The Periodic-Linear Model of Program Behavior Capture

Philippe Clauss, Bénédicte Kenmei, and Jean Christophe Beyler

ICPS/LSIIT, Université Louis Pasteur, Strasbourg
Pôle API, Bd Sébastien Brant
67400 Illkirch, France
{clauss,kenmei,beyler}@icps.u-strasbg.fr

Abstract. Understanding and controlling program behavior is a challenging objective for the design of advanced compilers and critical system development. In this paper, we propose an analysis and modeling strategy of program behavior characteristics by considering traces generated from opportune code instrumentation. The proposed models consist in periodic and linear interpolations separated into adjacent program phases. It is shown that these models exhibit apparent and useful information on program behavior. Moreover they can directly be used to guide static optimizations or to build dynamic optimization processes as it is shown for the implementation of efficient dynamic data prefetching processes for some benchmark programs.

1 Introduction

Many works have shown that software controlled policy of hardware mechanisms can significantly improve their efficiency. A compiler can be able from a static analysis of the source code to generate some instruction hints [1]. However, such an approach is only exploitable for static control and data structures as for-loops accessing multi-dimensional arrays through affine reference functions. When considering more general control structures accessing data through pointers, static optimizations generally can not be applied since essential information is not known at compile-time and can only be observed during execution. Hence dynamic analysis and optimization have become an important area of research.

In this paper, we propose an off-line analysis and modeling strategy for traces generated from opportune code instrumentation. We consider Input Independent Programs (IIPs) as programs whose execution behaviors are not influenced qualitatively by their input data. IIPs are interesting candidates for trace driven analysis and profile feedback optimizations, since information common to any of their runs and input data can be extracted. IIPs, or input independent program sub-parts, can be identified through several approaches: input-dependency analysis by abstract interpretation [2]; static code analysis of control structures and conditionals by variable propagation; comparisons of traces resulting from a sufficient number of executions and showing the same execution behavior. Notice

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 325–335, 2005.

that this last approach can never be as reliable as the previous ones, but can give useful information relevant to the most frequent case.

Traces resulting from IIPs are candidates for standard data-mining methods based on statistical and machine learning algorithms. Although some relevant information can be found from their use, programs should take advantage of more dedicated approaches. Interesting observations are necessarily related to software and/or hardware mechanisms involving some specificities: many traces can be reduced to binary data or at least integers, repetitive and/or periodic behaviors can be often expected from program executions, observations can hopefully lead to simulation models implemented as programs, ...

With this purpose of a dedicated approach, we propose a model based on periodic interpolation by intervals of the trace values. Periodic interpolation consists in interpolating a sequence of values by a periodic polynomial function. A periodic polynomial is a polynomial whose coefficients are periodic numbers, *i.e.*, a sequence of values indexed by the modulo of the variable relatively to the number of these values. For example, the periodic polynomial $[1,2,3]x^2 + [3,4]x + 5$ is equal to $x^2 + 3x + 5$ if $x \ mod\ 3 = 0$ and $x \ mod\ 2 = 0$.

Periodic interpolation allows to extract periodic behavior information of the observed program as well as reduce the complexity of the interpolation function. For example, the sequence $[3,3,7,13,11,23,15,33,19,43,23]$ where each element is respectively indexed by $0,1,2,...$ is interpolated classically by the polynomial:

$$\frac{-4}{2835}x^{10} + \frac{197}{2835}x^9 - \frac{277}{189}x^8 + \frac{16348}{945}x^7 - \frac{16912}{135}x^6 + \frac{77408}{135}x^5 - \frac{932752}{567}x^4 + \frac{7998976}{2835}x^3 - \frac{814336}{315}x^2 + \frac{59518}{63}x + 3$$

which is a quite high degree polynomial exhibiting no apparent information about periodicity. Instead, periodic interpolation would give the following interpolation function: $[2,5]x + [3,-2]$, showing a periodic behavior of period 2 and a linear relation between 2-spaced elements.

The elements inside a large program trace generally represent several different behaviors associated to several different program phases. Hence, a unique periodic interpolation function with a low degree can rarely be found on the whole trace, but on some contiguous values in separated intervals. These successive intervals covering the whole trace are then associated to successive program phases. We target originally linear functions, i.e., polynomials of degree 1, but construct a non-linear model by recursive compositions of the linear functions.

Since each periodic coefficient of the periodic interpolation function can itself be interpreted as a trace, we recursively apply the model to the coefficients. This approach yields the definition of a multi-dimensional time space and a granularity hierarchy of the program behavior.

Our phase definition criteria are different than those of other works based either on hardware or software metrics. A phase is classically defined as intervals characterized by values staying near a given average [4, 5]. Such an approach ables the extraction of some specific hardware behavior of a program. Our approach is closer to the program semantic since it can be seen as a way trying to re-write the original program from the unique knowledge of some observation traces, but in a more "behavior-expressive" way.

Our representation model is detailed in next section where periodic-linear functions are defined, as well as periodic-linear interpolation and our notion of program phases. Model construction algorithms are presented in section 3 where important considerations related to the nature of the extracted models are discussed. Applications and experiments are presented in section 4. Conclusions and perspectives are given in section 5.

2 Formal Definition of the Periodic-Linear Model

2.1 Periodic-Linear Function

A periodic-linear function f is a function of the form $f(x) = ax + b$ where a and b are periodic numbers. A periodic number is a finite list of n numerical values $[a_1, a_2, ..., a_n]$ where the rank of the selected value at a given time to evaluate f is given by $y \ mod \ n$, $y \in Z$:

$$f(x) = ax + b = [a_1, a_2, ..., a_n]x + b = \begin{cases} a_1x + b, \text{ if } y \ mod \ n = 0 \\ a_2x + b, \text{ if } y \ mod \ n = 1 \\ \quad ... \qquad\qquad ... \\ a_nx + b, \text{ if } y \ mod \ n = n - 1 \end{cases}$$

Notice that since b is also a periodic number of m values $[b_1, b_2, ..., b_m]$, f is also defined depending on $y \ mod \ m$. The number of values of a periodic number is called the *period*. Two periodic numbers can be reduced to the same period equal to the lowest common multiple (lcm) of their respective periods.

2.2 Periodic-Linear Interpolation

A periodic-linear interpolation of a time-serie links non-overlapping successive intervals (slices of the trace) such that any element in interval i at position j, e_{ij}, is linearly dependent of $e_{i-1,j}$: $e_{ij} = e_{i-1,j} + a_j$, where a_j is constant. The number of elements in each interval is the lowest common multiple of both periods of the periodic coefficients a and b in the interpolation function f.

We distinguish 4 possible cases:

1. all intervals are adjacent and their size p is constant;
2. all intervals are adjacent and their respective sizes $p(t)$ can vary depending on the interval occurrence t;
3. intervals are not necessarily adjacent and their size is constant;
4. intervals are not necessarily adjacent and their respective sizes can vary (see figure 1).

Adjacent intervals correspond to a unique behavior model while non adjacent intervals represent several interleaved behaviors: the dots in figure 1 are other intervals interpolated by some other periodic linear functions. A model with constant size intervals considers the duration as being a criterion characterizing a behavior, saying that two behaviors are identical if their durations are equal.

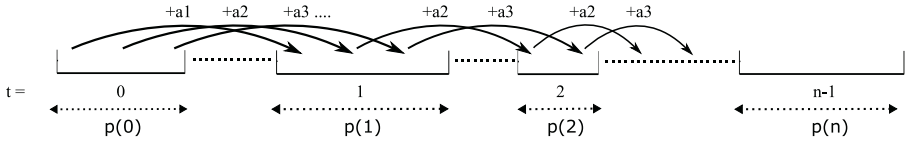

Fig. 1. an illustration of the case with non adjacent intervals of different sizes

On the other hand, with intervals of different sizes, the model does not consider the duration as being discriminant.

Consider two successive interpolated intervals i and $i-1$. If both intervals have the same size then the definition of periodic-linear interpolation given previously holds. Otherwise, we redefine periodic-linear interpolation in the following way. Let i_{max} be the largest interval of size max over all the interpolated intervals. Then for all successive elements e_{ij} in another interval i of size s, $0 \leq j \leq s-1$, there exist $\alpha \in Z$ and s successive elements $e_{i_{max}k}$ in interval i_{max}, $0 \leq k \leq max-1$, such that $e_{ij} = e_{i_{max}k} + \alpha a_k$. Moreover, the value of α is uniquely associated to interval i. In other words, any interval i can be mapped onto the interval i_{max} such that the difference between each inter-mapped elements is equal to αa_k, and no other mapping onto the interval i_{max} yields the same value of α.

With each of the n intervals is associated a time instant t, $0 \leq t \leq n-1$, defining the time space of the model. All n intervals are modeled by a periodic linear function $f(t) = at + b$ where a and b are periodic numbers. Their periods are equal to either p in cases (1) and (3), or the maximum of the $p(t)$'s in cases (2) and (4). These periodic numbers can have a large period and therefore constitute by themselves new time-serie. Hence we recursively apply our periodic-linear model to these new traces, i.e., to both periodic numbers, yielding an additional time dimension. Finally the whole application of the model yields a multi-dimensional time space $(t_1, t_2, ...)$.

Application of the model on both time-serie, or periodic numbers, a and b can result in two different periodic linear functions f_a and f_b. If their periods are different, we need to reduce them to the same period which is the lcm of their initial respective periods.

The whole recursive process yields a binary tree where each node is either a f_a or a f_b function and at each depth level is associated a time dimension. All functions of a same depth level have the same period and model simultaneously occurring traces. Finally, this multi-dimensional time model can be fully represented as a loop nest of depth d of the following general form, where the instruction of the innermost loop serves to output the element value associated to a time instant $(t_1, t_2, ..., t_d)$:

```
for t1 = 0 to n
    for t2 = l(t1) to u(t1)
        for t3 = l(t1,t2) to u(t1,t2)
            ...
                for td = l(t1,t2,...,td-1) to u(t1,t2,...,td-1)
                    f(t1,t2,...,td);
```

where $f(t_1, t_2, ..., t_d)$ is the final multi-variable function resulting from the d-depth recursive application of the model. This function is linear relatively to each variable t_i and globally non-linear. Moreover if d is maximum, i.e., the model has been applied as far as possible, then the function is no longer periodic, since any period associated to a time dimension t_i is now expressed as a loop index.

The recursive application of the model terminates as soon as the computed coefficients a and b are no more periodic numbers, i.e., are reduced to one single value. In the case where no more interpolation of any coefficients a or b is possible, with a or b consisting in at least three values, coefficients are decomposed into phases as explained in the next subsection. However the application depth can be fixed according to a chosen analysis granularity. All time dimensions can be seen as different granularity levels of the behavior model.

Functions $l(t_1, t_2, ..., t_i)$ and $u(t_1, t_2, ..., t_i)$ give the sizes of the interpolated intervals at depth $i+1$. When the model consists in constant size intervals, these functions are constants: $l = 0$ and $u+1$ is equal to the period of the interpolation function at depth $i+1$. When the model consists in non-constant sized intervals, functions $l(t_1, t_2, ..., t_i)$ and $u(t_1, t_2, ..., t_i)$ are functions interpolating the positions of the first, respectively the last, elements in all the intervals. Although these functions are generally not linear functions we limit ourselves to the cases where they are affine.

Example 1. Consider again the time-serie $[3, 3, 7, 13, 11, 23, 15, 33, 19, 43, 23]$. Following the model of adjacent intervals of constant sizes, at the first time dimension, it can be interpolated by $f_1(t_1) = [4, 10]t_1 + [3, 3]$, $0 \leq t_1 \leq 5$. At the second dimension, serie $[4, 10]$ and $[3, 3]$ are interpolated respectively by the functions $f_a(t_2) = 6t_2 + 4$ and $f_b(t_2) = 0t_2 + 3$, $0 \leq t_2 \leq 1$. Hence the recursive process stops and the following loop nest can be generated:

for $t_1 = 0$ to 5
    for $t_2 = 0$ to 1
        $(6t_2 + 4)t_1 + 3$;

2.3 Program Phase Intervals

In our model, we define phases as the largest adjacent slices of the trace allowing periodic-linear interpolations of their elements. Hence successive phases can occur at different granularity levels yielding a hierarchy of phases. This can be represented as successive loops whose loop indices range from the first to the last element of each phase, and where each loop contains itself successive loops associated to inner level phases and so on.

The size of the generated program can be seen as a complexity measure of the modeled input trace, in the same way as it is stated in the Kolmogorov complexity theory [6].

3 Model Construction Algorithms

Our algorithms have been implemented and can obviously represent a high computation cost for large and highly irregular input traces. However modeling a

very regular behavior with a few number of phases is fast. For example, each memory access information considered in [3] in several benchmarks are instantaneously modeled by our tool while giving similar results. Anyway it is generally worth the time to model critical systems behavior.

In the following and due to space limitation, only the algorithm dedicated to adjacent intervals of constant size is presented.

3.1 Quality Criteria of the Model

Since our model is hierarchically organized as a multi-dimensional time-space, the deeper we go into the hierarchy, the more accurate is the model. Interpolations involving a minimum number of phases per level is preferred since it corresponds to a minimum number of general behaviors associated to each current levels. Hence between the four model alternatives presented in subsection 2.2, a preference order related to the regular layout of the model and the number of phases is applied: (1),(2),(3) and (4).

A convenient number of phases is related to their different sizes and the size of the input trace. Each phase must include a sufficient number of elements. However since phases are related between each other through the whole interpolation model, some interesting and large phases can be coupled with some small phases. Hence an opportune quality criterion can just consider the large phases.

On the other hand, some solutions with only a few phases have to be evicted. For example, the solution consisting in modeling the whole input trace as two half traces interpolated by one periodic-linear function is obviously not interesting and does not represent any behavior specificity, since the same can be done for any sequence of numbers. Hence we constraint each phase to contain at least three interpolated intervals. Moreover, between several possible solution phases of the same size modeling the same elements, the phase containing the maximum number of interpolated intervals is selected, since it involves a lower periodicity of the interpolation function, each period corresponding to a larger number of interpolated elements.

From these observations, we can define an heuristic criterion consisting in a lower bound for the covering range of the phases. For example, we can state that at least 80% of the input trace has to be covered by all the phases whose sizes are greater than 5% of the input trace.

3.2 Phase Detection

Our model construction algorithms have the following general scheme:

1. find a periodic linear interpolation function covering the largest possible slice of the trace with at least three interpolated intervals. Define this slice as a program phase.
2. for all the remaining elements not belonging to the previously defined phases, repeat the previous step in order to define more program phases.
3. at this step, a hierarchical level has been fully modeled.

4. for each of the previously defined phases and their associated periodic-linear interpolation functions, repeat all steps with each of the periodic coefficients a and b considered themselves as traces, thus defining a deeper hierarchical level.

3.3 Adjacent Intervals of Constant Size

The frequency of linear relations between p-spaced elements in a trace can be detected by computing the autocorrelation coefficients for several values of p. The highest obtained coefficients, *i.e.*, the closest to 1, associated to given values of p, give some good indications on the best possible sizes, or periods, of the interpolated intervals. Hence our algorithm tries successively all interval sizes from their highest to their lowest associated autocorrelation coefficients in order to find the largest phase of interpolated intervals. Since at least three intervals have to take part in a phase, autocorrelation coefficients for all values of p less than $n/3$ are computed, n being the number of elements in the input trace.

The general algorithm is shown in figure 2. It is defined as a recursive function devoted to finding the largest phase of interpolated intervals from a given input trace. As it has been found, the function is recalled to find some previous or next phases necessarily smaller and covering the remaining parts of the trace. Some added comments in the figure explains some further details. When phases have been found covering the whole trace, the first time dimension has been defined and the next step consists in applying the function find_phase for each phase and their periodic-linear interpolation function to the periodic coefficients a and b. This will define the second time dimension. The same process is applied recursively until there is no remaining phase interpolated by a periodic function, i.e., the interpolation function has constant coefficients a and b.

4 Application Examples and Data Prefetching

In these experiments, we model memory addresses accessed by some time-consuming program instructions. We initially profile the execution using the *gnu gprof* tool in order to exhibit the most time consuming functions. We then instrument the code in order to store the accessed virtual memory addresses in output files. Those files are then used as input in our model construction algorithms.

4.1 Building an Hybrid Model

Hybrid models can be constructed from the observation of some input dependent events mixed with input independent events. Through abstraction of the input dependent events that cannot be modeled, a model characterizing some linear and periodic behavior of these non-deterministic occurring events can be constructed.

We consider the program `ks` from the pointer intensive benchmarks, and model memory addresses accessed through the pointer `mrB` in the most time-consuming function `FindMaxGpAndSwap`. We observe the following in the trace

function find_phase(T: input trace) {n is the trace size}
$phase_{max} = NULL$
for all p such that $1 \leq p \leq n/3$ **do**
  compute the autocorrelation coefficient r_p of order p
for the highest to the lowest coefficient r_p and its associated period p, $r_p \geq 0.1$ **do**
  for i=1 to p **do**
    find the lowest integer value $\alpha \leq \frac{n-3p}{p}$ such that at least the 3 elements $T[i+\alpha p]$, $T[i+(\alpha+1)p]$ and $T[i+(\alpha+2)p]$ are linearly dependent
    while a value α has been found **do**
      for all q such that $1 \leq q \leq p-1$ **do**
        check if elements $T[i+\alpha p+q]$, $T[i+(\alpha+1)p+q]$ and $T[i+(\alpha+2)p+q]$ are also linearly dependent
      if so **then**
        extend this sequence of intervals to the right to the maximum possible size
        if size($phase_{current}$) > size($phase_{max}$) {the last found phase is the largest at the moment}
        OR (size($phase_{current}$) = size($phase_{max}$) AND $p_{current} < p_{max}$) {the last found phase has smaller interpolated intervals} **then**
          $phase_{max} = phase_{current}$;$p_{max} = p_{current}$
        find the next greater value for $\alpha \leq \frac{n-3p}{p}$

{the following is useful to find the last phases from a few remaining elements:}
if $phase_{max} = NULL$ {no phase with at least 3 intervals has been found} **then**
  allow to select phases with less than 3 elements
build the periodic-linear interpolation function $f_{phase_{max}}$ of coefficients a and b and of period p_{max}
let T_{left} be the left part of $T - phase_{max}$;let T_{right} be the right part of $T - phase_{max}$
{recursive calls}
find_phase(T_{left});find_phase(T_{right})

Fig. 2. The general algorithm to find phases of adjacent interpolated intervals of constant sizes

of memory addresses: same sequences of addresses are accessed successively several times; after a sequence has been accessed, a new sequence, being the same as before but with one element less, is accessed again successively several times; after the last sequence of one element has been accessed, a completely different sequence of `numModules/2` elements is accessed successively several times, `numModules` being the second value in the input file; then the same process goes on with the same sequence having one element less; the number of times a sequence is accessed successively is equal to its number of elements; values into a sequence cannot be interpolated linearly and periodically; elements evicted from a sequence cannot be determined and depend on the considered input.

In conclusion, non-predictable sequence of known sizes are accessed in a predictable manner. Hence an hybrid model can be constructed consisting in a learning phase storing occurring address sequences, and a following prediction phase that outputs in a exact way occurring addresses. This model can be represented as the loop nest shown in figure 3.

```
M = numModules/2 − 1
for t1 = 0 to N
 for t2 = 0 to M
   for t3 = 0 to 0 // * Learning phase *
     for t4 = 0 to M − t2
       T[t4] = accessed address ;// store values in an array of size M
   for t3 = 1 to M − t2 // * Prediction phase *
     for t4 = 0 to M − t2
       T[t4] ;
```

Fig. 3. Hybrid model capturing program `ks` memory behavior

4.2 Using Models for Data Prefetching

Let us consider the program `mcf` from the Spec2000 benchmarks: 31% of the whole running time is spent in function `price_out_impl`. By looking at the source code of this function, we can see that two main instructions access some data structures defined as chained lists. We then instrument the code in order to store the accessed virtual memory addresses in two output files and run the program using the `test.in` input file provided in the SPEC2000 benchmarks.

For both memory accesses, an hybrid model of adjacent intervals of different sizes, enclosed in adjacent intervals of constant size is constructed. In the first dimension t_1, the constant size intervals are identical. In the second dimension, successive intervals sizes grow from one element at each interval, and corresponding elements between successive intervals are spaced by 120 for the first instruction, and by 192 for the second instruction.

In a given interval, values are decreasing by 120, or 192, until 0. Hence, a 3-dimensional model represents entirely the whole trace. It is stated by the loop nest shown in table 1.

Table 1. Nested loop models, execution times and speedups

Program: opt. function	Model	Orig. time	Opt. time	Speedup
mcf: price_out_impl	for $t_1 = 0$ to M for $t_2 = 0$ to $nb\_timetable\_trips - 2$ for $t_3 = 0$ to t_2 $120t_2 - 120t_3 + offset$;	512 sec.	405 sec.	20%
equake: smvp	for $t_1 = 0$ to $timesteps - 1$ for $t_2 = 0$ to $N - 1$ for $t_3 = 0$ to 2 $128t_2 + 32t_3 + offset$;	350 sec.	262 sec.	25%

Values M and N vary depending on the program input file. Value $M + 1$ denotes the number of constant size intervals in the first dimension and $N + 1$ denotes their size. We use the three input files provided in the SPEC2000 benchmarks, `test.in`, `train.in` and `ref.in`, and analyze the generated traces

to extract the associated values of M and N, by checking the model adequacy. We notice that interval sizes are directly given by the first input parameter of the input files. In the `mcf` program documentation, this parameter is defined as being *the number of timetabled trips*. It is equal to $N + 2$. The number of intervals cannot be linked directly to an input parameter, since it rather depends qualitatively on the convergence speed of the implemented optimisation process for the considered problem. Nevertheless, a generic model can still be described, since values in dimensions t_2 and t_3 do not depend at all on t_1. Moreover, the use of the model for some dynamic optimization is not constrained at all by the ignorance of M, since the optimization process runs until the end of the whole program run.

We use both generated models to implement a dynamic prefetching mechanism for improving the program performance on an *Itanium-2* processor. This mechanism is simply built as two functions prefetching data three accesses in advance from the address computed due to our model. They are called before each memory access in function `price_out_impl`. Significant speedups are obtained for reference input runs of the whole program as it is shown in table 1. Original and optimized programs have been compiled at O3 optimization level.

In the same way, we model the program `equake` from the SPEC2000 benchmarks as shown in table 1.

Notice that other optimizations could have been constructed from these models as for example the generation of cache hints from the knowledge of data-reuse distances due to our models, as it is done from static analysis in [1].

5 Conclusion

The presented dynamic analysis and modeling approach constitutes a rich framework to formalize behavior capture of programs. The representation model facilitates program behavior understanding and analysis, and also allows the construction of efficient static or dynamic optimizations. It is for example pleasant to notice that array-like memory accesses are identified through our model, as it generates an access function of the same form as an access function resulting from a linearized multi-dimensional array accessed through affine functions indices. We argue that a lot of important behavior characteristics can be handled through our approach: as we were working on the experiments of the previous section, we observed that a lot of memory instructions can be nicely modeled. Moreover, even non-deterministic events can be considered as they are enclosed in a behavior that can be represented.

Our immediate objective is to improve performance of our algorithms in order to build a global profiling and modeling system. Such a system could then advantageously be used for applications whose performance or behavior control are critical.

References

1. K. Beyls and E. H. D'Hollander. Reuse distance-based cache hint selection. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 265–274. Springer-Verlag, 2002.
2. J. Gustafsson, B. Lisper, R. Kirner, and P. Puschner. Input-dependency analysis for hard real-time software. In *Proc. 9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Oct. 2003.
3. I. Issenin and N. D. Dutt. Foray-gen: Automatic generation of affine functions for memory optimizations. In *DATE*, pages 808–813, 2005.
4. J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.
5. J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *11th International Symposium on High Performance Computer Architecture*, February 2005.
6. M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, New York, 1993.