An Entropy-Based Algorithm for Time-Driven Software Instrumentation in Parallel Systems

Ahmet Özmen

Dumlupınar University
Dept. of Electrical and Electronics Engineering
Tavşanlı Yolu 12 Km., Kütahya 43100 Turkey
ozmen@dumlupinar.edu.tr

Abstract

While monitoring, instrumented long running parallel applications generate huge amount of instrumentation data. Processing and storing this data incurs overhead, and perturbs the execution. Techniques that eliminates unnecessary instrumentation data and lower the intrusion without loosing any performance information is valuable to tool developers. This paper presents a new algorithm for software instrumentation to measure the amount of information content of instrumentation data to be collected. The algorithm is based on entropy concept introduced in information theory, and it makes selective data collection for a time-driven software monitoring system possible.

1. Introduction

Performance evaluation and debugging of parallel systems 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 a widely used alternative. However, it is intrusive because instrumentation introduces overhead that perturbs the behavior of the original programs. Static configuration of instrumentation, although simple, is not in general efficient in terms of information gathered to overhead introduced. The problem is more severe for long running applications, servers and operating systems.

Last decade many parallel monitoring systems have been developed using different instrumentation techniques; including on-the-fly instrumentation [10], library instrumentation [4] and source-to-source instrumentation [16, 1]. An important work that relates to our work is Paradyn [10], which has been developed for monitoring long running parallel applications. It collects data using dynamic instrumentation technique and adjustable sampling rate so as to reduce the performance data amount. The time- driven control changes the sampling rate using a hypothesis set of performance problems. If incoming data shows a problematic pattern then sampling rate is increased to get more data and a better vision about the problem.

XPVM and AIMS are other parallel monitoring systems that employ event driven data collection mechanisms for PVM and MPI applications [4, 16]. Interested parts of an application are instrumented before execution, e.g. sends, receives or function entries and exits. When an instrumented application executes, it generates performance data as a side effect. Provided on-line or off-line tools consumes this data and generate graphical outputs about the state and the performance of the application.

Our research focus on low intrusion software instrumentation systems [13, 12]. This paper presents an algorithm for time driven software instrumentation systems 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. The algorithm has been developed using the *entropy* concept introduced in the information theory to evaluate and quantify the information content of the performance data.

The rest of the paper is organized as follows. Section 2 presents background information about PVM, software instrumentation, informativeness of performance data and entropy definition. Section 3 describes

1-4244-0054-6/06/$20.00

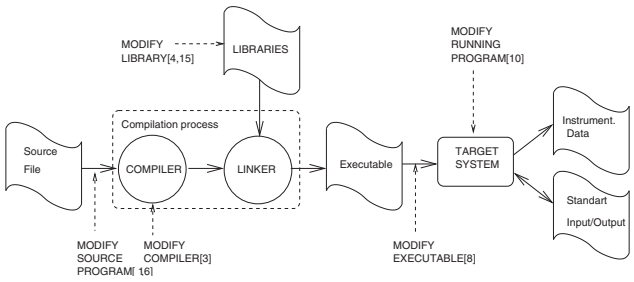

Figure 1. Software instrumentation on different stages of compilation.

the algorithm, and Section 4 shows an example how the algorithm can be used to reduce redundant performance data.

2. Background

2.1. PVM: Parallel Virtual Machine

PVM is a software library package which provides a virtual parallel environment for distributed message passing programs using existing networked computers when parallelism is advantageous [15]. PVM was developed by Oak Ridge National Laboratory in collaboration with several universities, principal among them being the University of Tennessee at Knoxville and Emory University. We have used PVM to test our algorithm.

2.2. Software Instrumentation

Performance information can only be obtained by *monitoring* program execution via *instrumentation*. In a software monitoring system, instances of software instrumentation are referred to as *sensors*. Software instrumentation may be inserted before, during or after compilation as shown in Figure 1 [1, 3, 5, 8, 10, 11, 15, 16].

Instrumentation data is logged by a sensor if an event of interest occurs. A sensor trigger policy may either be *event-driven* or *time-driven* (these correspond to *tracing* and *sampling* respectively) [6]. In an event-driven policy, instrumentation data is collected synchronously with the occurrence of an event, whereas in a time-driven policy the data is collected asynchronously. Instrumentation data is recorded as an *event record* that includes information related to that particular event; such as a time-stamp that shows when the event occurred, a location-stamp that shows where the event occurred, and additional data about the state of the system. Software instrumentation can incur substantial overhead and redundant instrumentation can be expensive.

2.3. Informativeness of a Performance Data

To reduce the intrusion, performance data elimination efforts take place in different phases; during instrumentation or execution. Many instrumentation tools allow user to select only some strategic points in the application to reduce the performance data [4, 16]. Others employ dynamic algorithms that changes instrumentation to reduce the data while the application is in execution [10].

We have focused on developing an algorithm based on informativeness of performance data that will change the data collection (instrumentation) dynamically. However, we needed a measurable quality of that property of a performance data, which is not known yet. Almost all events that happen during an execution are known priori, however, the order in which they occur is not known. This is because control-flow of a program can not be inferred from a static analysis. Since we do not know the program flow, it may still be generating redundant data during execution (for example; repeated program segment executions). From the traces, it was clear that instances of performance data do not contain equal amounts of information; many of them are redundant. So, a redundant data is whether (1) a data contain no information, which means causes of an event are obvious and known by interested observer, or (2) the information, of which data may contain, can be extracted using previous data.

As a result, we started searching possible ways of quantifying the informativeness of a performance data, and ended up with the entropy definition in the information theory.

2.4. The Entropy Definition

The concept of entropy is introduced as a measure of uncertainty of a random variable in the information theory [14]. The entropy, H, is defined in Equation 1, where n is the number of possible events whose possibilities of occurrence p_i:

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

Since $\sum_{i=1}^{n} p_i = 1$, it can be shown that $0 \leq H \leq log(n)$. 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 [7, 9]. 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 $p_1 = p_2 = \frac{1}{2}$. The entropy is then $H = 1$ bit. Receiving the message clearly adds new information.

3. The Algorithm

The entropy concept can be used to evaluate information content of instrumentation data. A link between parallel monitoring and the information theory is set up with the following two statements:

- 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.

We have slightly modified the original entropy definition to get it work in monitoring properly. 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.

Transition probabilities: The transition scheme between events is shown in Figure 2. 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*.

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 locality of reference. The conditional probabilities are assigned to the transitions empirically based on the locality of reference. Any unexpected event (such as sudden changes in a sequence) with a very low probability will produce a big entropy value, likewise an expected event sequence (occurrence of similar events) at the destination will produce a smaller value. Table 1 shows example empiric transition probabilities assigned based on repetition count for a window size of three.

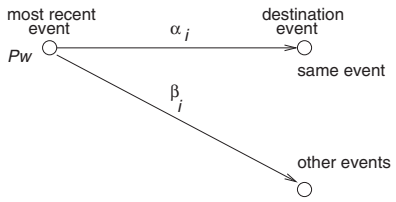

Figure 2. Trace transition diagram that shows the transition probabilities between the most recent event and destination events.

i	α	β
0	0.5	0.5
1	0.7	0.3
2	0.9	0.1

Table 1. Empiric α and β values based on locality of reference.

The window: The window holds events happened in the past, and produces a probability of this sequence p_w based on Markov model [2]. 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 αs are the transition probabilities, then p_w is equal to:

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

The probabilities of all the events are calculated before execution. These events are extracted with an analysis tool, and then each event is assigned with equal amount of probability to occur which makes $p_{event} = (1/total\ event\ count)$ assuming that all these events will happen during execution.

The window size defines "how deep we want to look at into the history of execution", and larger window sizes create more variations in entropy values. For example, if the window size is one, then the output becomes bi-level since $p_w = p_0 = p_{event}$ is the same for

all the time for all the events. Hence, the output of the algorithm for window size one depends only on the transition probability.

However, for larger window sizes, the p_w changes based on the history of execution. Then, the output of the algorithm depends on both the history with p_w and the current state of the transition. Hence, the variation of the output increases with the window size which gives a better measure about informativeness. However, the larger the window size, the more overhead incurs due to calculations. For example, a window size four produced acceptable results during the experiments.

Quantification: 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 4). Then, the following formulas are used to quantify how significance of an event.

$$S = K\ p_w\ h \tag{3}$$

$$h = -\begin{cases} \alpha_i\ log(\alpha_i) & \textit{if same event occurs,} \\ \beta_i\ log(\beta_i) & \textit{otherwise.} \end{cases} \tag{4}$$

where K is an arbitrary number for scaling, and p_w windowed event sequence probability, h is an information theoretic measurement of how abnormal it is the next event to be different from the last event or remain the same. The h is calculated with the Formula 4. A flow chart of the algorithm is shown in the Figure 3.

An Example:

Let us assume that a set contains four events $\{A, B, C, D\}$ with probabilities to occur $p_i = 0.25$ each, and these events occur in a sequence shown in the Figure 4. The modified entropy values can be calculated using a window (size of three) and transition probabilities (shown in Table 1), for two different window 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 4. Then, the informativeness of the event, S, can be calculated as $S = 0.0866$ 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 S is calculated as $S = 0.0006$ for $K = 1$. These meaningful values show that information amount of an event sequence can be extracted.

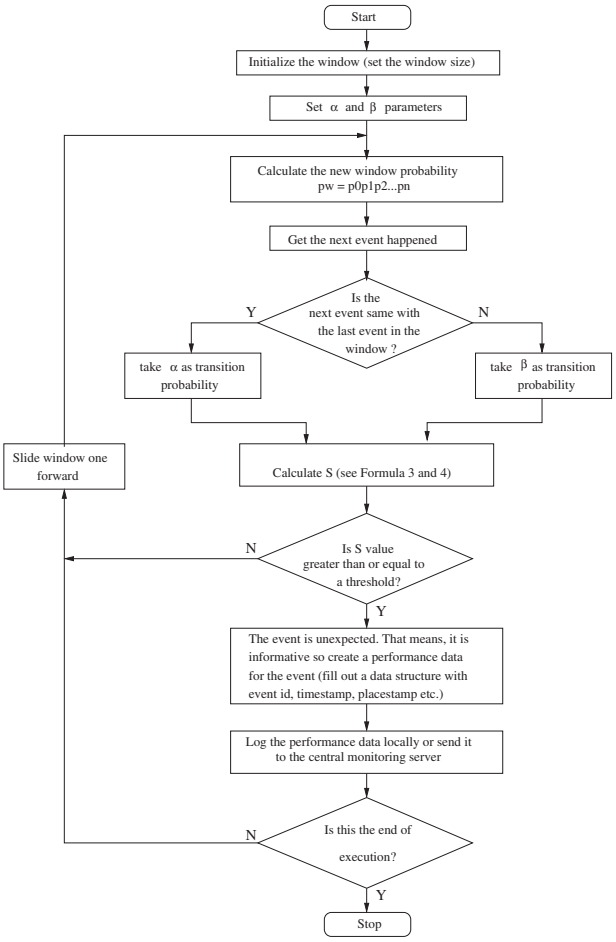

Figure 3. A flow chart of the entropy algorithm for monitoring.

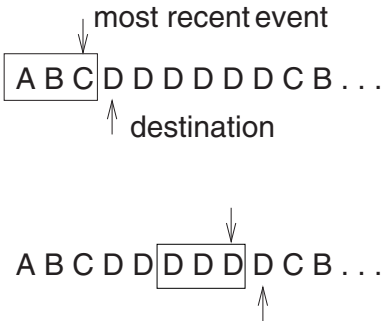

Figure 4. Example sequence and the window in two different positions.

4. Using the Algorithm with Synthetic Data

Time-driven instrumentation mechanisms are used in profiling tools to collect the samples from multiple processes. The data, then integrated in a central place to show total behavior. The sampling interval in nodes is usually set short enough to catch every important event, and usually unwanted multiple samples are taken from structures. However, the first and the last data from a structure are the most informative samples and the rest reports the same event repeatedly. The entropy-based algorithm can easily detect the first and the last points.

Figure 5 shows a simple parallel program to test our algorithm, in which a parent process spawns the slave processes and sends initial parameters with `pvm_spawn()` library call. Each process (master and slaves) first receives its own data with `pvm_recv()` call, does its own calculations in `doCalc()`, and finally sends the results back to the parent with `pvm_send()` call. In this example, there are eight different functions executed; four of them are library calls: `pvm_spawn()`, `pvm_send()`, `pvm_recv()` and `pvm_exit()`. Entry and exit points of these functions becomes interested events during execution. They all are extracted during static analysis using AIMS. Since these events are known to happen during execution, an equal amount of probability to occur is given to each of them.

Since we do not have an available time driven performance tool, we instrumented this application with AIMS, run it under PVM, and recorded the trace data. Then, we created a sampling data train to test our algorithm from the recorded trace data. We applied this created sampling data to our algorithm and obtained the entropy values shown in Figure 6. In the middle of the figure, modified entropy values drop because samples come from repeating function foo(), which is not a long running segment.

Figure 7 shows a hierarchical view of the execution segments of above program. The vertical lines in the figure show samples; some of which are solid meaning a beginning of a new function, and the remaining are dashed just to distinguish from the others. A conventional profiler tool like gprof, collects whole samples for a full trace, although samples shown with the solid black lines were sufficient for profiling.

When compared, entropy values in Figure 6 and the solid black lines in the Figure 7 show some similarities. The samples pointed by the solid black lines in the Figure 7 get the maximum entropy values, and these values can be used to control a data collection mechanism.

```
main(){
  if(parent)
    for(i = 0; i < proc ; i++)
       pvm_spawn(...);
  else
    pvm_recv(...);
doCalc();
pvm_recv(...);
pvm_exit();
}
doCalc() {
  init();
  for(i = 0; i < 4; i++) {
      foo();
  }
  for(i = 0; i < proc; i++) {
     if(me)
       pvm_send(...);
     else
       pvm_recv(...);
  }
}
foo() { /* procedure body */   }
```

Figure 5. A simple PVM program to test entropy algorithm for performance data elimination

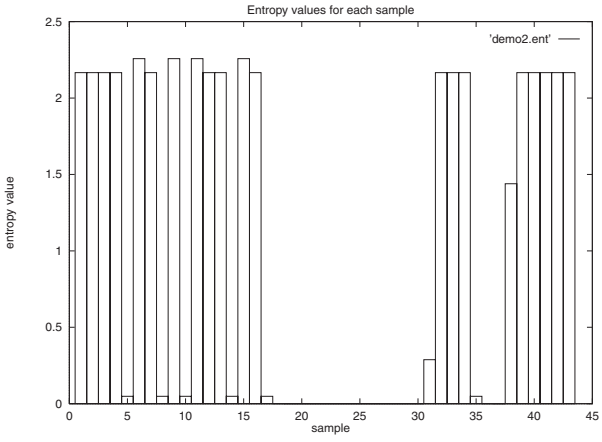

Figure 6. Entropy values of the execution in a node. Entropy values drop dramatically when the samples starts coming from the same segment

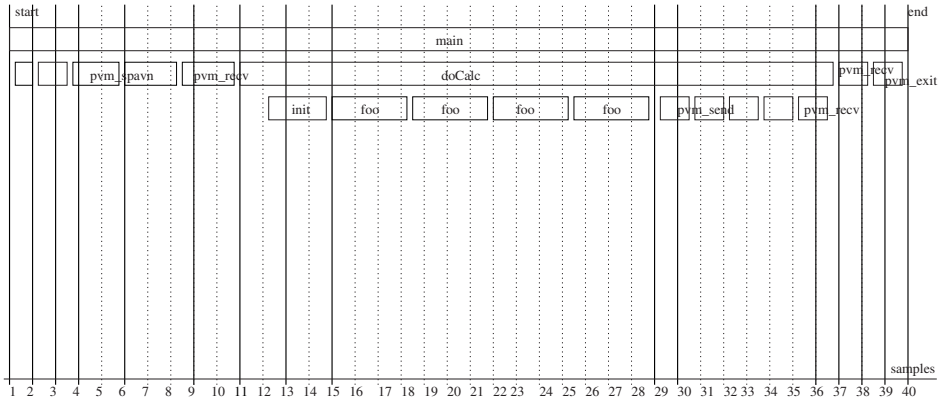

Figure 7. Hierarchical view of execution segments: Solid lines show the most informative samples.

5 Conclusion

Time-driven instrumentation systems are preferred for monitoring long running applications since it is possible to reduce the performance data amount by changing the sampling rate. In conventional time-driven instrumentation systems the sampling rate is usually fixed. If the sampling interval is set short, then a lot of performance data is generated. Or, if the sampling interval is set larger to reduce the data amount when the program executes longer, then some important events can be overlooked.

This paper presented an algorithm to solve this dilemma by adjusting sampling interval of a time-driven instrumentation system based on the informativeness of a performance data. A quantified value of the informativeness gives another dimension to the performance data which becomes an important parameter to control its collection.

References

[1] V. Adve, J. Crummey, M. Anderson, K. Kennedy, J.-C. Wang, and D. Reed. An integrated compilation and performance analysis environment for parallel programs. In *Proceedings of Supercomputing '95*.

[2] G. Bolch, S. Greiner, H. de Meer, and K. Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. John Wiley and Sons, 1998.

[3] Convex Computer Corporation. *Convex CXpa Refrence*, second edition edition, Dec. 1994.

[4] A. Geist, , J. Kohl, and P. Papadopoulos. Visualization, debugging, and performance in pvm. In *Proceedings of Visualization and Debugging Workshop*, Oct. 94.

[5] L. Harris and B. Miller. Practical analysis of stripped binary code. In *Workshop on Binary Instrumentation and Applications (WBIA-05)*, Sept. 2005.

[6] R. Hoffmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. Distributed performance methods, tools, and applications. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):585–598, June 1994.

[7] D. Jones. *Elementary Information Theory*. Oxford University Press, 1979.

[8] J. Larus. Abstract execution: A technique for efficiently tracing programs. *Software: Practice and Experience*, 20(12):1241–1258, Dec. 1990.

[9] J. Lim. *Two Dimensional Signal and Image Processing*. Prentice-Hall Signal Processing Series, 1990.

[10] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, Nov. 1995.

[11] B. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski. Ips2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):206–217, Apr. 1990.

[12] A. Ozmen. A minimal overhead instrumentation system. In *Proceedings of The Fifteenth International Symposium on Computer and Information Sciences, (ISCIS XV)*, pages 102–110, Nov. 2000.

[13] A. Ozmen and J. Lumpp. Dynamic configuration of software instrumentation in parallel systems. In *Proceedings of The Twelfth International Symposium on Computer and Information Sciences, (ISCIS XII)*, Nov. 1997.

[14] C. Shannon, W. Weaver, R. Blahut, and B. Hajek. *The Mathematical Theory of Communication*. The University of Illinois Press - Urbana, 1999.

[15] V. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[16] J. Yan. Performance tuning with AIMS — an Automated Instrumentation and Monitoring System for multicomputers. In *Proceedings of the 27 th Hawaii International Conference on System Sciences*, pages 625–633, Jan. 1994.