Modeling Program Traces with Nested Loops

Philippe Clauss
INRIA Rocquencourt
A3 Project
78153 Le Chesnay Cedex - France
clauss@icps.u-strasbg.fr

Bénédicte Kenmei Youta
ICPS/LSIIT, Université Louis Pasteur, Strasbourg
Pôle API, Bd Sébastien Brant
67400 Illkirch - France
kenmei@icps.u-strasbg.fr

Odile Rousselet
ICPS/LSIIT, Université Louis Pasteur, Strasbourg
Pôle API, Bd Sébastien Brant
67400 Illkirch - France
rousselet@icps.u-strasbg.fr

ABSTRACT

Tracing can be an opportune way of analyzing programs in order to understand their behaviors, to overcome static analysis limitations and to build some efficient optimizations. However, it can be difficult to translate traces in a comprehensive way or to represent them through a model which can be used for simulations, for predictions and for optimizations. In this paper, we propose to model program traces with nested loops by first recognizing some repetitive and periodic patterns. The generated loop nest can then be used for all the above mentioned purposes while taking advantage of all the developments made in the polytope model.

Keywords

program analysis, feedback-directed optimization, program behavior, trace analysis, polytope model.

1. INTRODUCTION

Understanding the behavior of a program is the key to many issues in software and hardware development today. Requirements asked to a system are not limited to its functionalities, but also to the way it satisfies them. This is particularly emphasized in embedded systems conception. Real-time constraints, power and memory consumption saving constraints constitute some current issues. Hence methods and tools are needed to analyze and control program behavior.

Static analysis of programs has received a lot of attention in the last decades and resulted in significant advances in program performance optimization. But with the growing complexity of nowadays software and hardware, and with the growing number of behavior constraints that has to be considered, the static analysis range is insufficient. The integration of dynamic information into static analysis offers promising new opportunities to complement static analysis.

Workshop on *Exploring the Trace Space for Dynamic Optimization Techniques,* held in conjunction with ICS'03, June 2003

In this paper, we consider the analysis of program traces consisting in some instructive information generated from code instrumentation. For example, the knowledge of the memory addresses accessed during the whole run, or partial run, of a program can be used to understand its memory behavior, to construct a prediction model for prefetching or to improve temporal and spatial locality of the accesses. The main task in analyzing such traces is to elaborate a model representing them in a "smart way":

- the model can be used eventually to reproduce the same data as in the traces,
- it has to include the general behavior in order to be well-suited for prediction,
- it should constitute a fruitful framework for optimizing transformations of the analyzed code.

The best known and simplest approach to model numerical measures is to compute regression or interpolation functions. However it is not suited for any kind of measures, and overall does not generally fit the above listed wishes.

In this paper, we only consider constant behaviors, or "symbolically constant" behaviors. Constant behaviors do not depend on input variables or intermediary results. "Symbolically constant" behaviors are behaviors that only depend on some program parameters. Constant behaviors can be identified by observing the same measures from several runs, or by static analysis of the considered instructions. "Symbolically constant" behaviors can be identified by interpolating several measures from runs with several parameter values. We are currently investigating approaches for analyzing varying behaviors in order to identify program invariants [3].

Analyzing measures is of current practice in many sciences, where measures are classically considered as *time-series*, i.e., an ordered sequence of observations. For example in economics, interest may be focused on the weekly fluctuations in the stock prices and their relationships to the unemployment figures. In medicine, systolic and diastolic blood pressures followed over time for a group of patients could be useful for assessing the effectiveness of a drug used in treating hypertension. Several well-known mathematical models

and tools are provided for time-series analysis (see for example [1] for a general dissertation on the subject).

However, these models are overall mathematical, and programming structures model have not yet been considered. We think that programming structures can be advantageously used to model time-series in any observation field, since it can take advantage of all the expression facilities and of all the program analysis and transformations advances of computer science. Such a modeling process can be seen as writing another program reproducing the behavior of a program which was solely written to cover some functionalities.

Such an analysis practice is not usual in computer science for observing software and hardware behaviors. Current opinions are that numeric systems do not have such a complex behavior. However due to the growing complexity of systems, this analysis approach is getting more and more attention.

In this paper, we show that nested loops structures can be used to model observations from running programs that have periodic and repetitive behaviors. Most resource consuming programs have such a behavior and this model can take advantage of the polytope model [4] to evaluate many quantitative behavior properties and to build efficient optimization methods.

The paper is organized as follows. In section 2, some general considerations about tracing are discussed. It is also explained how the programs we consider are instrumented. Section 3 describes our loop model of nested patterns, while section 4 presents our pattern recognition procedure. In section 5, we show how traces depending on some program input parameters can be modeled symbolically. Section 6 deals with the generation of the loop nest from the recognized patterns and section 7 shows how the model can be transformed to provide useful information. Finally, some considerations about optimizations built from the model are discussed in section 8.

2. TRACING DATA AND INSTRUCTION ADDRESSES

Between all the memory accesses achieved by a running program, a convenient and useful tracing has to focus on some specific accesses that can help in understanding the scanning of a given data structure from a given procedure, or in defining the control flow followed during the run. Hence instrumenting a program can be a difficult task and may need a preliminary static analysis or even some preliminary profiling, in order to detect what accesses are relevant to expensive and representative behavior.

When focusing on a data structure, instrumentation has to be elaborated according to the structure organization. Accesses to a structure whose nodes are defined by another structure are traced on several dimensions: address of the accessed object of the main structure, address of the accessed object of the first inner structure, and so on.

Moreover when tracing data accesses, the memory allocation process has an important impact on the way the obtained memory addresses can be interpreted and used. Generally, when structure nodes are created separately with necessary memory allocated just for one node, it is not guaranteed that successively created nodes are allocated consecutively in memory. Hence we must be aware of this fact and eventually transform the node creation procedure in order to ensure consecutively allocated nodes. For example, the memory allocator `ccmalloc` proposed by Chilimbi *et al.* [2] can be used: the user provides a pointer to an existing object that is likely to be accessed contemporaneously with the newly allocated object. Whenever possible, `ccmalloc` allocates the new object in the same cache block as the existing object.

```
int TreeAdd (t,prev)
register tree_t *t;
{if (t == NULL)  { return 0;}
 else {
  int leftval, rightval, value;
  tree_t *tleft, *tright;
  printf("%d ",((int)t-(int)prev)/16);//<-
  tleft = t->left;
  leftval = TreeAdd(tleft,t);
  if (tleft != NULL)
   printf("%d, ",((int)t-(int)tleft)/16);//<-
  tright = t->right;
  rightval = TreeAdd(tright,t);
  if (tright != NULL)
   printf("%d, ",((int)t-(int)tright)/16);//<-
  value = t->val;
  return leftval + rightval + value; } }
```

Figure 1: `treeadd` instrumentation.

When the interesting data accesses have been identified, code instrumentation is simply done by printing the memory address accessed at each data access instruction. This address is usually obtained by printing the pointer value. When focusing on a data structure whose nodes have constant sizes, the memory addresses can be normalized by subtracting the base address of the structure and dividing them by the node size. Hence obtaining a value q in the measures means that the q-th node has been accessed.

Tracing instruction addresses is more technical: we use the debugger GDB to print all the executed instructions addresses, by giving as input a file with convenient number of `stepi` commands.

Example 1 We consider the program `treeadd` from olden benchmarks. This program adds the values in a binary tree. Nodes are first created in RLN mode and then are scanned in LRN mode. The input parameter is the chosen tree depth. We observe that all the nodes are allocated in memory in the same order as they are created. The scanning function is instrumented in order to print the strides between two successive node accesses as shown in figure 1. Strides are normalized since each node is 16 bytes. Running the program for a 6 tree depth gives the answer of figure 2. □

Although we focus in this paper on memory addresses, many other information can be useful. For example, power consumption can be measured at regular frequency or at given

32, 16, 8, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -8, 1, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -1, -16, 1, 8, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -8, 1, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -1, -1, -32, 1, 16, 8, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -8, 1, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -1, -16, 1, 8, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -8, 1, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -1, -1, -1

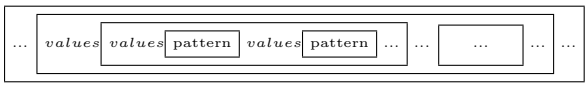
Figure 2: Memory strides from running `treeadd` for a 6 tree depth.

steps during the run of a program, but it may need some specific hardware. Cache or branch misses can be measured at regular frequencies or at given steps as well. Correlation between several kinds of information can also be analyzed.

3. THE NESTED LOOPS MODEL OF NESTED PATTERNS

A pattern is defined as a sequence of successive values that is repeated at least 2 times in the analyzed series. It can also be defined recursively as a sequence of successive values and smaller patterns. Hence any pattern can be modeled by a loop of the form shown in figure 3, where *nb_occurrences* is the number of times the pattern occurs in the series and $values_q(i)$ denotes a sequence of values depending on the value of index i.

Finally, the whole series is modeled as a pattern represented by a loop nest, where each inner loop nest represents a sub-pattern or some surrounding values and so on. Hence this model can also be seen as a hierarchy of englobing patterns as shown hereunder:



where the innermost patterns are patterns of constant values, and the other values are changing at each iteration. This model results in a multi-dimensional indexing of the values, where each dimension is associated to a pattern level.

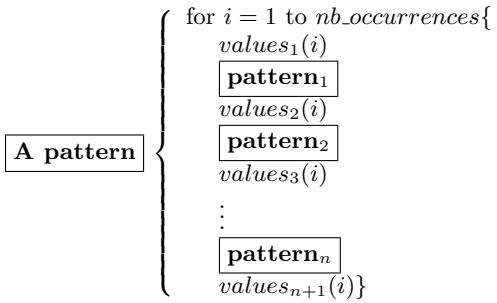


Figure 3: Loop modeling a pattern.

4. IDENTIFYING REPETITIVE AND PERIODIC BEHAVIORS

Repetitive and periodic address patterns can be observed on programs accessing iteratively the same memory addresses. But repetitive and periodic behavior can also be observed for more general programs by observing some neighboring facts: instead of analyzing the sequence of accessed memory addresses, strides between successive accesses can be even more informative about memory behavior. Moreover, in the perspective of temporal and spatial locality improvement, it is even better suited. Constant behavior can also be more easily identified from strides.

More generally, periodic behavior is characterized by a value k such that any q-th measured value such that $q \ mod \ k = r$ can be deduced from the $(q-k)$-th value by adding a constant value $v_r \in \mathbb{Z}$. Hence repetitive patterns $[v_0, v_2, ..., v_{k-1}]$ are observed on the sequence of strides occurring between k-spaced accesses: let $[a(i)], i = 1..n$, be a sequence of n accessed memory addresses, repetitive patterns of size k are observed on the sequence $[a(i+k) - a(i)]$. If such a value k exists, it can be found by computing the autocorrelation coefficients of the input sequence:

Let m be the average value in the input sequence: $m = \frac{\sum_{i=1}^{n} s[i]}{n}$ where n denotes the number of input values and $s[i]$ denotes the i-th input value. Let

$$c(q) = \sum_{i=1}^{n-q} \frac{(s[i] - m)(s[t+q] - m)}{n}$$

The autocorrelation coefficient for value q is defined by $r(q) = c(q)/c(0) \in [-1..1]$. We compute $r(q)$ for $q = 1..n/2$. Value k may be given by the highest autocorrelation coefficient $|r(k)|$. The graphic representation of all computed $r(q)$ results in the *correlogram* of the analyzed sequence as shown on figure 4 [1].

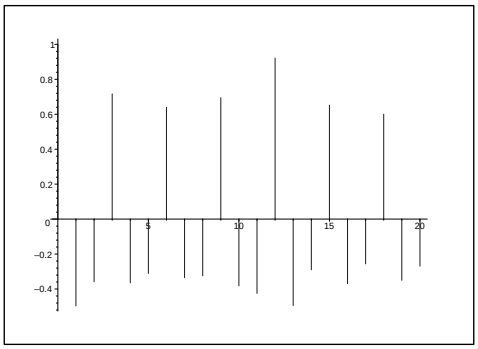


Figure 4: A correlogram

When focusing on memory objects of constant sizes, strides indicate the number of objects that are skipped between two accesses. But since instruction words are of different sizes on VLIW processors, printing such meaningful steps may need some additional processing on the obtained memory addresses.

We developed a tool that identifies repetitive patterns from a given sequence of measures that can be addresses or strides between successive accesses. Patterns are identified according to the nested loops model described in section 3. The algorithm works as follows:

The identified patterns are stored in a tree structure whose root is the whole input sequence s. The search process works

through the following steps:

- for each value in the input sequence, three information are computed and stored: the sum of all the preceding values and the current one, the number of elements (the distance) to the next element with same value, and finally the number of times the current value occurs in the whole sequence. The sums of all preceding values will be used as a hash function in order to accelerate sub-sequences comparisons.
- while there exists a value occurring at least 2 times
  - search for a value v occurring the minimum number of times, but at least two times, and whose distance to the next element v_{next} with same value is maximum. Select this value.
  - extend the selection to the left element per element until one of the following situations occurs:
    - element at the same position relatively to v_{next} is different.
    - an element occurring strictly less than two times is encountered.
    - the number of selected elements is equal to the distance from v to v_{next}.
    - the input sequence extremity has been reached.
  - if the number of selected elements is less than the distance from v to v_{next}, repeat the same selection procedure to the right.
  - compare the final selection to all possible selections of the same size with an element equal to v occurring at the same position in the selection. In order to accelerate comparisons, use the following: let i be the index of the first element and j the index of the last element in the initial selection. Let k be the index of v and l be the index of another element equal to v. Sequences $s[i]$ to $s[j]$ and $s[l-(k-i)]$ to $s[l+(j-k)]$ have to be compared. It is rapidly known whether the sequences are different by checking that $sum[j] - sum[i-1] \neq sum[l+j-k] - sum[l-k+i-1]$. Else, compare element per element both sub-sequences in order to ensure that they are equal. Search for another element equal to v and repeat the procedure. Create a branch tree with this identified pattern and store the pattern size, the number of occurring times, and the first elements positions.
  - set to zero the number of occurring times of all elements that are included in the recognized patterns. Update this number for all the other elements.
- repeat the loop for all the recognized patterns in order to find embedded patterns and until no new pattern is recognized.

Example 2 Let us run our algorithm with a simple example. Consider the input sequence 1, 1, 8, 1, 1, 8, 1, 1, -21. The recognition procedure must first identify patterns 1, 1, 8 and 1, and then pattern 1 inside pattern 1, 1, 8. At the beginning, some information shown hereunder are computed:

	1	1	8	1	1	8	1	1	-21
index	0	1	2	3	4	5	6	7	8
sum	1	2	10	11	12	20	21	22	1
nb_occ.	6	6	2	6	6	2	6	6	1
distance	1	2	3	1	2	-1	1	-1	-1

where *sum* is the sum of all the preceding values and the current one, *nb_occ* is the number of times the current element appears in the sequence and *distance* is the number of elements between two successive equal elements.

The first element appearing the minimum number of times is 8 at position 2. Next element with same value is at position $2+3=5$. Element 8 at position 2 is selected and the selection is extended to the left element per element:

- 1, 8 is selected since 1 is equal to the element at position $5-1=4$;
- 1, 1, 8 is selected since 1 is equal to the element at position $5-2=3$;
- selection extension to the left is stopped since the extremity has been reached ;
- the selection is not extended to the right since the number of already selected elements, 3, is equal to the distance to next element equal to 8.

Pattern 1, 1, 8 has been identified. For the so-identified elements, *nb_occ* is set to zero and updated for the other elements:

	1	1	8	1	1	8	1	1	-21
index	0	1	2	3	4	5	6	7	8
nb_occ.	0	0	0	0	0	0	2	2	1

The next element appearing the minimum number of times is 1 at position 6. Next element with same value is at position $6+1=7$. Element 1 at position 6 is selected and since the number of selected elements is equal to the distance to next element, no selection extension is done. Pattern 1 has been identified. *nb_occ* is set to zero for the identified elements and hence there is no more elements appearing at least two times.

The procedure is now repeated on each previously identified patterns. All initial information are computed for the input sequence 1, 1, 8. Pattern 1 is obviously detected in the same way as before.

On the strides sequence obtained from running `treeadd`, our tool finds the following patterns:

- pattern 1 repeated 2 times:

  16, 8, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -8, 1, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -1, -16, 1, 8, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -8, 1, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -1, -1

  with surrounding values $(\{32\}, \{-32\})$, and $(\{1\}, \{-1\})$.

- pattern 2 repeated 2 times inside pattern 1:

8, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -8, 1, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -1

  with surrounding values $(\{16\}, \{-16\})$, and $(\{1\}, \{-1\})$.

- pattern 3 repeated 2 times inside pattern 2:

4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1

  with surrounding values $(\{8\}, \{-8\})$, and $(\{1\}, \{-1\})$.

- pattern 4 repeated 2 times inside pattern 3:

2, -2, 1, -1

  with surrounding values $(\{4\}, \{-4\})$, and $(\{1\}, \{-1\})$.

5. SYMBOLICALLY CONSTANT BEHAVIORS

Consider several traces from a "sufficient number" m of program runs with several input parameter values. We denote by $trace(n)$ the measures obtained from running the program with n as input parameter. We classify symbolically constant behaviors in 3 non-exclusive kinds:

(1) $trace(n)$ is included in $trace(n+1)$,

(2) the measured values are not the same but are dependent on the parameters values,

(3) the pattern sizes and/or the number of surrounding values are different in $trace(n)$ and $trace(n+1)$.

Case (1) is obviously detected and several functions characterizing the added values in $trace(n+1)$ are computed:

- a function interpolating the number of new englobing patterns,
- a function interpolating the number of new elements in the new englobing patterns,
- a function interpolating the number of values surrounding the new englobing patterns,
- a function interpolating the number of times $trace(n)$ is repeated in the new englobing pattern,
- functions interpolating the new values in the new englobing patterns: if the number of new values is different in the traces, we consider an additional variable q denoting the q-th new value in the new englobing pattern.
- functions interpolating the new surrounding values of the new englobing patterns: if the number of new surrounding values is different in the traces, we consider an additional variable q denoting the q-th new surrounding value of the new englobing pattern.

Case (2) is detected by computing, for all obtained values at the same position in each of the m traces, a function interpolating the parameter values and these values in the traces: let $v_{1,q}, v_{2,q}, ..., v_{m,q}$ be the q-th values obtained in the traces from m program runs with respective parameter values $n_1, n_2, ..., n_m$. For all q, we compute a function f_q interpolating points $(n_1, v_{1,q}), (n_2, v_{2,q}), ..., (n_m, v_{m,q})$. All the so-computed functions f_q define a symbolic trace $f_1, f_2, ..., f_k$ where k is the number of values in each trace. Such a symbolic trace is satisfactory if considering any additional program run with some other parameter values, the generated trace fits the symbolic trace.

Case (3) is detected by examining each identified patterns from the smallest to the largest, and by computing a function interpolating the parameter values and the pattern sizes. For example, let $s_1, ..., s_m$ be the smallest pattern sizes identified from m program runs with respective parameter values $n_1, n_2, ..., n_m$. We compute a function interpolating points $(n_1, s_1), ..., (n_m, s_m)$. This function represents a symbolic pattern size of the smallest patterns. We do the same for the next smallest patterns and so on. Such symbolic pattern sizes are satisfactory if considering any additional program run with some other parameter values, the generated pattern sizes fits the symbolic sizes.

If none of these cases has been detected, we consider that there is no symbolically constant behavior and specific modeling is done for instantiated input parameter values.

Example 3 Program `treeadd` is relevant of case (1). Running the program with 7 as tree depth results in the measures of figure 5. The interpolation functions are computed:

- number of new englobing patterns: $nep(n) = 1$, since the new englobing pattern is the whole new sequence.
- number of new elements in the new englobing patterns: $nne(n) = 4$.
- number of values surrounding the new englobing patterns: $nvs(n) = 0$.
- number of times $trace(n)$ is repeated in the new englobing pattern: $ntr(n) = 2$.
- new values in the new englobing patterns: $vp_1(n) = 2^{n-1}, vp_2(n) = -2^{n-1}, vp_3(n) = 1, vp_4(n) = -1$.

6. GENERATING THE NESTED LOOPS

The nested loops are built by considering each pattern from the innermost to the outermost ones. First, the innermost patterns are composed by a sequence of values. A loop is generated, whose trip count is equal to the number of times the considered pattern is repeated in its englobing pattern.

Each pattern is generated by an innermost loop whose trip count is equal to the number of values occurring in it. In order for the correct value to be generated at each iteration, an interpolation function is computed: let $p_1, p_2, ..., p_n$ be the values composing the pattern, a function interpolating points $(1, p_1), (2, p_2), ..., (n, p_n)$ is computed.

64, 32, 16, 8, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -8, 1, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -1, -16, 1, 8, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -8, 1, 4, 2, -2, 1, -1, -1, -1, -32, 1, 16, 8, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -8, 1, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -1, -1, -64, 1, 32, 16, 8, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -8, 1, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -1, -16, 1, 8, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -8, 1, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -1, -32, 1, 16, 8, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -8, 1, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -1, -16, 1, 8, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -8, 1, 4, 2, -2, 1, -1, -4, 1, 2, -2, 1, -1, -1, -1, -1, -1, -1

Figure 5: Memory strides from running `treeadd` for a 7 tree depth.

For the values surrounding patterns, since the number of occurring values and the values themselves can vary, two kinds of interpolation functions have to be computed:

- the varying number of occurring values is obtained by generating a loop whose trip count is given by a function on the englobing loop index. This function is computed by interpolating couples of englobing loop index values and associated number of values: let $1, 2, .., q$ be the englobing loop index values and $n_1, n_2, ..., n_q$ be the number of values that have to occur, a function interpolating $(1, n_1), (2, n_2), ..., (q, n_q)$ is computed. This function is used as the upper bound of the loop.
- The correct values are generated from a function computed by interpolating index values of the above created loop and the values that have to be generated.

The same scheme is applied for the next englobing patterns until the whole measures have been considered. The final loop nest can be used to generate all the input measures. It can be schematized as it is shown on figure 6, where:

- a is the upper bound of the preceding loop at the same pattern (or loop) level ;
- $nb\_occurrences_{l,m}$ denotes the number of times the so-described m-th pattern of level l occurs in its englobing pattern ;
- b_q denotes the upper bound of a loop generating some surrounding values. This bound can be a function on i as described above ;
- $value_q(j)$ denotes a function interpolating surrounding values ;
- $pattern_q(j - b_q)$ denotes either a loop nest of the same scheme or a function interpolating the values of the q-th pattern of the current pattern level.

When the values are measured from a partial run, hypothesis of repetitive behavior on the whole run can be considered by generating an englobing loop whose trip count is parameterized.

$$
\begin{array}{l}
\vdots \\
\text{for } i = a + 1 \text{ to } a + nb\_occurrences_{l,m} \ \{ \\
\quad \text{for } j = 1 \text{ to } b_1 \\
\quad\quad value_1(j); \\
\quad \text{for } j = b_1 + 1 \text{ to } b_1 + nb\_occ(pattern_1) \\
\quad\quad pattern_1(j - b_1); \\
\quad \text{for } j = b_1 + nb\_occ(pattern_1) + 1 \text{ to } b_2 \\
\quad\quad value_2(j); \\
\quad \text{for } j = b_2 + 1 \text{ to } b_2 + nb\_occ(pattern_2) \\
\quad\quad pattern_2(j - b_2); \\
\quad \vdots \\
\quad \text{for } j = b_n + 1 \text{ to } b_n + nb\_occ(pattern_n) \\
\quad\quad pattern_n(j - b_n); \\
\quad \text{for } j = b_n + nb\_occ(pattern_n) + 1 \text{ to } b_{n+1} \\
\quad\quad value_{n+1}(j); \ \} \\
\vdots
\end{array}
$$

Figure 6: General scheme of the final loop nest.

Example 4 For any value n of the tree depth, the loop nest modeling the trace is shown figure 7, where *occ_val* denotes the value occurring in the trace at each step.

$$
\begin{array}{l}
\text{for } i_{n-1} = 1 \text{ to } 2 \ \{ \\
\ \text{for } i_{n-2} = 1 \text{ to } 1 \\
\ \ occ\_val = (1 - 2^{n-1}) * i_{n-1} + 2^n - 1; \\
\ \text{for } i_{n-2} = 2 \text{ to } 3 \ \{ \\
\ \ \text{for } i_{n-3} = 1 \text{ to } 1 \\
\ \ \ occ\_val = (1 - 2^{n-2}) * (i_{n-2} - 1) + 2^{n-1} - 1; \\
\ \ \text{for } i_{n-3} = 2 \text{ to } 3 \ \{ \\
\ \ \ \text{for } i_{n-4} = 1 \text{ to } 1 \\
\ \ \ \ occ\_val = (1 - 2^{n-3}) * (i_{n-3} - 1) + 2^{n-2} - 1; \\
\ \ \ \vdots \\
\ \ \ \ldots \text{for } i_2 = 2 \text{ to } 3 \ \{ \\
\ \ \ \ldots \quad \text{for } i_1 = 1 \text{ to } 1 \\
\ \ \ \ldots \quad\ occ\_val = -3 * (i_2 - 1) + 7; \\
\ \ \ \ldots \quad \text{for } i_1 = 2 \text{ to } 5 \\
\ \ \ \ldots \quad\ occ\_val = -2 * (i_1 - 1)^3 \\
\ \ \ \ldots \quad\ +31 * (i_1 - 1)^2/2 - 73 * (i_1 - 1)/2 + 25; \\
\ \ \ \ldots \quad \text{for } i_1 = 6 \text{ to } 6 \\
\ \ \ \ldots \quad\ occ\_val = 3 * (i_2 - 1) - 7; \ \} \\
\ \ \ \vdots \\
\ \ \ \text{for } i_{n-4} = 4 \text{ to } 4 \\
\ \ \ \ occ\_val = (2^{n-3} - 1) * (i_{n-3} - 1) - 2^{n-2} + 1; \ \} \\
\ \ \text{for } i_{n-3} = 4 \text{ to } 4 \\
\ \ \ occ\_val = (2^{n-2} - 1) * (i_{n-2} - 1) - 2^{n-1} + 1; \ \} \\
\ \text{for } i_{n-2} = 4 \text{ to } 4 \\
\ \ occ\_val = (2^{n-1} - 1) * i_{n-1} - 2^n + 1; \ \}
\end{array}
$$

Figure 7: Loop nest modeling the trace of program `treeadd`.

7. MODEL TRANSFORMATIONS AND INTERPRETATIONS (PREDICTION)

The generated loop nest can be used to reproduce the same data as in the trace, but it can also be used to get more useful information through some transformations. For example, if the loop nest models strides between successive memory accesses, it can be useful to transform it in order to get

directly the memory address accessed at a given iteration. The model can then be used to predict accessed memory addresses:

For any given iteration defined by its indices $(i_1, i_2, ..., i_n)$, the accessed memory address $addr$ is given by the algorithm shown figure 8.

1. $q = n; addr = base\_address$
2. for $1 \leq t \leq number\_of\_patterns\_in\_level\_q + 1$
 if $b_{t-1} + nb\_occ(pattern_{t-1}) \leq i_q \leq b_t$ then
  $addr+ = \sum_{k=1}^{t-1} \sum_{r=1}^{b_k} value_k(r) + \sum_{k=1}^{i_q} value_t(k)$
  $+ \sum_{k=1}^{t-1} stride_k \times nb\_occ(pattern_k)$
 if $b_t + 1 \leq i_q \leq b_t + nb\_occ(pattern_t)$ then
  $addr+ = \sum_{k=1}^{t} \sum_{r=1}^{b_k} value_k(r)$
  $+ \sum_{k=1}^{t-1} stride_k \times nb\_occ(pattern_k)$
  if $pattern_t(i_q - b_1)$ is an embedded loop nest then
   $addr+ = stride_t \times (i_q - b_t - 1)$
   $q- = 1$; go to step 2
  else $addr+ = \sum_{k=b_{t+1}}^{i_q} pattern_t(k - b_t)$

Figure 8: Algorithm to compute accessed addresses.

Example 5 Algorithm to compute accessed addresses for any tree depth n is shown in figure 9.
□

if $i_{n-2} = 1$ then
 $addr = base\_address + (1 - 2^{n-1}) * i_{n-1} + 2^n - 1$
if $2 \leq i_{n-2} \leq 3$ then
 if $i_{n-3} = 1$ then
  $addr = base\_address + (1 - 2^{n-1}) * i_{n-1}$
  $+(1 - 2^{n-2}) * i_{n-2} + 7 * 2^{n-2} - 3$
 if $2 \leq i_{n-3} \leq 3$ then
  if $i_{n-4} = 1$ then
   $addr = base\_address + (1 - 2^{n-1}) * i_{n-1}$
   $+(1 - 2^{n-2}) * i_{n-2} + (1 - 2^{n-3}) * i_{n-3}$
   $+17 * 2^{n-3} - 5$
   ⋮
   ...if $2 \leq i_2 \leq 3$ then
   ... if $i_1 = 1$ then
   ... $addr = base\_address + \sum_{k=1}^{n-2}(1 - 2^{n-k}) * i_{n-k}$
   ... $+5 * 2^{n-1} - 2n - 7$
   ... if $2 \leq i_1 \leq 5$ then
   ... $addr = base\_address$
   ... $+\sum_{k=1}^{n-2}(1 - 2^{n-k}) * i_{n-k} + 5 * 2^{n-1}$
   ... $-i_1^4/2 + 37 * i_1^3/6 - 53 * i_1^2/2$
   ... $+275 * i_1/6 - 2n - 32$
   ... if $i_1 = 6$ then
   ... $addr = base\_address + \sum_{k=1}^{n-3}(1 - 2^{n-k}) * i_{n-k}$
   ... $+5 * 2^{n-1} - 2n - 17$
   ⋮
  if $i_{n-4} = 4$ then
   $addr = base\_address + (1 - 2^{n-1}) * i_{n-1}$
   $+(1 - 2^{n-2}) * i_{n-2} + 7 * 2^{n-2} - 3$
 if $i_{n-3} = 4$ then
  $addr = base\_address + (1 - 2^{n-1}) * i_{n-1} + 2^n - 1$
if $i_{n-2} = 4$ then $addr = base\_address + 0$

Figure 9: Accessed addresses for program `treeadd`.

In our model, each memory access is indexed by a tuple $(i_n, i_{n-1}, ..., i_1)$. In order to predict what memory address is accessed at the m-th access, we have to determine the tuple corresponding to this m-th access. It is computed using the algorithm shown in figure 10, where $size_{q,k}$ denotes the number of elements in the k-th pattern of level q. (A.) is relevant for the case where the m-th reached value is a surrounding value of level q, while (B.) is relevant for the case where a value inside a pattern of level q is reached.

#define $surround(x)$
 $\sum_{k=1}^{x} (b_k - b_{k-1} - nb\_occ(pattern_{k-1}))$
#define $pattern(x)$
 $\sum_{k=1}^{x} size_{q,k} \times nb\_occ(pattern_k)$
$q = n$; $end = false$
while not end do
 for $1 \leq t \leq number\_of\_patterns\_in\_level\_q + 1$
  [A.] if $surround(t-1) + pattern(t-1) < m$
   $\leq surround(t-1) + pattern(t-1)$
   $+b_t - b_{t-1} - nb\_occ(pattern_{t-1})$ then
   $i_q = \sum_{k=1}^{t-1} nb\_occ(pattern_k) + m$
    $-pattern(t-1)$
   for $k = q - 1..1, i_k = 0$
   $end = true$
  [B.] if $surround(t-1) + pattern(t-1)$
   $+b_t - b_{t-1} - nb\_occ(pattern_{t-1}) < m$
   $\leq surround(t) + pattern(t)$ then
   $i_q = \sum_{k=1}^{t-1} nb\_occ(pattern_k) + surround(t)$
    $+\lfloor \frac{m - surround(t) - pattern(t-1)}{size_{q,t}} \rfloor + 1$
   $m = (m - surround(t) - pattern(t-1))$
    $mod\ size_{q,t}$
   if $pattern_t$ contains embedded patterns then
    $q- = 1$
   else $i_q+ = m$; $end = true$
    for $k = q - 1..1, i_k = 0$

Figure 10: Algorithm to compute the tuple associated to an access occurrence.

$i_5, i_4, i_3, i_2, i_1 = 0$
$i_5 = \lfloor \frac{m}{62} \rfloor + 1$
$m = m\ mod\ 62$
if $0 < m \leq 1$ then $i_4 = 1$; end
if $1 < m \leq 61$ then $i_4 = \frac{m-1}{30} + 2$
 $m = (m-1)\ mod\ 30$
if $0 < m \leq 1$ then $i_3 = 1$; end
if $1 < m \leq 29$ then $i_3 = \frac{m-1}{14} + 2$
 $m = (m-1)\ mod\ 14$
if $0 < m \leq 1$ then $i_2 = 1$; end
if $1 < m \leq 13$ then $i_2 = \frac{m-1}{6} + 2$
 $m = (m-1)\ mod\ 6$
$i_1 = m$

Figure 11: Algorithm to compute the tuples for program `treeadd`.

Example 6 For program `treeadd` and $n = 6$, computation of $(i_5, i_4, i_3, i_2, i_1)$ from a value m runs as shown in figure 11. Any m-memory access can now be predicted. For example, let $m = 88$. The corresponding tuple computed by the above algorithm is $(2, 2, 3, 3, 3)$. Since $n = 6$, the 88-th accessed memory address is:

$base\_address$
$+ \left(\sum_{k=1}^{4} (1 - 2^{6-k}) * i_{6-k} + 5 * 2^{6-1} - 3^4/2 + 37 \right.$
$\left. * 3^3/6 - 53 * 3^2/2 + 275 * 3/6 - 2 * 6 - 32 \right) \times 16$ bytes
$= base\_address + 19 \times 16$ bytes

```
for i5 = 1 to 2 {
 for i4 = 1 to 1
  occ_val = −31 ∗ (3 − i5) + 63 = 31 ∗ i5 − 30;
 for i4 = 2 to 3 {
  for i3 = 1 to 1
   occ_val = −15 ∗ (5 − i4) + 46 = 15 ∗ i4 − 29;
  for i3 = 2 to 3 {
   for i2 = 1 to 1
    occ_val = −7 ∗ (5 − i3) + 22 = 7 ∗ i3 − 13;
   for i2 = 2 to 3 {
    for i1 = 1 to 1
     occ_val = −3 ∗ (5 − i2) + 10 = 3 ∗ i2 − 5;
    for i1 = 2 to 5
     occ_val = −2 ∗ (7 − i1)^3 + 31 ∗ (7 − i1)^2/2
               −73 ∗ (7 − i1)/2 + 25
             = 2 ∗ i1^3 − 53/2 ∗ i1^2 + 227/2 ∗ i1 − 157;
    for i1 = 6 to 6
     occ_val = −3 ∗ i2 + 5; }
   for i_{n−4} = 4 to 4
    occ_val = −7 ∗ i3 + 13; }
  for i_{n−3} = 4 to 4
   occ_val = −15 ∗ i4 + 29; }
 for i_{n−2} = 4 to 4
  occ_val = −31 ∗ i5 + 30; }
```

Figure 12: Loop nest simulating the new trace.

8. FROM THE MODEL TO PROGRAM OPTIMIZATION

Optimization transformations of the analyzed program can be built from the model. For example, temporal data locality can be improved by re-ordering memory accesses such that accesses to the same address are closer. However such transformation has to take care of the dependences and has to be validated through static analysis, while data layout transformations can be applied without any restriction in order to improve spatial data locality.

Any change of memory allocation can be immediately simulated through the model in order to be evaluated. The new allocation has to be carefully defined such that two different objects are not allocated to the same address. This can be ensured by modifying the reference functions in the loop nest through a bijective substitution of the loop index tuple.

Example 7 For $n = 6$, tuple $(3-i_5, 5-i_4, 5-i_3, 5-i_2, 7-i_1)$ can be substituted to $(i_5, i_4, ..., i_1)$ in the functions defining occurring steps. The resulting loop simulating the new trace is shown figure 12. Running the simulation yields the strides given on figure 13. This induced data layout transformation is equivalent to creating the nodes in LRN mode.

```
1, 1, 1, 1, 1, -1, 2, -2, -1, 4, 1, -1, 2, -2, -4, -1, 8, 1, 1, -1,
2, -2, -1, 4, 1, -1, 2, -2, -4, -8, -1, 16, 1, 1, 1, -1, 2, -2, -1,
4, 1, -1, 2, -2, -4, -1, 8, 1, 1, -1, 2, -2, -1, 4, 1, -1, 2, -2, -4,
-8, -16, -1, 32, 1, 1, 1, 1, -1, 2, -2, -1, 4, 1, -1, 2, -2, -4, -1,
8, 1, 1, -1, 2, -2, -1, 4, 1, -1, 2, -2, -4, -8, -1, 16, 1, 1, 1, -1,
2, -2, -1, 4, 1, -1, 2, -2, -4, -1, 8, 1, 1, -1, 2, -2, -1, 4, 1, -1,
2, -2, -4, -8, -16, -32
```

Figure 13: Simulated new memory strides from running `treeadd` for a 6 tree depth.

9. CONCLUSION

The model has been successfully applied on several traces from different programs as `fir2dim` from olden benchmarks or `mcf` from spec2000 benchmarks, either for data or instruction addresses. The pattern recognition procedure has been implemented while the loop generation procedure is currently being implemented.

Although it is already well-suited for prediction or simulation, the next step is now to systematically build some optimization transformations from this model. We are also investigating approaches for analyzing varying behaviors and identifying program invariants [3].

10. REFERENCES

[1] C. Chatfield. *Time-Series Forecasting*. Chapman & Hall, 2000.

[2] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[3] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[4] P. Feautrier. *The Data Parallel Programming Model*, volume 1132 of *LNCS*, chapter Automatic Parallelization in the Polytope Model, pages 79–100. Springer-Verlag, 1996.