An Offline Approach for Whole-Program Paths Analysis using Suffix Arrays

G. Pokam and F. Bodin

IRISA
Campus Universitaire de Beaulieu
35042, Rennes Cedex, France
{gpokam,bodin}@irisa.fr

Abstract. Software optimization techniques are highly reliant on program behavior to deliver high performance. A key element with these techniques is to identify program paths that are likely to achieve the greatest performance benefits at runtime. Several approaches have been proposed to address this problem. However, many of them fail to cover larger optimization scope as they are restricted to loops or procedures. This paper introduces a novel approach for representing and analyzing complete program paths. Unlike the whole-program paths (WPPs) approach that relies on a DAG to represent program paths, our program trace is processed into a *suffix-array* that can enable very fast searching algorithms that run with time $O(\ln(N))$, N being the length of the trace. This allows to process reasonable trace sizes offline, avoiding the high runtime overhead incurred by WPPs, while accurately characterizing hot paths. Our evaluation shows impressive performance results, with almost 48% of the code being covered by hot paths. We also demonstrate the effectiveness of our approach to optimize for power. For this purpose, an adaptive cache resizing scheme is used that shows energy savings in the order of 12%.

1 Introduction

The increasing processor complexity makes the optimization process a compelling task for software developers. These latter usually face the difficult problem of predicting the impact of a static optimization at runtime. One approach used to meet this challenge is to rely on path profiling to collect statistics about dynamic program control flow behavior. While this has proved to be very effective to assist program optimization [12, 20], the way this information is recorded fails to reveal much insight about dynamic program behavior. One main concern with current path profiling techniques is that they are often restricted to record intra-procedural paths only [4].

More recently, Larus [14] has proposed an efficient technique for collecting path profiles that cross procedure boundaries. In his proposed approach, an input stream of basic blocks is compacted into a context-free grammar using SEQUITUR [16] to produce a DAG representation of a complete program. SEQUITUR, however, is a compression algorithm that proceeds online; hence, the

grammar production rules are far from being minimal such that in practice, the achieved compression ratio is likely to incur a high runtime overhead. In addition, as each grammar rule is processed into a DAG, the information pertaining to a particular dynamic path is lost since all dynamic instances of a given path are fused into a unique DAG node.

In this paper, we propose to collect and analyze whole-program paths offline. In this way, we make it possible to manage reasonable trace sizes, while shifting the cost of online processing off-line. Since, however, the relatively large sizes of the trace may render the path analysis cumbersome, an approximation of the trace is needed which also can enable efficient path analysis techniques. We introduce a novel program trace representation to deal with path analysis in an efficient way. In particular, our approach stems from the fact that the data retrieval nature of the path analysis problem makes it very tempting to consider pattern-matching algorithms as a basis for path identification. One such approach is given by *suffix-arrays* [15], which have already proved to be a very efficient data structure for analyzing biological data or text. Conceptually, looking for DNA sequences in biological data, or patterns in a text, is an analogous problem to searching for hot paths in a trace; thus making *suffix-array*-based searching techniques appropriate for path analysis. In addition, in contrast to a DAG representation, a *suffix-array* provides the advantage of treating each dynamic sub-path differently from one other.

This paper makes two contributions. The first and the foremost contribution of this paper is to demonstrate the effectiveness of using *suffix-array*-based techniques for analyzing hot program paths. More specifically, we show the appropriateness of suffix arrays to represent program paths, and to identify and characterize the exact occurrences of hot sub-paths in a trace. One particular strength of suffix arrays which make them very attractive for this purpose is their low computational complexity, which usually requires $O(\ln(N))$ time, N being the length of the input trace. The second contribution of this paper is to indeed illustrate the effectiveness of the proposed path analysis technique to guide power-related compiler optimizations. For this purpose, an adaptive cache resizing strategy is used and its potential benefits evaluated at the hot program paths frontiers.

The remainder of this paper is organized as follows. Section 2 introduces the background on suffix arrays. The profiling scheme used to collect paths is described in Section 3. In Section 4, we introduce the offline algorithm used to identify the sequences of basic blocks that appear repeated in the trace, while in Section 5 we show how these sequences can be qualified as hot paths. In Section 6, we present our experimental results and discuss a practical application of our scheme to reducing power consumption. Related work is presented in Section 7, while Section 8 concludes.

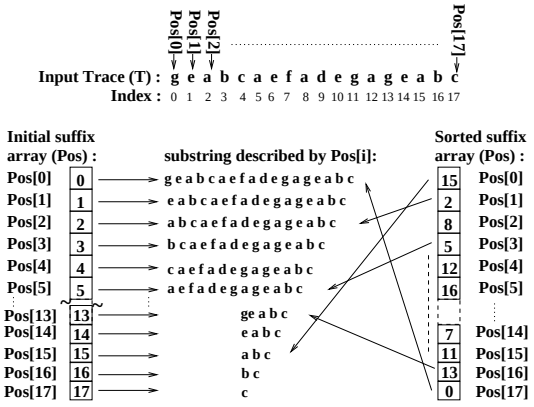

Fig. 1. Processing of an input trace T into an initial suffix array Pos.

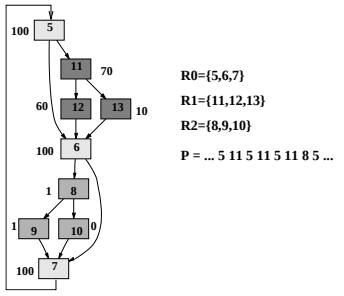

Fig. 2. Example of a sub-path.

2 Suffix arrays background

Suffix arrays have been intensively used in several research areas such as genome analysis or text editing to look for DNA or text patterns. However, despite their widespread use in these domains, we are not aware of any attempt to use this technique in the context of program path analysis. In this section, we briefly introduce the background of suffix arrays and discuss why they may be an efficient data structure for analyzing a whole-program trace.

Given an N-length character string S, the suffix array of S, denoted by Pos in the remainder of this paper, is defined as the sorted array of the integer indices corresponding to all the N suffices of S. Hence, $Pos[i]$ denotes the string starting at position i in S which extends until the end of the string. Figure 1 illustrates a simple example representing a program execution trace T which is first processed into an initial suffix array data structure and then sorted according to a lexicographical ordering.

One key characteristic of suffix arrays is that they can enable computation of search queries in time complexity $O(\mathrm{p} + \ln(N))$, where p is the length of the searched pattern and N the length of the string; making it very convenient to implement very fast searching algorithms. The query computation principally undergoes a binary search phase on the sorted suffix array, taking advantage of the fact that every substring is the prefix of some suffix. In addition to matching the searched query, suffix arrays also permit to compute the frequency of the queried pattern along with the exact positions of all of its occurrences in the string. As for instance, in the given example shown in Figure 1, the basic block sequence *geabc* appears 2 times in the trace, respectively at position $i = 16$ and $i = 17$ in the sorted suffix array Pos. By generalizing this concept on variable length substrings, several interesting items of information pertaining to dynamic program behavior can be efficiently retrieved. These include, for instance:

1. finding the longest repeated sequence of basic blocks in a trace, $lmax$;
2. finding all n-length repeated sequences of basic blocks in a trace, $n \leq lmax$;
3. determining the distribution frequency of each specific n-length basic blocks sequence in a trace;
4. identifying the positions of each different n-length basic blocks sequence in a trace.

Many of the above items may be of interest for several program optimizations. For instance, item 4 can be used for grouping hot sub-paths together to drive the formation of ILP regions. In addition, if two neighbor hot paths have associated distinct dynamic profiles, this information can be used to decide if their respective profile can be merged or not. This may be helpful for inferring a common configuration to adjacent hot paths in case of an adaptive compilation strategy scheme.

Although suffix arrays present very interesting properties regarding program path analysis, they still have some drawbacks. The most noticeable of them is the memory space required to construct the suffix array, which is linear with the size of the processed trace. This latter issue has since been the subject of intensive studies and some compression algorithms have already emerged that significantly reduce the amount of memory space required [10]. While this work can also be accommodated with such a compression scheme, this is not our main concern in this study.

3 Profiling scheme

In this section, we describe our general profiling scheme. In particular, we describe how the whole-program path trace is collected, and what kind of dynamic information is recorded together with the trace.

3.1 Collecting the trace

Profiling can be used in a straightforward manner to collect a whole-program trace by instrumenting each basic block of the CFG. This approach is however very costly in terms of memory space. A more efficient approach is to instrument only a subset of the executed basic blocks to capture nearly the same amount of control flow information.

We use Ball's definition of a *strong region* [5] to reduce the amount of basic blocks that need to be instrumented. Given a directed control flow graph, CFG, with nodes set V and edges set $E \subseteq V \times V$, a *strong region* identifies the set of basic blocks S in which any two nodes $v, w \in S$ occur the same number of time in any complete control flow path. Strong regions are actually computed as part of the loop analysis phase. The computation relies on a generalized notion of dominance information to identify nodes of a *strong region*. Simply stated, given a loop region L with entry node h, set of exit nodes E and set of backedge sources B, two nodes $(v, w) \in L$ are in the same *strong region* iff:

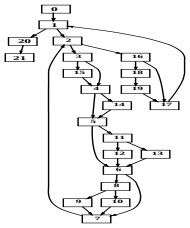

Fig. 3. CFG.

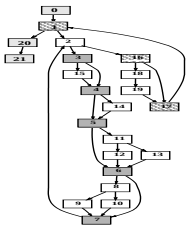

Fig. 4. CFG with three strong regions.

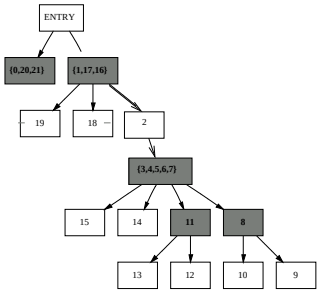

Fig. 5. CDG with instrumented nodes.

$h = loophead(v) = loophead(w)$ and (v **dom** w, w **pdom** v *with respect to* $B{+}E$)

The definition of the *strong region* given above allows us to identify regions in the CFG in which all basic blocks execute with nearly the same dynamic frequency, whatever the taken control flow path is. In this respect, any node belonging to such a region can be used to capture the dynamic control flow path induced by the other nodes of that region. This drastically reduces the number of instrumented basic blocks, as illustrated in Figure 4. The figure shows three strong regions, two of them are composed of 3 basic blocks while the last one has 4 nodes. The amount of instrumented basic blocks reduces from 22 to 14 because in each strong region we need to instrument only one basic block.

In some cases, however, the remaining number of instrumented basic blocks can still be large. Of most concern here are the strong regions composed of a single-node. Figure 4 shows for instance that these regions can represent almost half of the nodes. To reduce this number further, we also consider the control dependence relation [8] on the CFG induced by the strong regions. The idea is to reduce the number of such instrumented single-node regions by selecting only those that are actually control condition block. In this way, we can make sure that the number of instrumented regions with one node get reduced as we only need to keep track of the execution frequency of single-node regions of same control condition rather than of the execution count of each such individual region. This is illustrated in Figure 5. As shown in the example, the number of instrumented nodes reduces from 22 to 5 overall, providing up to 80% reduction of the total number of instrumented basic blocks.

We applied another compression technique to further reduce the size of the trace. This technique principally targets cyclic regions such as loops in which basic blocks execute repeatedly. In such a case, it is not necessary to record all the back-to-back dynamic occurrences of the same region. Instead of that, we can choose to record in the trace the last such dynamic occurrence with all attached information updated accordingly. Combined with our profiling scheme, this shows a real improvement in the compression ratio, typically up to 47%, on average, for our benchmark sets.

3.2 Control-flow information accuracy

The profiling approach presented in the previous section makes it difficult to rebuild a copy of the control flow path. This, however, is of less a concern since we are merely interested of knowing which *regions* are executed more often than others, rather than knowing exactly the execution count of each basic block. Such a region-directed path profiling approach is at the advantage of the program optimizer since it may permit him to focus the analysis only on the predominant paths in the trace. In an another phase, however, each *region* can be investigated more closely to identify individual hot basic blocks.

Consider for instance the example shown in Figure 2. Three *regions* are identified: the strong region $R0$ and the regions of same control condition $R1$ and $R2$. Assuming that only nodes 5, 11 and 8 get executed in each of these regions respectively, the corresponding dynamic execution trace is shown with name P in the figure. This simple example indicates that sub-path $5, 11$ is predominant. In the figure, we also show the cumulated execution count of each node. It is then straightforward to derive from the sub-path $5, 11$ the exact set of the most representative basic blocks by excluding those which execute less frequently, i.e. node 13. In our abstraction, nodes $12, 13, 9, 10$ are subsumed by the control dependence relation. While this effectively reduces the space, it also emphasizes the rapid identification of the main sub-path $5, 11$. This is central to our program sub-path detection technique.

3.3 BBWS signature

When a sequence of basic blocks appears repeated in the trace, we denote by *basic block working set* (BBWS) the set of static basic blocks that constitutes this sequence. The annotation attached to each such sequence is called a basic block working set signature. This annotation can be used to describe such a sequence in a unique manner, depending on the kind of dynamic information that is appended to it. For our experiments, we have considered the region id, *reg-id*, which identifies each instrumented *region*, the performance parameters *cyc, dyn, dmiss, imiss* which represent the number of elapsed cycles, the dynamic instructions count, the number of data and instruction cache misses attached to each region, respectively. This information will become more apparent during the formation of the hot program sub-paths, to determine the pertinence of a candidate hot region.

4 Identifying BBWS

The key idea to search for BBWS is to rely on the suffix array data structure to implement an efficient suffix sorting algorithm. We employ an adapted version of the KMR [13] algorithm used in genetic and text querying systems to achieve this.

4.1 KMR algorithm

The KMR (for Karp, Miller and Rosenberg) algorithm is a well known algorithm for computing the occurrence of repeated patterns in a string. The idea is dictated by the observation that each suffix in P can be defined as the k-length suffix of another suffix starting at position i. This implies that, at the j-th stage of the sorting algorithm, $j \geq 1$, the suffix array indices $i + 2^{j-1}$ computed at stage $j-1$ are used to initially sort each suffix i obtained at stage j. This technique allows to double the suffix length at each stage, requiring only $O(\log(N))$ processing time. The ordering relation used in the KMR sorting algorithm is based on the definition of an equivalence relation over the suffix positions of the path P. Given a path $P = p_1 p_1 p_2 ... p_n$, two suffices starting at positions i and j in P are said to be k-equivalent, $k < n$, denoted by $i \ E_k \ j$, if and only if the path of length k starting at these positions are the same.

4.2 Sorting algorithm description

We can easily make an analogy between the suffix array $Pos_k^{(j)}$ obtained at the j-th stage of the sorting algorithm described in the previous section and a partition of all E_k equivalent integer indices obtained from $Pos_k^{(j)}$, k being the length of the expanded suffix at that stage. Interestingly, the number of elements in the partition gives the actual number of BBWS of length k, whereas their integer indices in the suffix array $Pos_k^{(j)}$ gives their position in the trace P. Hence, it becomes straightforward to identify a BBWS according to its size (i.e length k), its dynamic frequency of occurrence (i.e cardinal of the partition E_k) as well as its dynamic coverage time (i.e start position in the trace until the position where a new BBWS is encountered).

The algorithm used to sort the suffix array Pos is shown in Algorithm 4. The alphabet is composed of the set of *regions* encountered in each CFG. The input search space P represents the execution trace. In line 6 of the algorithm, we first build the partition E_n corresponding to the set of BBWS with maximal repeated occurrence of length n. As each element of the partition is identified, it is hashed into a table of BBWS partitions with the hash key featuring the length of the BBWS. This is done for E_n as well as for the other partition elements used to iteratively compute it (see Algorithm 3). In lines 8-10 of the algorithm, the program terminates as soon as the set of BBWS identified so far is representative enough of the whole trace P. This issue is addressed in the next section. If this is not the case, i.e. the set of BBWS is not representative enough of the trace, then the algorithm undergoes a binary search to look for other BBWS, as shown in lines 11-19. The idea is to incrementally add new BBWS until the condition of the trace representativeness is met. At the end of the algorithm, the partition table T contains, for each valid entry k, the set of all k-length BBWS that appear repeated in the trace. The processing time for this algorithm is quite feasible. For instance, a 40MB trace size requires less than a few minutes to process, whereas for trace sizes ranging from several hundreds of MB to a GB, the processing time is within the order of hours.

Algorithm 1 Initialization

Require: P : control flow path defined over Σ^m
1: Construct the suffix array $Pos_{k=1}^{(0)}$
2: Add class elements E_1 to $T[1]$

Algorithm 2 Construct suffix array $Pos_k^{(j)}$ from $Pos_{k'}^{(j-t)}$

1: **repeat**
2: Use $Pos_{k'}^{(j-t)}$ to construct $Pos_{k'+1}^{(j-t+1)}$
3: Add class elements $E_{k'+1}$ to $T[k'+1]$
4: $k' := k' + 1$
5: **until** $k' < k$

Algorithm 3 Construct suffix array $Pos_{k=max}^{(N)}$ and E_{max}, N number of processing steps

1: Use Algorithm 1 to initialize the suffix array $Pos_{k=1}^{(0)}$
2: **repeat**
3: $r = r' + 2^{j-1}$
4: Construct $Pos_{k=r}^{(j)}$ from $Pos_{k=r'}^{(j-1)}$
5: Add class elements E_r to $T[r]$
6: **until** $Pos_{k=r}^{(j)}$ is unchanged
7: **return** r

Algorithm 4 Basic block working set partitioning

1: n, k : Integer := 0
2: T : BBWS partition table
3: Σ := {Set of reg-id}, $|\Sigma| = m$
4: $P : p_1\ p_2\ p_3\ ...\ p_m \in \Sigma^m$
5:
6: Use Algorithm 3 to suffix sort the array Pos, obtaining n, the longest repeated BBWS
7:
8: **if** all BBWS are representative of P **then**
9: stop here
10: **end if**
11: **for** $k = n - 1$ to 2 **do**
12: **if** $T[k]$ is empty **then**
13: Find $T[d]$ such that $d < k$, d is a power of 2 and $T[d] \neq \emptyset$
14: Use Algorithm 2 to iteratively construct E_k from E_d
15: **end if**
16: **if** all BBWS are representative of P **then**
17: stop here
18: **end if**
19: **end for**

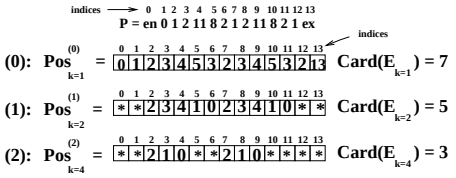

Fig. 6. Example of BBWS identification

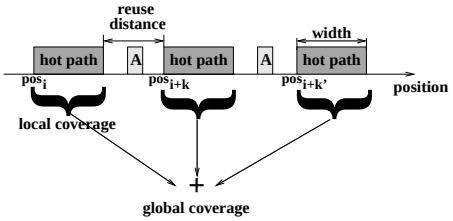

Fig. 7. Hot path characteristics.

4.3 Sorting example

Let us consider the example shown in Figure 6. We illustrate next the different processing steps involved when searching for the longest repeated BBWS. Step 0 shows the suffix array $Pos_{k=1}^{(0)}$ that corresponds to the initial sorting stage with suffix length $k = 1$. The partition elements of the equivalent class $E_{k=1}$ are deduced directly from $Pos_{k=1}^{(0)}$. The cardinal of the partition gives the number of BBWS of length $k = 1$. At the next iteration step, the array $Pos_{k=2}^{(1)}$ is computed from the array $Pos_{k=1}^{(0)}$ in the following way. The suffix positions of P that correspond to the integer indices in $Pos_{k=1}^{(0)}$ are sorted into buckets of same equivalent class. For instance, suffix positions $2, 7, 12$ in P will belong the same bucket since their integer indices in $Pos_{k=1}^{(0)}$ are identical (i.e. 2). Note that, at this stage, the number of elements in each bucket yields the dynamic execution frequency of the considered BBWS in P. From here, each bucket is sorted according to the b-equivalent relation, where $b = k^{(j)} - k^{(j-1)}$, i.e. $b = 1$ at stage 1. The result of this sorting is a new set of buckets where two suffix positions belong the same bucket iff they are E_b equivalent with regard to their integer indices in $Pos_{k=1}^{(0)}$. For instance, suffix positions $2, 7$ belong the same bucket because they are E_1 equivalent with respect to $Pos_1^{(0)}$. The array $Pos_k^{(j)}$ is obtained by renumbering the integer indices of the suffix positions contained in a bucket list with a same equivalent class number if they satisfy to the b-equivalent relation. Note that BBWS that appear only once are systematically discarded from the suffix array since we are only interested in identifying those that appear at least twice in the trace. This explains the stars in the arrays $Pos_{k=2}^{(1)}$ and $Pos_{k=4}^{(2)}$.

5 Qualified BBWS for hot sub-paths

Not all BBWS that are identified with the algorithm described in the previous section are of interest. Of course, there are some BBWS that effectively appear repeated in the trace but which inherently bring no value for the optimization. To distinguish among the BBWS those who are the most representative, we apply three selection criteria, as illustrated in Figure 7.

The first criterion is the *local coverage*. This metric is an indication of the number of elapsed cycles in the region, or the dynamic instructions count of that region, before a transition to another region occurs. Either one of the number of cycles or the dynamic instructions count can be directly obtained from the trace, as indicated in Section 3.3.

The second criterion is the *global coverage*. This metric is related to the *local coverage* by the dynamic execution frequency of a BBWS, $global\_coverage = frequency \times local\_coverage$. With respect to the overall program execution, this metric assigns to each potential hot path a global cycle weight or a dynamic instructions count weight.

The last criterion is the *reuse distance*, measured in number of dynamic basic blocks. The reuse distance is an approximation of the temperature of a BBWS. As the reuse distance gets larger, the probability that the underlying BBWS is a hot path lowers. This can be mainly attributed to the fact that, although the BBWS appears repeated in the trace, it is not too often executed to infer a hot temperature. In contrast, tighter reuse distances indicate a high probability that the considered BBWS is a hot path. A consequence of this is that cold blocks in the vicinity of a hot path may also be inferred a hot temperature since the heat may propagate to them indirectly. In Figure 7 for instance, if the reuse distance of the highlighted hot path is below a given threshold, block A can be included in the BBWS induced by the nodes of the hot path to form a coarser region. Assuming *Position* designates the set of all consecutive, non-overlapping positions of a BBWS in the trace, the average reuse distance $\overline{D}$ is computed as shown in Equation (1), where *width* refers to the size of the BBWS (number of basic blocks) and % represents the modulo function.

$$\overline{D} = \frac{\sum (pos_{i-1} + width) \ \% \ pos_i}{|Position|} \tag{1}$$

Note that, since a *region* is represented with a single basic block, the computation of $\overline{D}$ must actually consider all basic blocks encountered in that *region*. Hence, the expression of $\overline{D}$ provides only an approximation of the reuse distance value. A hot path candidate is then formed by selecting BBWS with a relatively high local coverage and low reuse distance. The global coverage serves as an indication of the hot paths weight in the program.

6 Experimental evaluation

This section presents an evaluation of the proposed approach. We first introduce the simulation platform and the benchmarks used in Section 6.1. Then, in Section 6.2, we evaluate and discuss our results.

6.1 Experimental methodology

We conducted our experiments using applications collected from MiBench [11] as illustrated in Table 1. The applications are first compiled with the PISA compiler

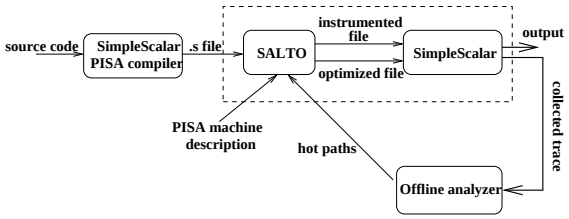

Fig. 8. Simulation framework.

Table 1. Benchmark

Bench.	size (MB)	compr. ratio
dijkstra	110	65%
adpcm	148	67%
bf	55	74%
fft	6	85%
sha	11	77%
bmath	6	78%
patricia	21	77%

Table 2. Machine parameters.

Issue	in-order 4-issue
Integer ALU	4
Mult. units	2
Load/Store unit	1
Branch unit	1
instr. cache	32K 1-way
data cache	32K 4-way
cache access latency	1 cycle
data cache replacement policy	LRU
memory access latency	100 cycles

from the SimpleScalar [7] tool suite, with optimization level 3, to obtain an input assembly file. Each assembly file is then processed by SALTO [6], which is a general, compiler-independent tool that makes the manipulation of the assembly code at the CFG level easier. SALTO is used essentially to instrument the code, using the SimpleScalar annotation feature, and to add new compiler optimization passes. The produced executable is processed by SimpleScalar to extract the compressed trace which is then fed to the offline analyzer. After the hot paths have been identified, this information can be re-injected into SALTO to drive the various compiler-dependent optimization passes. An overview of the different processing stages is shown in Figure 8.

Our measurements were performed with SimpleScalar, which we use to model a 5-stage in-order issue processor such as those encountered in the embedded computing domain, e.g. the Lx processor [9]. Details on the processor configuration parameters used in this study are shown in Table 2.

6.2 Evaluation

This section presents the evaluation results of using our scheme on the set of benchmarks described in the previous section. The evaluation consisted to measuring the relative compression ratio achieved by our approach and to analyzing the quality of the detected hot paths with respect to the criteria introduced in Section 5.

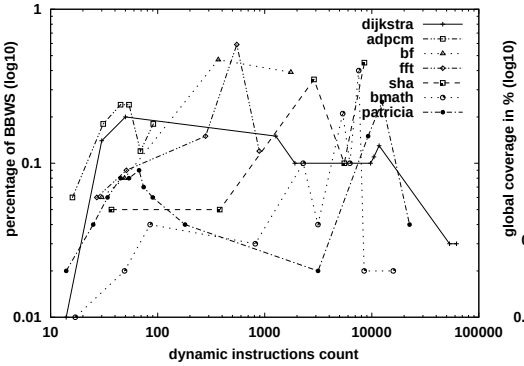

Fig. 9. Local coverage.

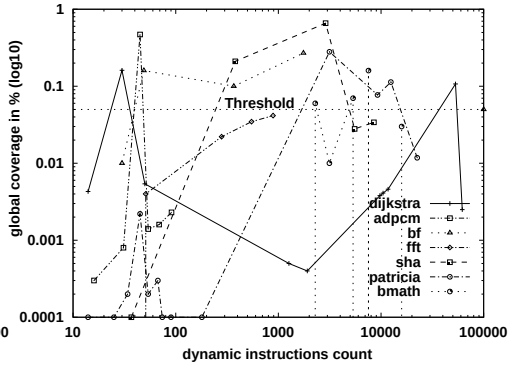

Fig. 10. Global coverage.

Trace size. The last column of Table 1 gives an estimate of the compression ratio achieved with our approach. Note that the size of the trace depends strongly on the information encoded with each trace line. For this experiment, we used 20 bytes for each trace line, one byte each for recording the region id, the number of cycles, the number of dynamic instructions, the number of data and instruction cache misses associated with each region respectively. More elaborate trace line representations can be imagined to reduce further the trace size; however, this is not the scope of this paper. The column labeled *trace size* shows the original size of the trace. As it can be seen from the table, the trace size can be reduced by up to 74% on average. This compression ratio includes the compaction of back-to-back occurrences of loop paths (see Section 3.1), which accounts for about 47% of the trace reduction.

Local coverage. This metric measures the time spent in a BBWS, or the number of dynamic instructions executed within that BBWS. Figure 9 shows the distribution of the local coverage for some representative BBWS when considering the dynamic instructions count. As it can be observed from the figure, some applications have their BBWS which extend from a few tens of instructions to a few hundreds or more, e.g. *adpcm, fft, bf, dijkstra, patricia, bmath.* These applications are therefore best candidates for local optimizations such as instruction coalescence that reduces a region's critical path, or local strength reduction which replaces expensive operations with cheaper ones. In the figure, some BBWS whose sizes extend beyond a few thousand of instructions are also distinguishable, e.g. *bmath, dijkstra, patricia, sha.* As these applications tend to spend a large amount of their execution time within a single region, they may best benefit from memory re-layout techniques such as cache-conscious placements or resizing. The local coverage is however not sufficient enough for deciding on the pertinence of a BBWS.

Table 3. Qualified BBWS as hot paths.

Bench	BBWS (%)	local cov. (%)	glob. cov. (%)	reuse (avg)
dijkstra	2.81	0.09	47	1.74
adpcm	5.88	< 0.005	90	0.00
bf	27.01	0.06	24	85.00
fft	11.7	< 0.005	7	4.21
sha	20.0	0.06	72	0.75
bmath	15.22	0.05	37	19.21
patricia	5.85	0.15	65	24.84

Table 4. Energy ratio.

config	energy/access
32K4W	1.00
32K2W	0.58
32K1W	0.37
16K2W	0.55
16K1W	0.35
8K1W	0.35

Global coverage. The local coverage must be interpreted in the light of global coverage to yield a fair understanding of the pertinence of a BBWS. Such a comprehensive reading can be provided with help of Figure 10. For this experiment, we have fixed an arbitrary threshold at 5% of the total instructions count as indicated in the figure with the *threshold* line. With regard to the local coverage of each BBWS, all the points above the threshold line are therefore these that are likely to provide substantial performance benefits across the whole program run. Of most concern are all the applications at the exception of *fft*, which has a global coverage value slightly below the threshold. Some applications such as *sha* and *patricia* exhibit BBWS whose sizes extend from a few hundreds to a few thousands of instructions, with a fairly good distribution among the two. This is an indication that these BBWS are good candidates for both local and global optimizations.

Reuse distance. The last criterion that qualifies a BBWS as a hot path is the reuse distance. This metric measures the heat of a BBWS by estimating the average number of accesses to different basic blocks between non-overlapping occurrences of this BBWS. Clearly, the larger is the reuse distance, less is the probability that it is a hot path. This trend can be well observed in Figure 11 where we show the cumulative distribution of the reuse distance for our benchmarks set. Applications with BBWS whose sizes extend to a few tens of instructions tend to have reuse distance values distributed among a few tens (e.g. *bmath, fft, bf*) to a few hundreds (e.g. *dijkstra, bmath, fft, sha, adpcm*) and thousands (e.g. *patricia, dijkstra, adpcm*) of basic blocks. Medium sized BBWS which extend from a few hundreds to a few thousands of instructions constitute the other category with reuse distance values less than a thousand, at the exception of *patricia, dijkstra* and *bmath*. Finally, as it is to be expected, very large BBWS tend to have also poor reuse distance values as evidenced with *patricia* and *dijkstra*. Tough, an exception with *dijkstra* is to be noted as a few number of these BBWS exhibit very good reuse distance value with $\overline{D} \approx 1$.

We summarize our experimental results in Table 3. The values were computed with a global threshold at 5%. This table presents results obtained by combining all the selection criteria together in order to qualify a BBWS as a hot path. As illustrated in the table, from 7% to 90% of the program dynamic instructions

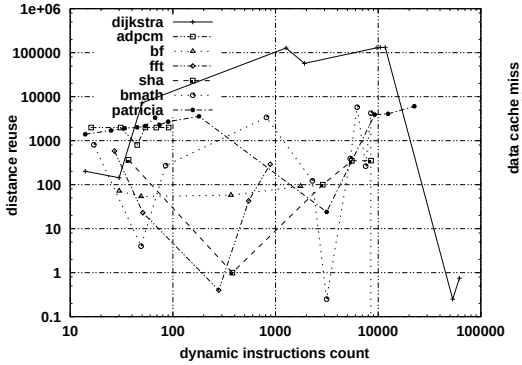

Fig. 11. Reuse distance.

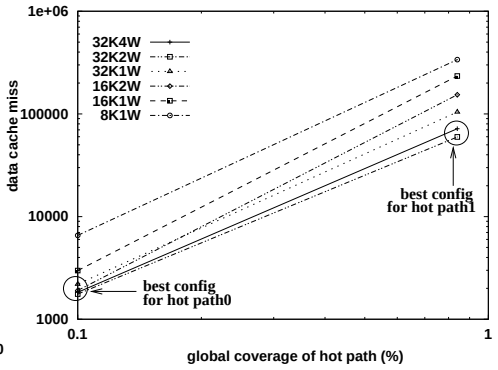

Fig. 12. Hotpath d-cache miss distribution for *dijkstra* (baseline config is 32K4W).

can be covered using our approach, with only as much as 0.15% of the dynamic instructions being executed within a single region.

6.3 Application example: adaptive cache reconfiguration

The cache hierarchy is the typical example where the power/performance trade-off plays a central role. While a large cache permits significant improvements in performance, only a small fraction of it is usually accessed during a program run. Henceforth, to address this source of inefficiency, much recent work has focused on the design of configurable caches [2, 21]. A key point with such work is deciding when to perform such a reconfiguration. With general purpose processors, this can be done dynamically with some mean of hardware, or at software following procedure boundaries [2]. With embedded systems, this is often done once on a per-application basis [21].

In this section, we examine the possibility of reconfigurating a cache at hot path boundaries. To do so, we assume a scheme similar to that presented in [21] in which the associativity of a cache can be modified while still preserving the whole cache capacity. Furthermore, we also assume an extension of this scheme, proposed in [18], in which the associativity as well as the size of a cache can be adapted at runtime with the help of a reconfiguration instruction. Because of space convenience, we will only discuss one result, namely *dijkstra* which is that having the best BBWS profiles with larger local and global coverage and low reuse distance. Figure 12 shows the cumulative distribution of the number of data cache misses, using varying cache configurations, for the two most representative hot paths of *dijkstra* with global coverage at 10% and 83%, respectively. As indicated in the figure, each hot path has a set of cache configuration candidates which vary according to either of the selection criteria introduced in Section 5. The first hot path, for instance, has a reuse distance of ≈ 0 and a local coverage of 0.09%, whereas the second occurs practically each 4 blocks with a relative

low local coverage ($\approx 0.004\%$). The first hot path is therefore more regular than the second, which could explain the larger choice for the former. Table 4 shows the relative energy per access obtained by means of CACTI [19] for each cache configuration. Each xKyW stays for a cache of size x and associativity y. The best configuration for *hot path0* is given by 32K1W which is from far more energy-efficient than the 32K4W case. On the other hand, for *hot path1*, 32K2W yields the best energy-performance ratio. However, although both hot paths yield substantial energy savings, only the first one may be of interest because it has a near 0 reuse distance value, which infers that reconfiguration will take place very infrequently. This is crucial for performance as each reconfiguration instruction consumes extra cycles and energy. The energy savings obtained in this way is in the order of 12% with almost no performance slowdown (less than 1%).

7 Related work

Many work have been proposed to collect profiling information. In [3], Ball and Larus propose to collect profile information via edges profiling. They extended their work in [4] to include path profiling information that are restricted to intra-procedural paths. Bala [1] then augmented the intra-procedural path profiling scheme to capture inter-procedural paths as well. A similar work has been proposed by Larus [14] which relies on a online compression scheme, SEQUITUR [16], to produce a compact representation of a whole-program paths. Our scheme is to some extent similar to [14] in that we also provide a representation of a whole-program paths. However, unlike the DAG representation used in [14], we rely on a suffix array representation that permits the implementation of very fast searching algorithms, allowing quick offline processing; thereby offsetting the high runtime overhead of Larus's scheme. In addition, this also permits us to treating each dynamic path distinctly from one other and consider large trace sizes. The performance of the proposed scheme can be rather significantly improved, namely by using other compression techniques which are complementary to that proposed in this paper. For instance, a direct improvement can be obtained by encoding the suffix array compression scheme described in [10]. Compression techniques such as that describe in [17] can also be used to further reduce the size of the trace to less than a fraction of a bit per reference.

8 Conclusions

While suffix arrays (SAs) have been widely used in biological data analysis or text editing, we are not aware of any prior published work that shows its application to compiler optimization. In this paper, we presented a first attempt to apply suffix array to the compiler domain. In particular, we showed how a SA can be used to represent a whole-program paths and to accurately identify hot program paths. Our evaluation results revealed that up to 48% of the code can be covered by hot paths, each one representing at most 0.15% of the total instructions count. Practical application of our approach has confirmed its effectiveness to

reduce power consumption. We showed that up to 12% energy savings can be obtained with a hot-path-directed adaptive cache resizing strategy that used our technique. Because of its power to precisely model program paths (reuse distance, local+global coverage), we believe that SAs can be of a crucial aid to assist a programmer during the optimization process.

References

1. Bala, V. Low overhead path profiling. Technical Report HPL-96-87, Hewlett Packard Labs, 1996.
2. Balasubramonian, R., Albonesi, D.H., Buyuktosunoglu, A., and Dwarkadas, S. Memory hierarchy reconfiguration for energy and performance in general purpose processor architectures. In *Proc. of the 33th Int'l Conf. on Microarchitecture*, pages 245–257, Dec. 2000.
3. Ball, T., and Larus, J.R. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.
4. Ball, T., and Larus, J.R. Efficient path profiling. In *Proc. of the 29th Annual Int'l Symposium on Microarchitecture*, Dec. 1996.
5. Ball, Thomas. What's in a Region? or computing control dependence regions in near-linear time for reducible control flow. *ACM Letters on Programming Languages and Systems*, 2(1-4):1–16, March-Dec. 1993.
6. Bodin, F., Rohou, E., and Seznec, A. Salto: System for Assembly-Language Transformation and Optimization. In *Proc. of the 6th Workshop on Compilers for Parallel Computers*, 1996 Dec.
7. Burger, D., and Austin, T. The SimpleScalar Tool Set, Version 2.0. *Computer Architecture News*, pages 13–25, 1997.
8. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., and Zadeck, K. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
9. Faraboschi, P., Brown, G., Fisher, J.A., Desoli, G., and Homewood, F. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In *Proc. of the 27th Int'l Symposium on Computer Architecture*, June 2000.
10. Grossi, R., and Vitter, J.S. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *Proc. of the ACM Symposium on the Theory of Computing*, 2000.
11. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., and Brown, R.B. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proc. of the 4th IEEE Int'l Workshop on Workload Characterization*, pages 3–14, Dec. 2001.
12. Jacobson, Q., Rotenberg, E., and Smith, J. Path-Based Next Trace Prediction. In *Proc. of the 30th Int'l Symposium on Microarchitecture*, Nov. 1997.
13. Karp, R.M., Miller, R.E., and Rosenberg, A.L. Rapid Identification of Repeated Patterns in Strings, Arrays and Trees. In *Proc. of the 4th ACM Symposium on Theory of Computing*, 1972.
14. Larus, J.R. Whole Program Paths. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, May 1999.
15. Manber, U., and Myers, G. Suffix Arrays: A New Method for on-line String Searches. In *Proc. of 1st ACM-SIAM SODA*, pages 319–327, 1990.

16. Nevill-Manning, C.G. and Witten, I.H. Identifying Hierarchical Structure in Sequences. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.
17. Plezkun, A.R. Techniques for compressing program address traces. In *Proc. of the 27th Int'l Conf. on Microarchitecture*, pages 32–40, 1994.
18. Pokam, G., and Bodin, F. Energy-efficiency potential of a phase-based cache resizing scheme for embedded systems. In *Proc. of the 8th Int'l Worskhop on Interaction between Compilers and Computer Architectures*, Feb. 2004.
19. Shivakumar, P., and Jouppi, N. Cacti 3.0: An integrated cache timing power, and area model. Technical report, DEC Western research Lab, 2002.
20. Young, Cliff., and Smith, Michael. Better Global Scheduling Using Path Profiles. In *Proc. of the 30th Int'l Symposium on Microarchitecture*, Dec. 1998.
21. Zhang, C., Vahid, F., and Najjar, W. A highly configurable cache architecture for embedded systems. In *Proc. of the 30th Int'l Symposium on Computer Architecture*, June 2003.