

IV. IMPLEMENTATION

In this section, we describe our implementations of two post pass compactors for suffix tree and reverse prefix tree procedural abstraction.

A. Construction of the Trees

Our compactors are language independent by working with size optimized assembler code emitted by `gcc -Os -S`. For global compaction, all assembler files of a program are concatenated into a single assembler file. Label and procedure names have to be made unique by appending the file name.

To identify fragments for abstraction, we based one implementation on traditional suffix trees and the other implementation on reverse prefix trees. The suffix trees are constructed with Ukkonen's fast algorithm [9] by reading the assembler file top-down. The reverse prefix trees are also constructed with Ukkonen's fast algorithm, but by reading the assembler file bottom-up.

For constructing either tree, equivalence of instructions can be based on syntactic or semantic equivalence. Some operations `op`, like addition or multiplication, are commutative and compilers have for three-address code some

liberties in ordering operands, e.g. `op r_src1 r_src2 r_target` \equiv `op r_src2 r_src1 r_target`. Commutativity of many operations is not reflected in two address code, anymore. Semantic equivalences without liberties in ordering operands like increments, e.g. `inc r` \equiv `add 1 r r` or `add 1 r`, are less exploitable because (optimizing) compilers are often picking in the same context uniformly one choice over the other.³ Our implementations are based on semantic equivalence of instructions.

There are different ways of modeling procedure calls. A procedure might be either called via a register or immediate, and the return address might be either stored in a register or on the stack. IA32, for instance, calls a procedure immediate with the full 32 bit address and pushes the return address on the stack. This instruction, `call`, is five bytes in length. On the same architecture, a procedure returns to the caller via the one byte long `ret` instruction by popping off the previously pushed return address and jumping to this address. Alpha, on the other hand, uses the instruction `jsr r_ret, (r_proc)` to call procedures via a register (`r_proc`) and saves the return address in a further register `r_ret`. Before the call, the address of the procedure must be loaded by a sequence of instructions in register `r_ret`. As the return address is already in a register, a jump instruction `ret r_ret` is enough to return from the procedure call. On Alpha, all instructions are four bytes in length.

For procedural abstraction on *register-based architectures*, free registers need to be found or made available. The register for holding the return address must be not only the same register at each call site, but it must be also available over the whole abstracted fragment. This is not necessary for the register storing the address of the procedure. At each

³Contexts might be different for instructions at the beginning (end) of fragments, as some instructions before (after) the fragments are also included.

call site, this can be a different register available just for the actual procedure call.

The focus of our current implementation is on *stack-based architectures* not requiring any registers for procedure calls. For procedural abstraction on stack-based architectures, care must be taken when abstracting instructions accessing the stack. The call to an abstracted procedure pushes the return address on the stack and thereby modifying the stack. This might result within the abstracted procedure in instructions accessing wrong stack slots.⁴ Only instructions accessing the stack via the frame pointer can be safely abstracted because the frame pointer, initialized in each function prologue, is used only to address function arguments which have been pushed on the stack before the actual call. Hence calls to abstracted procedures cannot modify the part of the stack addressed by the frame pointer and it is safe to abstract such stack accesses. Other stack accesses can not be abstracted. This is done by enforcing push and pop instructions as well as instructions with the stack pointer as an operand to fail any equality test.

We do not abstract fragments with procedure calls, either. Arguments to procedures might be passed via the stack. If not all arguments are pushed within the abstracted procedure, then the return address will be located within the arguments on the stack. The called procedure accesses then the return address instead of an argument. The exclusion of procedure calls from fragments is done, as before, by enforcing call instructions to fail any equality test.

Single entry–single exit regions must be used as fragments for procedural abstraction [1], [2], [10]. As basic blocks fulfill this requirement trivially, we limit fragments up to the extent of single basic blocks. This is done the usual way by enforcing jump instructions and labels (which generally serve as targets for jumps) to fail any equality test.

Our compactors are highly architecture independent and can be easily retargeted to architectures which do not use registers for procedure calls. Only the abstraction and equality functions are architecture dependent. The equality function for testing instruction equality must be parametrized with architecture specific mnemonics for push, pop, jump, and procedure call instructions as well as symbolic names of the stack- and frame pointer. Optionally, semantic equivalences can be declared, as well. The abstraction function for replacing code fragments with procedure calls and for generating abstracted procedures must be parametrized with architecture specific mnemonics for procedure call and procedure return instructions. By default, the system assembler `/usr/bin/as` and the system linker `/usr/bin/ld` are called for analyzing and generating native code. Cross assemblers and cross linkers can be specified for analyzing and generating non-native code as well.

⁴This is also a problem on register-based architectures if a register is temporarily made available by storing it on the stack.

B. Selection of Sets

Not all sets identified by the tree algorithms of Sections III-A and III-B should or can be used for abstracting procedures. We discuss in this subsection which sets are discarded and which are selected for the actual abstraction.

1) *Benefit of Abstracting Sets*: An overhead is inherent in abstracting procedures. Procedure call instructions replace the original fragments and an additional return instruction is added to the abstracted fragment. If a set of fragments is identified by a pair of internal vertices $(v_i \neq \tau, v_j)$ on a path $\tau = v_0 \rightarrow^+ v_i \rightarrow v_{i+1} \rightarrow \dots \rightarrow v_{j-1} \rightarrow v_j, 0 < i \leq j$, then its benefit bf can be calculated by

$$\begin{aligned} \text{bf}((v_i, v_j)) &= -|l(\tau \rightarrow v_j)| - |\text{ret}| \\ &+ \sum_{\text{leafs}(\mathfrak{T}_{v_j})} (|l(\tau \rightarrow v_j)| - |\text{call}|) \\ &+ \sum_{q=i}^{j-1} \sum_{\text{leafs}(\{\mathfrak{T}_{v_p} | v_q \rightarrow v_p\} \setminus \{\mathfrak{T}_{q+1}\})} (|l(\tau \rightarrow v_q)| - |\text{call}|) \end{aligned} \quad (3)$$

or by

$$\begin{aligned} \text{bf}(\{s_1, \dots, s_m\}) &= -\max\left(\bigcup_{i=1}^m |s_i|\right) - |\text{ret}| \\ &+ \sum_{i=1}^m (|s_i| - |\text{call}|), \end{aligned} \quad (4)$$

where $\text{leafs}(\mathfrak{T}_v)$ is the set of all leafs of a tree \mathfrak{T}_v , $s_1 \dots s_m$ are the $m = |\text{leaf}(\mathfrak{T}_{v_i})|$ fragments identified by a pair (v_i, v_j) , and $|\text{call}|$ and $|\text{ret}|$ are the lengths of the call and return instruction.

Eqs. (3) and (4) can be simplified to Eqs. (5) and (6) for suffix tree abstractions as all fragments have the same length.

$$\begin{aligned} \text{bf}(v_i) &= -|l(\tau \rightarrow v_i)| - |\text{ret}| \\ &+ \sum_{\text{leafs}(\mathfrak{T}_{v_i})} (|l(\tau \rightarrow v_i)| - |\text{call}|) \end{aligned} \quad (5)$$

$$\begin{aligned} \text{bf}(\{s_1, \dots, s_m\}) &= -|s_1| - |\text{ret}| \\ &+ m(|s_1| - |\text{call}|). \end{aligned} \quad (6)$$

For some sets, the overhead may dominate, resulting in a non-positive benefit. Such *non-beneficial* sets are eventually discarded (see Section IV-B2).

A *necessary* condition for abstraction is the lengths of the fragments must be longer than the length of the function call instruction. On our target architecture IA32, instruction lengths range from one to 17 bytes. The length of the function call instruction is five bytes; one byte for the opcode and four bytes for the absolute address of the function. The length of the function return instruction is one byte. Table I is derived from Eq. (6) and shows for fragment lengths $|s_1|$ the required number m of fragments for a set $\{s_1, \dots, s_m\}$ to become beneficial. For this, short fragments must appear

quite often, possibly up to 8 times. Because of the relatively large sizes of some instructions, even single instructions can be abstracted if they are longer than five bytes and appear often enough.

Table I
PROCEDURAL ABSTRACTION BENEFIT

$ s_1 $	≤ 5	6	7	8	9	10	11	≥ 12
m	—	8	5	4	3	3	3	2

2) *Conflicting Sets*: Sets identified by the tree algorithms can overlap, either internally or with other sets. Such *conflicts* need to be resolved. Internally overlapping sets can be identified by comparing start and end instruction numbers of all the fragments within each set. Overlapping fragments within a set are discarded in order of smaller size. Next, non-beneficial sets are discarded. The remaining sets are sorted according to their benefit.

Overlapping sets are discarded by giving priority to more beneficial sets. This is achieved by screening the sets from highest to lowest benefit. A screened set is selected for abstraction if it does not conflict with previously selected sets and is discarded, otherwise.

Checking sets for overlaps becomes trivial if instructions of selected sets are marked in the assembler file. Then, the instructions of screened sets must be checked against the marks in the assembler file, only. If there are overlaps, then the screened set is discarded; otherwise it is selected for abstraction and its instructions are marked as well.

A suffix tree has exactly n leafs and between 1 and $n-1$ internal vertices. For suffix tree abstraction, a fragment is identified by an internal vertex $v_i \neq \tau$. Hence, there are between 0 and $n-2$ fragments to sort. For all vertices v_i , the length of the labels along the path $\tau \rightarrow^+ v_i$ as well as the number of leafs in the sub-tree \mathfrak{T}_{v_i} can be calculated by a single depth first traversal of the suffix tree, e.g. in linear time. If this information is stored in the vertices, then benefits can be calculated with Eq. (5) in constant time. This makes suffix tree abstraction an $O(n \cdot \log(n))$ algorithm.

A reverse prefix tree has exactly n leafs and between 1 and $n-1$ internal vertices. For reverse prefix tree abstraction, a fragment is identified by a pair of internal vertices $(v_i \neq \tau, v_j)$ on a path $\tau = v_0 \rightarrow^+ v_i \rightarrow^* v_j \rightarrow^+ v_{j,k} = c_{j,k}$ from τ to leafs $c_{j,k}$ in the sub-tree \mathfrak{T}_{v_j} . Not all such fragments need to be considered for abstraction. Consider a pair of internal vertices $(v_i \neq \tau, v_j)$ on the path $\tau = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_i \rightarrow v_{i+1} \rightarrow \dots \rightarrow v_{j-1} \rightarrow v_j, 0 < i \leq j$, keep the second vertex v_j fixed while vary the first vertex v_i over $1 \leq i \leq j$. Eq. (3) can be rewritten recursively as

$$\begin{aligned} \text{bf}((v_j, v_j)) &= -|l(\tau \rightarrow^+ v_j)| - |\text{ret}| & (7) \\ &+ \sum_{\text{leafs}(\mathfrak{T}_{v_j})} (|l(\tau \rightarrow^+ v_j)| - |\text{call}|) \end{aligned}$$

$$\text{bf}((v_i, v_j)) = \text{bf}((v_{i+1}, v_j)) \quad (8)$$

$$+ \sum_{\text{leafs}(\{\mathfrak{T}_{v_p} | v_i \rightarrow v_p\} \setminus \{\mathfrak{T}_{i+1}\})} (|l(\tau \rightarrow^+ v_i)| - |\text{call}|).$$

For a fixed v_j , bf becomes a function in i with the slope $\text{bf}((v_{i+1}, v_j)) - \text{bf}((v_i, v_j)) = -\sum (|l(\tau \rightarrow^+ v_i)| - |\text{call}|)$. The string along the path $\tau \rightarrow^+ v_i$ grows strictly monotone in i . The slope is positive for strings shorter than the call instruction, e.g. it is *not* beneficial to abstract them as they are too short; the slope is negative for strings longer than the call instruction, e.g. it is beneficial to abstract them. Hence, bf has a maximum at $v_{\hat{i}}$ with $\hat{i} := \min\{i \mid |l(\tau \rightarrow^+ v_i)| > |\text{call}|\}$. The vertices $v_{\hat{i}}$ can be computed in linear time on all paths from root to leafs. For a maximal benefit over all sets, it is not necessary to consider fragments $(v_i \neq \tau, v_j)$ other than those with $v_i = v_{\hat{i}}$ as the first component. There are between 0 and $n-2$ pairs $(v_i \neq \tau, v_j)$ describing sets of fragments for reverse prefix tree abstraction.

For calculating the benefits, each vertex v_j needs to store the length of the string from the edge $v_{j-1} \rightarrow v_j$ as well as the numbers of leafs of each sub-tree $\mathfrak{T}_{v_p}, v_j \rightarrow v_p$. As for suffix trees, this can be calculated in linear time. Benefits can then be calculated incrementally in linear time. For this, Eq. (3) needs to be rewritten recursively as

$$\begin{aligned} \text{bf}((v_i, v_i)) &= -|l(\tau \rightarrow^+ v_i)| - |\text{ret}| & (9) \\ &+ \sum_{\text{leafs}(\mathfrak{T}_{v_i})} (|l(\tau \rightarrow^+ v_i)| - |\text{call}|) \end{aligned}$$

$$\begin{aligned} \text{bf}((v_i, v_{j+1})) &= \text{bf}((v_i, v_j)) + |l(v_j \rightarrow v_{j+1})| & (10) \\ &+ \sum_{\text{leafs}(\mathfrak{T}_{v_{j+1}})} |l(v_j \rightarrow v_{j+1})| \end{aligned}$$

$$\begin{aligned} &= \text{bf}((v_i, v_j)) & (11) \\ &+ \sum_{1 + \text{leafs}(\mathfrak{T}_{v_{j+1}})} |l(v_j \rightarrow v_{j+1})|. \end{aligned}$$

With $i = \hat{i}$, the benefit of each pair $(v_i \neq \tau, v_j)$ can be calculated in a single breadth first traversal of the reverse prefix tree, e.g. in linear time. This makes reverse prefix tree abstraction an $O(n \cdot \log(n))$ algorithm, as well.

Our prioritization of overlapping sets considers only expected benefits of sets in isolation. This heuristic, originally proposed in [1] and also used in [2], is simple and can be implemented, as seen, in $O(n \cdot \log(n))$ time. Kim and Lee consider for prioritizing overlapping sets the impact on benefits of other sets as well. For this, they provide in [11] an $O(n^3)$ algorithm, which we consider for our implementation as too expensive.

3) *Generation of Abstracted Procedures and their Calls*: The selected sets are abstracted by creating new procedures at the beginning or end of the assembler file. These new procedures are representatives of each set with final return instructions.

For suffix tree abstraction, a representative of a set can be any fragment of the set, and for reverse prefix tree

abstraction, a representative must be any of the longest fragments.

All fragments of a set are then replaced by procedure call instructions to or into the abstracted procedure.

For suffix tree abstraction, the called instruction is, for any fragment, the first instruction of the abstracted procedure. For reverse prefix tree abstraction, the called instruction is for a fragment s_i the $\|s_{\max}\| - \|s_i\| + 1^{\text{st}}$ instruction of the abstracted procedure, where s_{\max} is one of the longest fragments of the set and $\|s_j\|$ is the number of instructions of fragment s_j .

V. EXPERIMENTAL RESULTS

In this section, we compare experimental results of suffix tree and reverse prefix tree procedural abstraction. We provide static analyses, e.g. an analysis of code size, number of abstracted fragments and procedures, and dynamic analyses, e.g. an analysis of the number of procedure invocations at run-time.

Our benchmarks consist of several programs from the MediaBench embedded systems benchmark suite [12] (cjpeg, djpeg, mpeg2enc, mpeg2dec, rasta, gsm (un)toast⁵, and ppp⁶).

Table II gives for IA32 statistics of optimizations with gcc -Os, gcc -Os + suffix tree abstraction, and gcc -Os + reverse prefix tree abstraction. We refer to such optimized programs as baseline code, suffix tree and reverse prefix tree optimized code. In columns 2, 3 and 5, we give the respective code sizes s_b , s_s , s_r and in columns 4 and 6 the code size reductions of baseline code to suffix tree optimized code $r_s = 100((s_b - s_s)/s_b)\%$ and reverse prefix tree optimized code $r_r = 100((s_b - s_r)/s_b)\%$. The last two columns compare both abstractions by giving for both reductions the difference $r_r - r_s$ and improvement $100(r_r/r_s)\%$.

Suffix tree abstraction reduces the code size by 1.379 to 7.971% with an average reduction of 3.419%. Large variations are not uncommon and were already observed by previous studies, notably [1], [2], [13]. For compacting various versions and configurations of the linux kernel, He et al. achieved on the same architecture (IA32) comparable results: with procedural abstraction in conjunction with whole function abstraction the code size was reduced between 1.34 and 4.98% [13].

Reverse prefix tree abstraction reduces the code size by 1.520 to 8.435% with an average reduction of 3.702%. Hence, by choosing reverse prefix trees over suffix trees, the

⁵As the two binaries toast and untoast are identical, providing functionality for *toasting* and *untoasting* files, we list them in the static analysis of tables II and III as one program and in the dynamic analysis of table IV as two programs.

⁶As the binary ppp provides functionality for encrypting and decrypting files, we list this program in the static analysis of tables II and III as one program and in the dynamic analysis of table IV as two programs ppp enc and ppp dec.

code size reduction can be improved by up to 13.387%, or 8.277% on average resulting in an *additional* code size reduction of up to 0.573%, or 0.283% on average.

Table III lists the number of abstracted procedures and the number of abstracted fragments for suffix tree as well as for reverse prefix tree abstraction and the percentage of procedures with multiple entry points as well as the percentage of fragments shorter than their abstracted procedures. In our benchmarks, between 320 and 1,180 fragments could be abstracted, resulting in 74 to 212 abstracted procedures. For reverse prefix trees, between 9.25 and 32.91% of all fragments have entry points *within* abstracted procedures and between 16.22 and 34.78% of all procedures have multiple entry points. For each benchmark, more fragments could be identified by reverse prefix tree abstraction than by suffix tree abstraction. The increase in abstracted fragments ranges from 3.78% for ppp to 11.37% for djpeg. The number of abstracted procedures does not increase in most cases as well. There was no change for mpeg2dec. There were decreases between 0.64% for ppp and 6.50% for mpeg2enc and only two increases of 1.94 and 2.26% for the two jpeg coders djpeg and cjpeg.

To understand the changes, we visualize the computational processes of our abstraction passes by drawing abstracted fragments in their original program. Fig. 4 gives the visualization of mpeg2enc.

Pixels represent single instructions in program order from left to right, starting in the upper left corner and wrapping around at the end of each line. In Figs. 4(a) and 4(b), gray pixels represent instructions that could be abstracted with suffix trees and reverse prefix trees, respectively. Fig. 4(c) shows the difference of both abstractions. Gray pixels represent instructions that could be abstracted with suffix trees as well as with reverse prefix trees. Black pixels represent instructions that could be abstracted with reverse prefix trees only, and light gray pixels represent instructions that could be abstracted with suffix trees only. There are more black pixels than light gray pixels, resulting in the higher code size reduction already observed in Table II.

Fig. 4(c) shows a lot of short black lines in isolation. Most represent newly emerging short fragments which join sets of larger fragments. This is again shown in Fig. 5(a): with suffix trees, two long fragments may be abstractable into one procedure p while with reverse prefix trees, further, small, fragments may be also abstractable into the same procedure. There are some gray lines preceded by short black lines, i.e. suffix tree abstracted fragments could be expanded by reverse prefix trees. This is because sets with fragments equivalent to suffixes of the longest fragment are shortened by suffix trees to fragments of the size of the shortest fragment. This is again shown in Fig. 5(b): with suffix trees, three long fragments may be abstractable into one procedure p, but with reverse prefix trees, two of them may be expandable towards the start. Suffix tree abstracted

Table II
CODE SIZE

program name	baseline code size s_b	suffix tree optimized code		reverse prefix tree optimized code		comparison	
		size s_s	reduction r_s	size s_r	reduction r_r	difference	improvement
cjpeg	95,708 bytes	92,324 bytes	3.536%	91,871 bytes	4.009%	0.573%	13.387%
djpeg	93,398 bytes	89,967 bytes	3.674%	89,518 bytes	4.154%	0.480%	13.087%
mpeg2enc	49,927 bytes	48,688 bytes	2.482%	48,610 bytes	2.638%	0.156%	6.295%
mpeg2dec	31,317 bytes	30,321 bytes	3.180%	30,261 bytes	3.372%	0.192%	6.024%
rasta	141,711 bytes	139,757 bytes	1.379%	139,557 bytes	1.520%	0.141%	10.235%
gsm (un)toast	26,673 bytes	24,547 bytes	7.971%	24,423 bytes	8.435%	0.464%	5.833%
pgp	127,786 bytes	125,598 bytes	1.712%	125,501 bytes	1.788%	0.076%	4.433%
mean			3.419%		3.702%	0.283%	8.277%

Table III
ABSTRACTED FRAGMENTS AND PROCEDURES

program name	number of abstracted fragments			number of abstracted procedures			multiple entry points	
	suffix trees	rev. prefix trees	increase	suffix trees	rev. prefix trees	increase	procedures	fragments
cjpeg	904	1,005	+11.11%	209	212	+2.26%	32.08%	25.07%
djpeg	906	1,009	+11.37%	206	210	+1.94%	31.43%	25.27%
mpeg2enc	512	550	+7.42%	123	115	-6.50%	33.91%	24.18%
mpeg2dec	320	335	+4.69%	74	74	$\pm 0.00\%$	16.22%	9.25%
rasta	877	945	+7.76%	198	190	-4.04%	32.63%	32.91%
gsm (un)toast	561	597	+6.42%	98	92	-6.12%	34.78%	26.97%
pgp	1,137	1,180	+3.78%	157	156	-0.64%	26.92%	24.41%
mean			+7.51%			-2.52%	29.71%	24.01%

fragments may be also split by reverse prefix trees for joining other sets resulting in some light gray lines.

The decrease of the number of abstracted procedures is due to merges of procedures. Two suffix tree abstracted procedures p_1 and p_2 can be merged into one procedure p_{12} abstracted by reverse prefix trees if one is a suffix of the other. An example is given in Fig. 5(c). Merges may come with further optimizations as well. Some fragments of the shorter set may expand as some instructions preceding such fragments are equivalent to corresponding instructions of the longer fragments (Fig. 5(d)). Fragments may also be shortened for a merge to become more beneficial (Fig. 5(e)).

Table IV gives the number of invocations of all abstracted procedures together. The increase in abstracted procedure invocations is not, as we expected, linear in the increase in abstracted fragments and appears purely random.

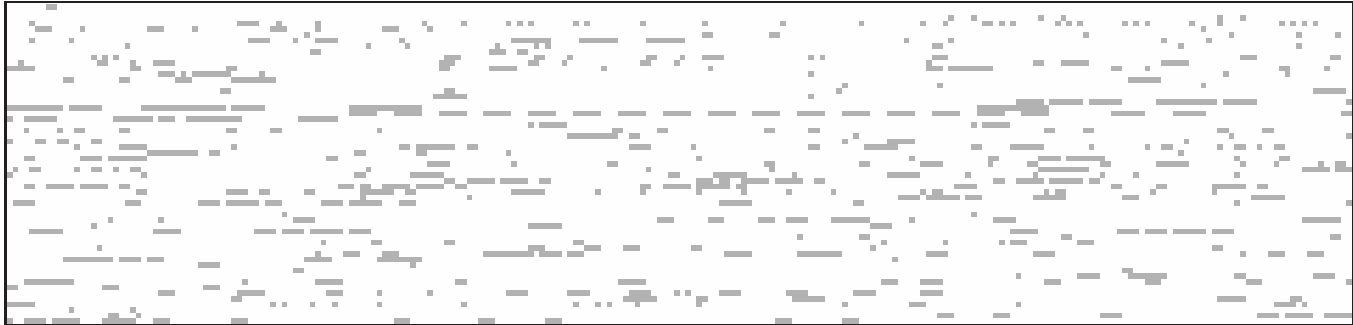
A detailed analysis of individual procedure calls shows their numbers of invocations varies greatly (Fig. 6). Some of the newly abstracted fragments lie in *hot spots* contributing overproportionally to the number of abstracted procedure invocations. For *gsm untoast*, for instance, the difference of $8,115,097 - 5,315,870 = 2,799,227$ executed calls is mainly due to the two additional call instructions with an execution frequency of 1,180,160 each, and so can the discrepancies of the other programs explained. Hence,

Table IV
DYNAMIC COUNT OF ABSTRACTED PROCEDURE INVOCATIONS

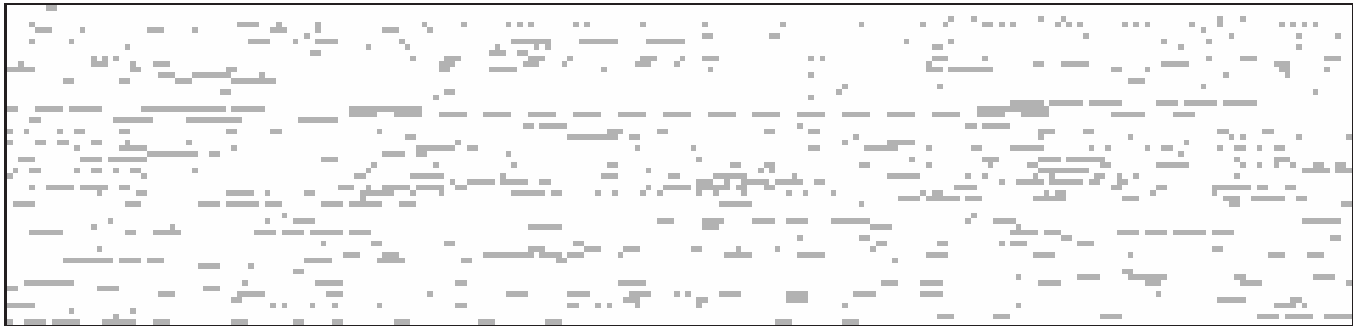
program name	suffix trees	reverse prefix trees	increase
cjpeg	286,679	398,780	+39.10%
djpeg	145,845	152,814	+4.78%
mpeg2enc	12,125,093	12,112,953	-0.10%
mpeg2dec	627,281	767,676	+22.38%
rasta	46,758	50,983	+9.04%
gsm toast	4,166,549	4,633,745	+11.21%
gsm untoast	5,315,870	8,115,097	+52.66%
pgp enc	371,669	482,222	+29.75%
pgp dec	344,970	439,449	+27.39%
mean			+21.80%

excluding a few frequently executed fragments from abstraction via reverse prefix trees results in an execution count of the same order as the execution count of abstraction via suffix trees.

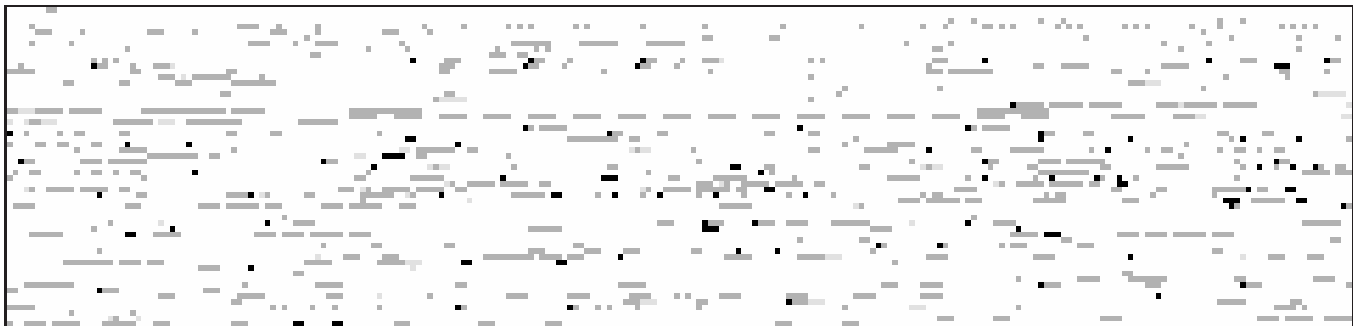
De Sutter et al. have shown in [14] that profiling based on simple basic block counts can be used to keep almost the full code size reduction while significantly reducing the number of abstracted procedure invocations. This is true for suffix tree abstraction as well as reverse prefix tree abstraction.



(a) Suffix Tree Abstraction



(b) Reverse Prefix Tree Abstraction



(c) Difference of both Abstractions

Figure 4. Visualization of Abstracted Fragments for mpeg2enc

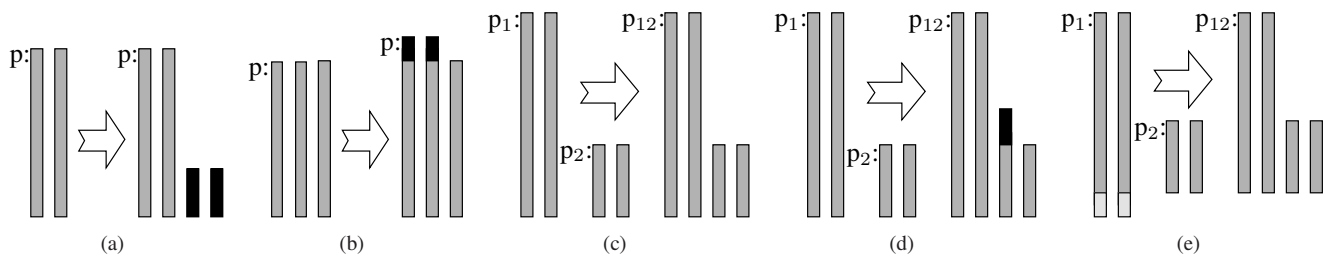


Figure 5. Abstraction Patterns of Suffix Trees and Reverse Prefix Trees

VI. FUTURE WORK

We use reverse prefix trees for identifying fragments equivalent to suffixes of the longest suffix in $O(n \cdot \log(n))$ time. With our notion of equivalence, e.g. semantic equivalence of instructions, we can improve code size reductions

up to 13.387% over standard suffix trees with an average improvement of 8.277%.

We would like to see how reverse prefix trees improve code size reduction for the procedural abstraction variants discussed in the related work section.

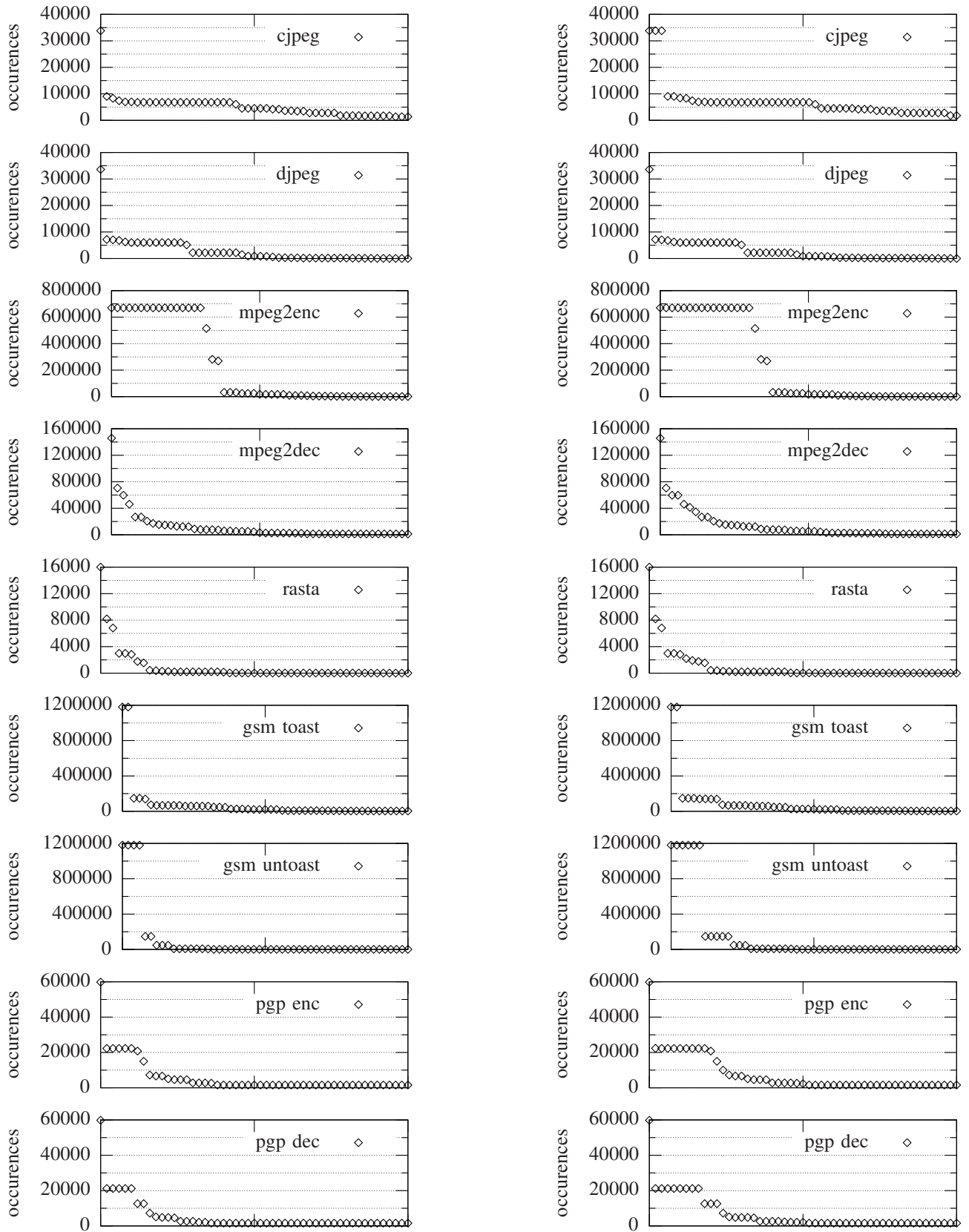


Figure 6. Dynamic Execution Count of the 50 most executed Procedure Call Instructions. Left for Suffix Tree and right for Reverse Prefix Tree Abstraction

Fragments that differ only in register names can be rendered equivalent with suffix trees as discussed in [2]. Fragments that differ in constants can be rendered equivalent by substituting a register for the constant and assigning the constant to the register before the actual call to the abstracted procedure.⁷ For identifying such fragments, standard suffix trees can be used as discussed by Cooper and McIntosh or parametrized suffix trees as described by Baker in [15]. Because suffix trees are used to identify such fragments, reverse prefix trees can be used to identify fragments, as well.

The algorithm suggested by Dreweke et al. for reordering instructions is based on graph mining algorithms [3]. As their algorithm is not based on suffix trees, reverse prefix trees cannot be directly used. Nevertheless, it can be extended to identify fragments equivalent to suffixes of a longest fragment.

VII. CONCLUSION

For memory constrained environments like embedded systems, optimization for size may be crucial. A common technique for compacting code is procedural abstraction. Usually, suffix trees are used for identifying fragments, resulting in fragments of the same length. In this paper, we have presented an algorithm based on reverse prefix trees to identify fragments of different lengths, resulting in more and longer fragments.

With reverse prefix trees, (in the worst case) equal or higher code size reductions can be achieved than with suffix trees. Our benchmarks have shown improvements of up to 13.387% with an average improvement of 8.277%. The computational complexity is in both cases $O(n \cdot \log(n))$ and implementations are very similar in nature. Hence, we recommend for procedural abstraction the use of reverse prefix trees over suffix trees.

ACKNOWLEDGMENT

The authors would like to thank Norman Rubin for his help on improving the paper.

REFERENCES

- [1] C. W. Fraser, E. W. Myers, and A. L. Wendt, "Analyzing and compressing assembly code," in *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*. New York, NY, USA: ACM Press, 1984, pp. 117–121.
- [2] K. D. Cooper and N. McIntosh, "Enhanced code compression for embedded RISC processors," in *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 1999, pp. 139–149.
- [3] A. Dreweke, M. Worlein, I. Fischer, D. Schell, T. Meinl, and M. Philippsen, "Graph-based procedural abstraction," in *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 259–270.
- [4] S. Liao, S. Devadas, and K. Keutzer, "Code density optimization for embedded DSP processors using data compression techniques," in *Proc. Conf. on Advanced Research in VLSI*, 1995.
- [5] T. Gyimóthy, R. Ferenc, G. Lehotai, A. Kiss, and A. Bicsak, "US patent nr. 7,293,264: Method and a device for abstracting instruction sequences with tail merging," Patent, 2005.
- [6] D. Gusfield, *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge Univ. Press, 1997.
- [7] P. Weiner, "Linear pattern matching algorithm," in *14th Annual IEEE Symposium on Switching and Automata Theory*, Washington, DC, 1973, pp. 1–11.
- [8] E. M. McCreight, "A space-economical suffix tree construction algorithm," *J. ACM*, vol. 23, no. 2, pp. 262–272, 1976.
- [9] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.
- [10] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, pp. 378–415, 2000.
- [11] D.-H. Kim and H. J. Lee, "Iterative procedural abstraction for code size reduction," in *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*. New York, NY, USA: ACM, 2002, pp. 277–279.
- [12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," in *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 330–335.
- [13] H. He, J. Trimble, S. Perianayagam, S. Debray, and G. Andrews, "Code compaction of an operating system kernel," in *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 283–298.
- [14] B. De Sutter, H. Vandierendonck, B. De Bus, and K. De Bosschere, "On the side-effects of code abstraction," in *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*. New York, NY, USA: ACM, 2003, pp. 244–253.
- [15] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 1995.

⁷This can be used to abstract fragments that differ in register names, as well.