A Dynamic Periodicity Detector: Application to Speedup Computation

Felix Freitag, Julita Corbalan, Jesus Labarta

Departament d'Arquitectura de Computadors (DAC),Universitat Politècnica de Catalunya(UPC)

{felix,juli,jesus}@ac.upc.es

Abstract

We propose a dynamic periodicity detector (DPD) for the estimation of periodicities in data series obtained from the execution of applications. We analyze the algorithm used by the periodicity detector and its performance on a number of data streams. It is shown how the periodicity detector is used for the segmentation and prediction of data streams. In an application case we describe how the periodicity detector is applied to the dynamic detection of iterations in parallel applications, where the detected segments are evaluated by a speedup computation tool. We test the performance of the periodicity detector on a number of parallelized benchmarks. The periodicity detector correctly identifies the iterations of parallel structures also in the case where the application has nested parallelism. In our implementation we measure only a negligible overhead produced by the periodicity detector. We find the DPD to be useful and suitable for the incorporation in dynamic optimization tools.

1 Introduction

In dynamic performance, analysis of applications the measurement and evaluation is done during the application's execution. Measuring the application performance requires the monitorization of certain parameters, such as for instance, subroutine calls, hardware counters, or CPU usage [Corbalan99]. During the execution of the application the values of these parameters represent a data stream to the measuring tool. The measuring tool computes the value of certain performance parameters associated with the conditions in each measuring interval. Based on the run-time performance of the application, decisions can be taken dynamically leading to dynamic optimization.

Dynamic performance measurement of applications is useful for several purposes. Based on the run-time performance of applications the resources of the system may be allocated differently. In [Corbalan2000] [NguyenZV96] authors propose to consider the run-time calculated application efficiency to perform the processor allocation. Moreover, performance measurements can be useful to dynamically optimize the application execution. In [Voss99][VossEigenmann99] authors propose to perform several run-time tests to dynamically detect data dependences, serialize parallel loops with great overheads, and to perform tiling.

The knowledge about the periodicity in data streams can be useful for different applications in dynamic performance analysis: 1) Knowing the periodicity of patterns can be used to perform the dynamic segmentation of the data stream in periods. Periods in a data stream or multiples of them may represent reasonable intervals for performance measurement. 2) If the detected period represents the execution of an iterative part of an application, then measuring the performance in one period can be used to predict the performance in future iterations without need to have a continuous measurement. 3) Given the periodicity of a data stream, future parameter values can be predicted.

2 Background

The data series which need to be analyzed with the described periodicity detector could be obtained in different ways. For certain parameters a reasonable method to obtain a data stream can be the sampling of the parameter value with a certain sampling frequency. For other kinds of parameters only the changes of the parameter value may be interesting to register. So the data stream is obtained if the magnitude of the parameter value changes.

We expect to observe in the data streams of several parameters of scientific applications locally periodic patterns, since applications often spend a large part of their execution time in loops which provide data series with repeating patterns [Sazeides98]. A frame based detector for the analysis of the data series shall therefore be appropriate to detect locally periodic patterns.

In the next section we how the proposed periodicity detector works with both of the above mentioned types of data series.

3 Dynamic Periodicity Detection

3.1 Periodicity Detection Algorithm

We propose the application of the periodicity detector to data series which contain segments with periodic patterns. We consider the periodicity of the data stream which we are interested to detect being the fundamental period it contains, where its amplitude is of larger magnitude than that of other frequencies contained in the data stream.

For the periodicity detector we use an adaptation of the distance metric given in equation (1), see Figure 1,

$$d(m) = \frac{1}{N}\sum_{n=0}^{N-1}|x[n] - x[n-m]| \qquad (1)$$

Figure 1: Distance metric.

[Deller87]. In equation (1) N is the size of the data window, m is the delay (0 < m < M), M<=N, x[n] is the current parameter value of the data stream, and d(m) is the value computed to detect the periodicity. It can be seen that equation (1) compares the data sequence with the sequence shifted m samples. Equation (1) computes a distance between two vectors of size N by summing the magnitudes of the L1-metric distance of N vector elements. The value d(m) becomes zero if the data window contains a identical periodic pattern with periodicity m. If d(m) > 0, then the two vectors **x**[n] and **x**[n-m] of equation (1) are not identical. However, the magnitude of d(m) provides a distance between the two vectors. Then, the value of m for which d(m) has a local minimum is considered the periodicity detected in the data frame.

According to equation (1), if the periodicity m in the data stream is several magnitudes less than the size N of the data window, then the value d(m) may become zero for multiples of m. On the other hand if the periodicity m in the data stream is larger than the data window size N, then the pattern and its periodicity cannot be captured by the detector. In this case the periodicity is not obtained since d(m) does not have a local minimum nor will d(m) become zero.

In certain data series the sample values do not represent meaningful magnitudes such as for instance when processing a sequence of events. Then, the implementation of the periodicity detector is made using equation (2), see Figure 2. If d(m)=0, then a periodic pattern with dimension m is detected in the data stream. The sign() function is used in equation (2) to set the value d(m) to 1 if it is not zero. If d(m) is zero, then the two patterns are identical. For all m where d(m) is not zero the patterns are not identical and no periodicity in m is indicated.

$$d(m) = sign\sum_{i=0}^{N-1}|x(i) - x(i-m)| \quad (2)$$

Figure 2: Distance metric used for data series consisting of events.

On the data series obtained from the execution of applications we experimented with we found that most periodicities are less than 100 samples, for which N=100 is sufficient. For some data series the size of the data window can be less than N=10, if very short periodicities appear (see section 6). We also used the periodicity detector with window sizes up to N=1024. With this setting periods with a length of up to 1023 samples can be detected. Implementation issues concerning memory requierements are given in [Freitag00]. In most cases the periodicities in the data series we found were small (see examples in this paper). For an unknown data stream, the window size N of the periodicity detector should be set initially to a large value, in order to be able to capture large periodicities. Once a satisfying periodicity is detected, the window size may be reduced dynamically (see section 4).

3.2 Using the periodicity detector on the parallelism of applications

In order to test and illustrate the performance and possible applications of the periodicity detector we have carried out a number of experiments. We show some of them in the following sections. More applications of the periodicity detector are shown in [Freitag00].

We apply the periodicity detector on a trace which represents the instantaneous number of active CPUs used by a parallel application, see Figure 3. The trace is from the FT application of the NAS benchmarks. The parallel application is MPI/OpenMp. Each process has a number of threads and messages are interchanged between the MPI processes. The application is executed in the NANOS environment [Nanos97] on a SGI Origin 2000 multi-processor platform. The sampling frequency of the CPU usage is set to 1 ms. It can be observed in the trace that during the execution of the application the parallelism is opened and closed a few times. Up to 16 CPUs are used by the applications in parallel. By visual inspection a periodic pattern in the CPU usage can be observed. Also, it can be noted that the pattern of CPU use is not exactly the same during the application's execution.

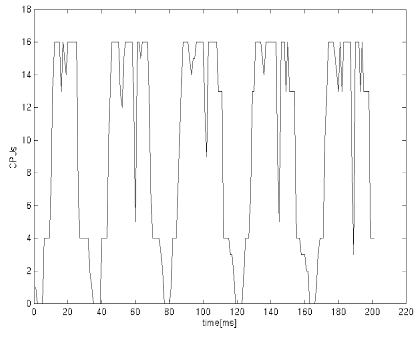


Figure 3: Number of CPUs used during the execution of a parallel application.

The periodicity detector is applied to this data sequence. The periodicity detector computes the values d(m). In Figure 4 the values of d(m) are shown. The periodicity is detected for the value m where d(m) has a local minimum. It can be seen that d(m) has a local minimum at m = 44. The periodicity detector indicates periodicity m = 44 in the data stream.

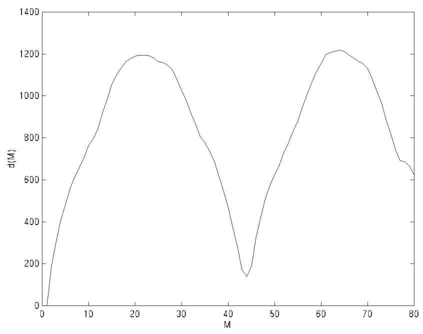


Figure 4: Estimation of the period with the periodicity detector. Periodicity m=44 samples.

4 Interface implementation

We have implemented the periodicity detector providing the interface shown in Table 1 . Two functions are implemented in the interface. The DPD function is the main function used for periodicity detection and segmentation. The DPDWindowSize function is an additional function which allows to modify the data window size of the DPD during the application's execution.

Table 1: Interface of the DPD implementation.

Interface	Description
int DPD (long sample, int *period)	Periodicity detection and segmentation
void DPDWindowSize (int size)	Adjust data window size

5 Case Study: Dynamic Performance Analysis

In this section we describe the integration of the DPD in the *SelfAnalyzer* [Corbalan99]. The *SelfAnalyzer* is a mechanism that dynamically analyzes the performance of parallel applications. This mechanism uses the DPD to automatically detect periodic patterns of parallel regions in parallel applications.

The *SelfAnalyzer* is a run-time library that dynamically calculates the speedup achieved by the parallel regions of the applications, and estimates the execution time of the whole application. The *SelfAnalyzer* exploits the iterative structure of a significant number of scientific applications. The main time-consuming code of these applications is composed by a set of parallel loops inside a main sequential loop. Iterations of the sequential loop have a similar behavior among them. Then, measurements for a particular iteration can be used to predict the behavior of the next iterations.

The *SelfAnalyzer* instruments the application and measures the execution time of each iteration of the main loop. The speedup is calculated as the relationship between the execution time of one iteration of the main loop, executed with a *baseline* number of processors, and the execution time of one iteration with the number of available processors.

To calculate the speedup, the *SelfAnalyzer* needs to detect the following points of the code: the starting of the application, the iterative structure, and the start and end of each parallel loop. In the current implementation, the invocation of the *SelfAnalyzer* at these points can be done in two different ways: (1) if the source code is available, the application can be re-compiled and the *SelfAnalyzer* calls are inserted by the compiler. (2) If the source code is not available, both the iterative structure and the parallel loops are dynamically detected.

5.1 Using the DPD to automatically detect iterative parallel structures

Compilers that process OpenMp directives typically encapsulates code of parallel loops in functions. Figure 5 shows the iterative parallel loop, once encapsulated. These functions are called by the application threads, and each one executes their range of iterations at runtime. Using dynamic interposition (DITools [Serra2000]) the calls to encapsulated parallel loops are intercepted.

```
do
  call omp_parallel_do_1(..)
  call omp_parallel_do_2(...)
  call omp_parallel_do_3(...)
end do
```

Figure 5: Encapsulated parallel loops

Each parallel loop is identified by the address of the function that encapsulates it. In this case we are only interested in detecting periodic patterns in sequences of calls to parallel loops, then the address of parallel loops is the value that we pass to the DPD. The iterative parallel region of the application can be nested, it can include more than one periodic pattern, with different starting points and lengths. In the current implementation calls to parallel loops are passed through the DPD mechanism, since we do not have any knowledge of the application structure and no assumptions can be made about the periodicities inside the application.

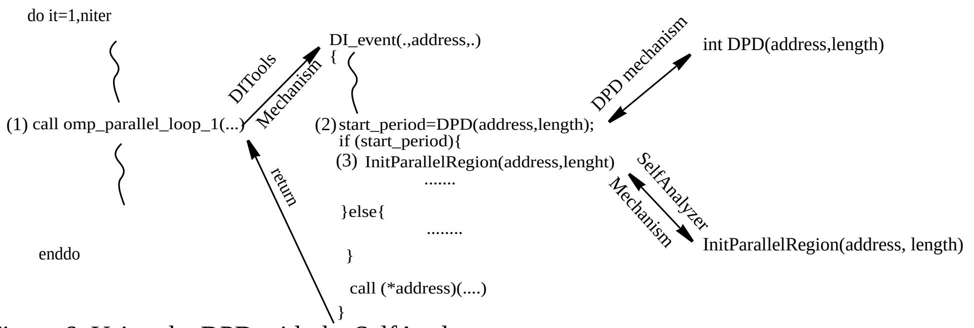
Figure 6 shows the three mechanism working at runtime. In (1) the parallel loop generation is intercepted through the DITools. Inside the code that process this event, the DPD is called in (2). If the value of *address* is the start of a period, the DPD returns a value different from zero. In that case the *SelfAnalyzer* is called (3). The *SelfAnalyzer* identifies a parallel region with the address of the starting function and the length of the period indicated by the DPD. We have adopted this solution, rather than using the complete sequence of loops that compound the period, by simplicity and assuming that the case of two iterative sequences of values with the same length and same initial function is not a normal case. In fact, this case has not appear in the applications we have processed till the moment. With these two mechanism, DITools and DPD, the *SelfAnalyzer* can be applied to those applications that do not have their source code available. The speedup calculated can be used to improve the processor allocation scheduling policy, providing a great benefit as we have shown in [Corbalan2000].

6 Evaluation

In order to evaluate the integration of the DPD mechanism in the *SelfAnalyzer* we have performed two kinds of experiments. The first experiment demonstrates the effectiveness of the DPD detecting periodic patterns in the applications. In the second experiment we measure the overhead introduced by the DPD. We want to demonstrate that the DPD is able to detect the periodic patterns inside the application without introducing a significant overhead.

6.1 Applications

We have selected five applications from the SPECFp95 benchmark suite: tomcatv (ref), swim (ref), hydro2d (ref), apsi (ref), and turb3d (ref). These applications have been parallelized by hand using OpenMp directives. Tomcatv, swim, and apsi have only one periodicity. hydro2d and turb3d have several nested iterative parallel structures.



```
do it=1,niter

(1) call omp_parallel_loop_1(...)

enddo
```

```
DI_event(.,address,.)
{
(2)start_period=DPD(address,length);
   if (start_period){
(3)  InitParallelRegion(address,lenght)
       .......
   }else{
       ........
   }
   call (*address)(....)
}
```

```
int DPD(address,length)
```

```
InitParallelRegion(address, length)
```

Figure 6: Using the DPD with the SelfAnalyzer

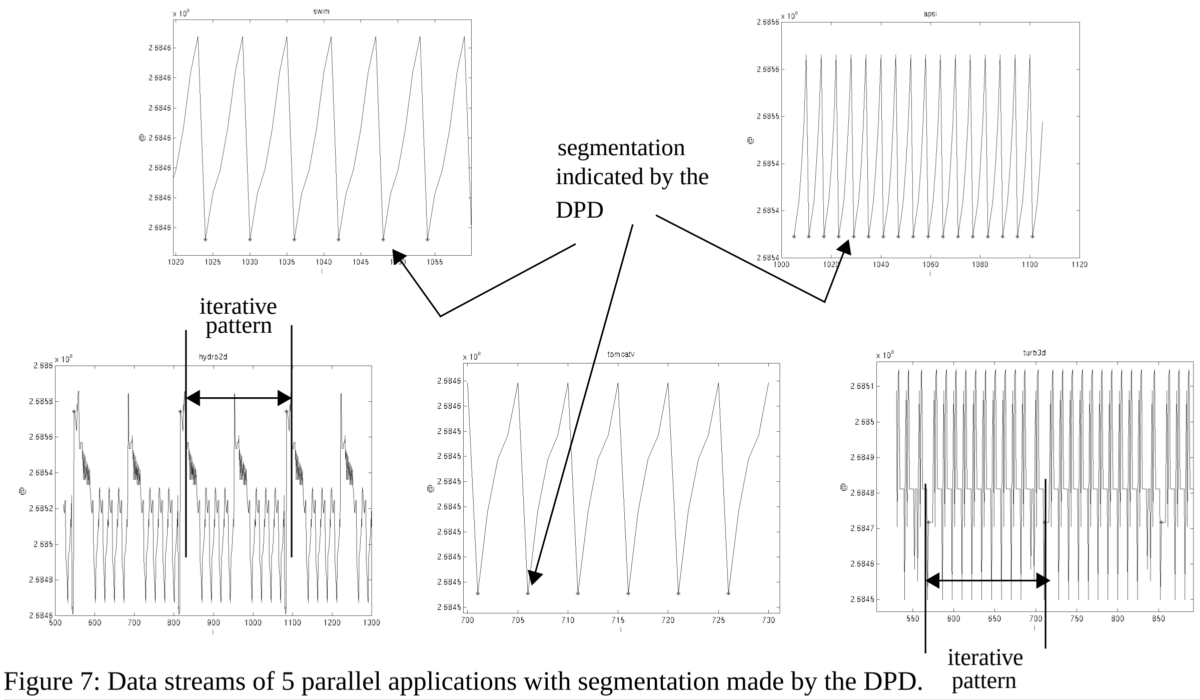


Figure 7: Data streams of 5 parallel applications with segmentation made by the DPD.

6.2 Dynamic periodicity detection in applications

In Table 2 some characteristics of the applications and the periodicities detected by the DPD are presented. Figure 7 shows a small part of the data streams and its segmentation by the DPD (the segmentation made by the DPD is marked by "*"). The data stream is a sequence of addresses. In the y-axis of the graphics the value of the addresses is indicated. Note that the data streams are drawn as continuous values in order to better observe the periodicity of the patterns. It can be seen that in the applications tomcatv, swim, and apsi the DPD correctly identifies the periodicity of the patterns with periodicity 5, 6, and 6, respectively. The data streams from the applications turb3d and hydro2d contain nested iterative structures. It can be seen in Figure 7 that the data streams of these applications contain a large iterative pattern within which smaller iterative patterns appear. Also for these applications with nested parallelism the DPD correctly identifies the periodicity of the large iterative pattern (periodicity=269 for hydro2d and periodicity=142 for turb3d).

6.3 Overhead

We have calculated the cost of processing the values by the DPD. A synthetic benchmark has been executed to measure this cost. This synthetic benchmark reads a trace file that corresponds to the execution trace of one application, and it calculates its periodicity. The synthetic benchmark measures the execution time consumed by processing the trace and calculates the cost of processing each value of the trace. Table 3 shows the results on the five SPECfp95 applications. *NumElems* column is the number of elements in the trace file, *ApExTime* column is the execution time of the benchmark in sequential, without the DPD mechanism, *TimeProc* column is the execution time (in seconds)

Table 2: Detected periodicities.

Appl.	Data stream length	Detected periodicities
Apsi	5762	6
Hydro2d	53814	1, 24, 269
Swim	5402	6
Tomcatv	3750	5
Turb3d	1580	12, 142

Table 3: Overhead analysis

	Num Elems	ApEx Time(sec)	Time Proc (sec)	Perc.	Timex Elem(ms)
tomcatv	3750	136.33	0.016678	0.012%	0.004
swim	5402	135.17	0.023476	0.017%	0.004
apsi	5762	95.9	0.025169	0.026%	0.004
hydro2d	53814	183.92	6.028188	3.27%	0.112
turb3d	1580	266.44	0.171326	0.064%	0.108

consumed in processing the trace by the DPD, *Percentage* column is the percentage of time consumed processing the trace with the DPD in relation to the execution time of the application, that is *Percentage*=*TimeProc/ApExTime**100, and *TimexElem* column is the execution time consumed each time the DPD is called (in milliseconds).

We observe in Table 3 that the overhead introduced by the DPD is very small compared to the execution time of the benchmark, see columns *ApExTime* and *TimeProc*. The additional load to execute the DPD along with the application showed to be negligible, see column *Percentage*. We conclude that the additional cost of running the DPD was very small, which allows to use the DPD during program execution in dynamic optimization tools.

7 Conclusions

We have presented a technique to dynamically detect the periodicity in data streams and some of its applications. The performance of the proposed mechanism is evaluated on a number of data streams and applications. We incorporate the Dynamic Periodicity Detector (DPD) in a tool for speedup computation, the *SelfAnalyzer* and show the interface of the DPD we implemented. We obtain that the DPD correctly identifies the iterative parallel structure in the applications and provides the segmentation of the data stream to the *SelfAnalyzer.* It is found that the overhead added by using the DPD dynamically is negligible. We observe in this application case that the DPD mechanism is useful and suitable to be incorporated in a dynamic optimization tool.

8 Acknowledgements

This work has been supported by the Spanish Ministry of Education under grant CYCIT TIC98-0511 and the Direcció General de Recerca of the Generalitat de Catalunya under grant 1999FI 00554 UPC APTIND. The research described in this work has been developed using the resources of the European Center for Parallelism of Barcelona (CEPBA).

9 References

[Amdahl67] G. M. Amdahl, “Validity of the single processor approach to achieving large-scale computing capabilities”, in Proc. AFIPS, vol. 30, pp. 483-485, 1967.

[Corbalan99] J. Corbalan, J. Labarta, “Dynamic Speedup Calculation through Self-Analysis”, Tech. Report UPC-DAC-1999-43, Dep. d’Arquitectura de Computadors, UPC, 1999.

[Corbalan2000] J. Corbalan, X. Martorell, J. Labarta, “Performance-Driven Processor Allocation”, in Proc. of the 4th Symposium on Operating System Design & Implementation (OSDI2000), 2000.

[Deller87] J. R. Deller, J. G. Proakis, J. H. L. Hansen. “Discrete-time processing of speech signals”, Prentice Hall 1987.

[Eager89] D. L. Eager, J. Zahorjan, E. D. Lawoska, “Speedup Versus Efficiency in Parallel Systems”, IEEE Transactions on Computers, vol. 38,(3), pp. 408-423, March 1989.

[Freitag00] F. Freitag, J. Corbalan, J. Labarta, “A Dynamic Periodicity Dectector: Application to Speedup Computation”, Tech. Report UPC-DAC-2000-58, Dep. d’Arquitectura de Computadors, UPC, 2000.

[Nanos97] Nanos Consortium, “Nano-threads Programming Model Specification”, ESPRIT Project No. 21907 (NANOS), Deliverable M1.D1, July 1997.

[NguyenZV96] T. D. Nguyen, J. Zahorjan, R. Vaswani, “Using Runtime Measured Workload Characteristics in Parallel Processors Scheduling”. *JSPP*, vol. 1162 of *Lectures Notes in Computer Science. Springer-Verlag*, University of Washington, 1996.

[Sazeides98] Y. Sazeides, J. E. Smith, “Modeling Program Predictability”, in Proc. ISCA, pp. 73-84, 1998.

[Serra2000] A. Serra, N. Navarro, T. Cortes, “DITools: Application-level Support for Dynamic Extension and Flexible Composition”, in Proc. of the USENIX Annual Tech. Conf., pp. 225-238 , June 2000.

[Voss99] M. Voss, R. Eigenmann, “Dynamically Adaptive Parallel Programs”, Int’l. Symp. on High-Performance Computing, pp. 109-120, Japan 1999.

[VossEigenmann99] M. Voss, R. Eigenmann, “Reducing Parallel Overheads Through Dynamic Serialization”, Int’l. Parallel Processing Symposium, pp. 88-92, 1999.