A MODIFIED ENTROPY ALGORITHM FOR PARALLEL MONITORING

Paralel Monitörlerde Modifiye Edilmiş Entropi Algoritması

Ahmet ÖZMEN*

ABSTRACT

Performance evaluation and debugging of parallel systems rely on instrumentation that traces or profiles program behavior. Software instrumentation that is statically inserted into the program source, run-time libraries, or Operating System can gather redundant information. The overhead incurred during run time to process and record this information can increase the execution time of the program and can even change program behavior. Static configuration of instrumentation, although simple, is not in general efficient in terms of information gathered to overhead introduced. Information content of instrumentation data typically depends on uncertainty of associated event to occur. This paper presents a dynamic algorithm for software instrumentation to measure the amount of information content of instrumentation data to be collected. This makes intelligent data collection for a software monitoring system possible. The algorithm can be used in software instrumentation to make monitoring system less intrusive looking at the information content of instrumentation data before collected.

ÖZET

Paralel sistemlerin izlenmesinde ve program davranışının belirlenmesinde yapılan performans analizi ve problem giderilmesi çalışmaları enstrumentasyona bağlıdır. Statik olarak programa, kütüphaneye veya işletim sistemine ilave edilerek yapılan yazılım enstrumantasyonu lüzumsuz bilgi toplayabilir. Çalışma anında; bu verileri toplarken, diske kaydederken programın icra süresi artabilir, hatta davranışı bile değişebilir. Statik enstrumantasyon kontrolü kolay olmasına rağmen, toplanan bilginin harcanan zamana oranı dikkate alındığında pek verimli değildir. Bir enstrumantasyon verisinin taşıdığı bilgi, o olayın olup olmayacağının kesin olmamasına bağlıdır. Bu çalışmada, yazılım enstrumantasyonu sistemlerinde toplanan performans verilerinin değerlendirildiği dinamik bir algoritma sunulmuştur. Böylece yazılım monitörleri için akıllı veri toplama sistemleri mümkün olacaktır. Bu algoritma yazılım enstrumantasyon sistemlerinde kullanılarak performans verisinin içeriğine bakılmak süratiyle, program icrasını daha az rahatsız eden monitör sistemler yapılabilir.

Key Words: Software Performance Monitoring, Parallel Processing.

* Dumlupınar Üniversitesi, Mühendislik Fakültesi, Elektrik-Elektronik Mühendisliği, Kütahya

1.INTRODUCTION

Performance evaluation of parallel programs rely on monitoring run-time behavior. Monitoring requires hardware or software instrumentation to capture run-time data from the concurrent processes. Flexibility and portability make software instrumentation the widely used alternative. However, it is intrusive because instrumentation introduces overhead that perturbs the behavior of the original programs [5][4]. The large volume of information gathered by a static software instrumentation is a problem because storing, processing and presenting it consumes system resources such as memory, disk space and CPU time. Our goal is to develop a dynamic instrumentation system that maximizes the amount of information gained via instrumentation while minimizing the amount of overhead incurred due that instrumentation. Although overhead can be measured with time, no techniques have been introduced to measure the amount of information in performance evaluation.

This paper presents an algorithm called *entropy based instrumentation*. The original entropy algorithm is modified to evaluate the information content of performance data. The algorithm can be used to observe parallel execution progress, and to detect program phases.

The concept of entropy is introduced as a measure of uncertainty of a random variable in the Information Theory [7][8]. We use this concept to evaluate information content of instrumentation data. Two aspects of the algorithm are:

- All the events that will occur during the execution can be known priori, but the exact order is not known.
- The uncertainty of events occurrence increases information content of its data.

From the trace history and the current program status, we can calculate the information content of the current instrumentation data. This happens every time execution progresses and a new event occurs. Hence progress of the program can easily be observed from entropy values. The following sections show how this algorithm is implemented and can be used to detect program phases in a parallel monitoring environment.

2.ENTROPY

Entropy, H, is defined in Equation 1, where K is a positive constant and n is the number of possible events whose possibilities of occurrence p_i.

$$H = -K \sum_{i=1}^{n} p_i \log (p_i) \qquad (1)$$

Since $\sum_{i=1}^{10} p_i = 1$, it can be shown that $0 \le H \le \log (n)$. The units in which the entropy is measured depend on the base of the logarithm used in the definition. *Bit* and *nats* are used for base *2* and *e* respectively.

The entropy H can also be interpreted as the average amount of information that a message contains [2][3]. Suppose there is a message which could be either a_1 or a_2 with probabilities $p_1 = 1$ and $p_2 = 0$ respectively; the entropy H is *0*, which means the message contains no new information. At the other extreme, suppose $p_1 = p_2 = 1/2$. The entropy is then $H = 1$ bit. Receiving the message clearly adds new information.

3.THE ALGORITHM

There are two concepts in the algorithm: a window that holds a sequence of events that happened in the past, and a probability of transition scheme that describes the possibility of an event sequence occurrence. An event in the right most position of the window is called the *most recent event*, and an event which has just occurred and is not in the window is called the *destination event*. Concerning a monitoring system a destination event represents an event to be decided either to be processed (collected, time-stamped, forwarded) or not. The length of a sequence to be observed determines the size of a window.

The probability of a windowed event sequence, p_w, is a product of each repeated event's probability in the window. Repetition is counted including the most recent event, however, if a most recent event is different from the previous one, then p_w is equal to the probability of only that most recent event. For example, if we have the "*AAAA*" event sequence in the window with probabilities $p_0 = p_A$, $p_1 = p_0\,\alpha_0$, $p_2 = p_1\,\alpha_1$, $p_3 = p_2\,\alpha_2$ respectively, where p_A is the probability of event A to occur and α's are the transition probabilities, then p_w is equal to:

$$p_w = p_0 p_1 p_2 p_3 \tag{2}$$

The transition scheme between events is shown in Figure [1]. The circle on the left represents a most recent event and the circles on the right represent possible events at the destination. The arcs between the circles show transitions both to a same kind of event and to a different event with probabilities α and β respectively, which are also known as *conditional probabilities*.

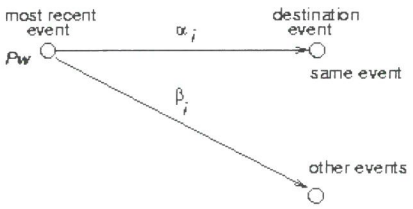

Figure 1. Trace Transition Diagram

If an event of interest occurs during program execution then it is more likely to occur again than those of any other events due to repeated execution of program segments. Based on this, the conditional probabilities can be assigned to the transitions properly. Any unexpected event (such as sudden changes in a sequence) with a very low possibility will produce a big modified entropy value, likewise an expected event sequence (occurrence of similar events) at the destination will produce a smaller value. Table [1] shows example transition probabilities assigned based on repetition count for a window size of three.

Table 1. α And β Values

I	α	β
0	0.5	0.5
1	0.7	0.3
2	0.9	0.1

The algorithm dynamically maintains a window that holds event history. At each iteration, the algorithm identifies both the most recent and the destination event. The transition table is then checked to determine which transitive edge to use (see Equation 3).

$$H = -K \begin{cases} p_w \alpha_i \log(\alpha_i) & \text{if same event occurs,} \\ p_w \beta_i \log(\beta_i) & \text{otherwise.} \end{cases} \quad (3)$$

An Example:

Suppose we have a set of events containing four elements, namely *A,B,C,D*, with probabilities to occur $p_i = 0.25$ each. We use a window size of three and transition probabilities shown in Table [1] to calculate the modified entropy values for two different window positions in an event sequence (see Figure 2).

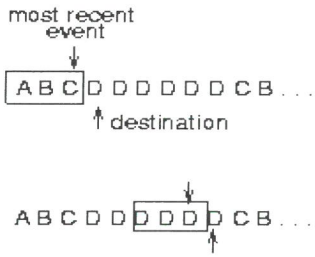

Figure 2. Example Sequence and the Window in Two Different Positions

In the first window position, *repetition_count* = *0*, thus $p_w = 0.25$. The destination is different from the most recent event so β_0 must be used in Equation 3. Then modified entropy, *H*, can be calculated as $H = 0.0866$ nats for $K=1$. Similarly, for the second window position; *repetition_count=2*, $p_w = p_0 p_1 p_2 = 0.25^3\ 0.5^2\ 0.7$ and α_2 must be used in the Equation 3, and *H* is calculated as $H = 0.0006$ nats.

Another Example:

For this example, the window size is set to 4, and transition probabilities shown in Table 2. Now, we have 3 different events in the set: *A,B,C*. We assume the following dynamic occurrences for the events, and the left most is the first event in the stream: ABBBBCCCCCCCCCCCCCCCA

Table 2. Transition Probabilities

I	α	β
0	0.1	0.45
1	0.3	0.35
2	0.6	0.20
3	0.9	005

The results are shown in Table 3 from the entropy generator tool. When the event stream goes into a repetitive pattern, the entropy value drops dramatically.

Table 3. Entropy Values For An Event Stream (window size 4)

Iteration	Curr. Event	Window	Mod. Entropy
0	A	* * * *	2.188759
1	B	A * * *	2.995732
2	B	BA * *	0.094824
3	B	BBA *	0.275846
4	B	BBBA	0.195044
5	C	BBBB	2.328448
6	C	CBBB	0.716873
7	C	CCBB	1.986090
8	C	CCCB	0.390087
9	C	CCCC	0.000000
10	C	CCCC	0.000000
11	C	CCCC	0.000000
12	C	CCCC	0.000000
13	C	CCCC	0.000000
14	C	CCCC	0.000000
15	C	CCCC	0.000000
16	C	CCCC	0.000000
17	C	CCCC	0.000000
18	C	CCCC	0.000000
19	C	CCCC	0.000000
20	C	CCCC	0.000000
21	A	CCCC	2.328448

From the experiments, the modified entropy values drop if the events repeat. In time driven monitoring systems, multiple samples are usually taken from the same segment. Next section discusses how the entropy algorithm can be used for monitoring.

4.USE OF ENTROPY CONCEPT FOR MONITORING

We sketch out an example to show how the modified entropy algorithm can be used to quantify information content. We assume that the monitoring system is event-driven. That means event occurrences will be reported as soon as they occur. Assume a master-slave program, where a master process dispatches the tasks to the slave processes; then every process, including the master, works on its own share, and finally the master collects the successive results (see Figure 3).

A dummy parallel code

```
main(){
  sensor_call(program_st);
  if (dispatcher) {
    for(i = 0; i < proc ; i++) {
      sensor_call(send_i);
      send(message_i);
      }
  } else {
      sensor_call(recv1);
      receive(message1);
  }
  for(i = 0 ; i < 100; i++) {
      sensor_call(do_calc);
      do_calc();
      }
  if (!dispatcher) {
      sensor_call(send_res);
      send(result);
  } else {
  for(i = 0; i < proc ; i++) {
      sensor_call(recv_res);
      receive(result);
      }
  }
sensor_call(program_sp);
}
```

Figure 3. An Instrumented Dummy Program

Lines with `sensor_call()` in Figure 3 are function calls to the instrumentation library with an event name as an argument. The procedure `do_calc()` does not contain any instrumentation. Exit points of structures are not instrumented because at the same level starting point of next structure generates a time stamp for the previous exit.

The next step is to calculate the modified entropy values for each instrumentation point for window size four. The results are shown in the Table 4. The first column in the table shows instrumentation points visited by the master and the second column shows the associated modified entropy values. The rest of the nodes (which correspond to the slaves) behave in a similar fashion.

Table 4. Modified Entropy Values For Node 0 and Node 1,2,3

Event in node0	Mod. Entropy (K=100)	Event in node1,2,3	Mod. Entropy (K=100)
prog_st	2.665951	Prog_st	6.931472
send_0	2.665951	Recv1	6.931472
recv_0	2.665951	do_calc	6.931472
send_1	2.665951	do_calc	0.499345
send_2	2.665951	do_calc	0.024992
send_3	2.665951	do_calc	0.000743
do_calc	2.665951	do_calc	0.000004
do_calc	0.073868	do_calc	0.000004
do_calc	0.001422	do_calc	0.000004
do_calc	0.000016	do_calc	0.000004
do_calc	0.000000	do_calc	0.000004
do_calc	0.000000	do_calc	0.000004
...	...	...	...
do_calc	0.000000	do_calc	0.000004
do_calc	0.000000	send_res	0.921034
do_calc	0.000000	Prog_sp	6.931472
do_calc	0.000000		
do_calc	0.000000		
send_res	0.354244		
recv_res	2.665951		
recv_res	2.665951		
recv_res	2.665951		
recv_res	2.665951		
prog_sp	2.665951		

The master produces different trace data from the slaves since its control flow is different. After first few instrumentation points, the information content of the instrumentation data drops dramatically for the master and the slaves. This is because the routine executes repeatedly reporting the same event.

5.PHASE DETECTION RESULTS

Phase is a period of time when a program exhibits similar behavior. Programs can go through several different phases during execution, such as initialization, computation, communication etc.. These phases can be detected by observing CPU usage, or periods of low/high message frequency [6].

To test our algorithm with real programs, we built a tool to process trace data collected with an event-driven monitoring system. The results show that the algorithm can be used to observe execution progress and to detect phases based on the performance data collected.

We conducted several experiments with an existing monitor (AIMS [9]). We present here one of them, which is *parallel matrix transpose* program. The program is provided with the AIMS performance monitor package, and "transposes" a matrix on the nodes of a hypercube. Initially the matrix is distributed by rows among a certain number of processor; at the end of the run, the matrix is distributed by columns. The following results are obtained after processing the trace output of this program which runs on 16 hosts.

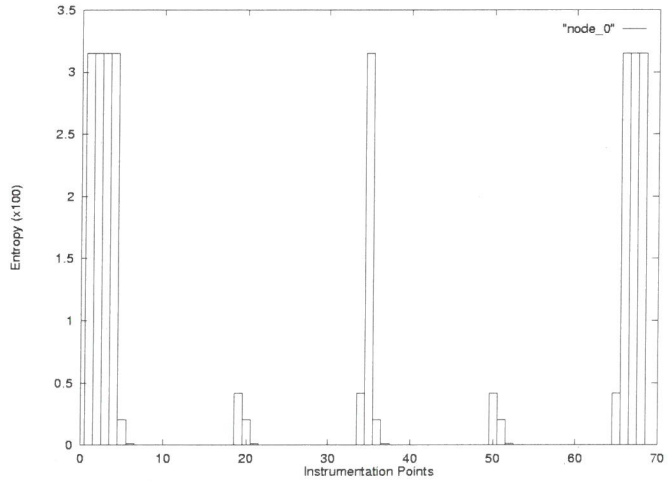

Figure 4. Master Processor: Modified Entropy Values For Matrix Transpose Program

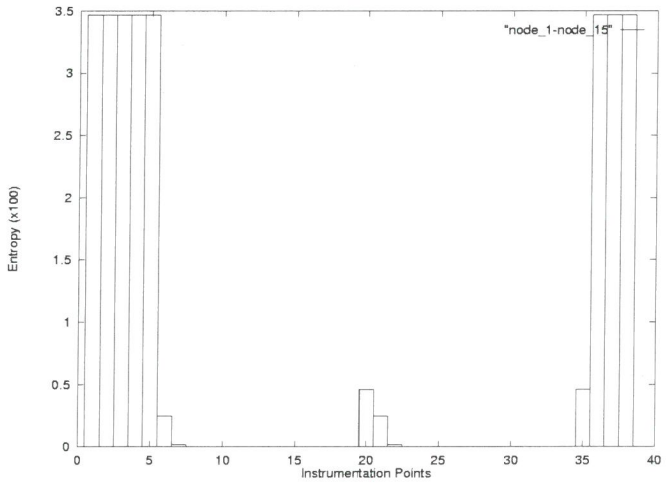

Figure 5. Slave Processors 1-15: Modified Entropy Values For Matrix Transpose, They All Behave Same As Expected

Figure [4] shows the master; it distributes the data to the slave nodes in row basis, and then column basis. Figure [5] shows the slaves; the slaves (1 to 15) receive the data from the master. The slaves all have similar characteristics, which are different from the master as expected. The modified entropy value goes down when a program repeats similar activities, and any changes make a peak in the graph. Both figures show execution progress and changes in behavior.

Paradyn [1] uses timers and counters to detect the phases of an application. A timer determines the length of an observation interval, and counters determine which event is dominant in that interval. At the end of each interval counters are checked, and the

maximum is found. This maximum counter represents the phase. We do not use timers and counters in our approach because our algorithm automatically determines a dominant event during execution.

6.CONCLUSION

This paper presented the modified entropy algorithm and have described its use with examples for phase detection and program progress observation. Progress observation works both with event-driven and data-driven systems. Time-driven systems collect runtime data in certain intervals, and samples are obtained from the execution. If samples come repeatedly from the same segment, we must know that similar event data comes from the sampler not because of the same segment executes repeatedly. It is not a repetitive pattern of application but sampled data. To obtain similar behavior as in event-driven data collection, samples must be removed between entry and exit of program constructs. Our entropy based instrumentation algorithm eliminates redundant samples on the fly.

7.REFERENCES

Hollingsworth, J.K., Miller B.P. ve Callaghan, M.D., "The Paradyn Parallel Performance Tools and PVM", *SIAM Press*, 1994.

Jones, D.S. "Elementary Information Theory", *Claredon Press*, Oxford, 1979.

Lim, S.J., "Two Dimensional Signal and Image Processing", *Prentice-Hall Inc.*, 1990

Özmen, A., "A Minimal Overhead Instrumentation System", *In Proceedings of the Fifteenth International Symposium on Computer and Information Sciences (ISCIS XV)*, İstanbul, Turkey, October 2000.

Özmen, A., "Paralel Gözlemleme (monitör) Sistem Mimarisi", *In ELECO 2000 – Elektrik-Elektronik-Bilgisayar Mühendisliği Sempozyumu*, Bursa, Turkey, November 2000.

Özmen, A. and Lumpp, J., "Dynamic Configuration of Software Instrumentation in Parallel Systems". *Proceeding of The Twelfth International Symposium on Computer and Information Sciences (ISCIS XII),* Antalya, Ekim 1997.

Schwartz, L.S., "The Principals of Coding, Filtering and Information Theory", *Cleaver-Hume Press*, 1963.

Shannon, C.E., "The Mathematical Theory of Communication", The University of Illinois Press, Urbana, 1963.

Yan, J. "Performance Tuning with AIMS-An Automated Instrumentation and Monitoring System for Multicomputers", *In Proceedings of the 27th HICS*, Wailea, Hawaii, January 1994.