Cluster Monitoring Platform Based On Self Adaptable Probes

Céline Boutros Saab, Xavier Bonnaire
Laboratoire d'Informatique de Paris 6, Université Pierre et Marie Curie Paris 6
4, place Jussieu, 75252 Paris Cedex 5 – France
{Celine.Boutros@lip6.fr, Xavier.Bonnaire@lip6.fr}

Abstract—

Distributed systems based on cluster of workstations are now largely adopted in industries and universities, but they are more complex to manage. Some value added tools offer new services to the operating system to efficiently exploit the cluster power. Usually, such a tool needs to monitor a set of resources with a variable granularity, and most existing tools use a fixed granularity.

In this paper, we propose to use a generic monitoring platform with a variable granularity based on a self adaptable probes, which will offer an automatic granularity tuning to avoid a permanent fine grain monitoring, and a reliable mechanism to handle a resource emergency state. Self adaptable monitoring also allows to reduce the overhead produced by the observation system.

Keywords— **Cluster management, monitoring, dynamic granularity, self adaptable probes.**

I. Introduction

Distributed systems are now largely adopted in industries and universities. This is mainly due to the low cost of personal computers and the possibility to take advantage of all the available resources through a fast local area network [FAU 99]. We consider a cluster as a set of interconnected computers, and a distributed application as a set of cooperating processes running on one or more processors [TEL 94].

Distributed applications designed to exploit this power are more complex to manage than centralized ones. To deal with this complexity, some value added tools offer new services, usually not included in the operating system [AHR 99] [BOR 95] (Load Balancing [ZHO 93] [FOL 94], Fault Tolerance [FOL 94], Debug and Replay [NET 92] [RON 99] [RON 97], Performance Measurement [MIL 95] [NOE 94], etc...). These tools require monitoring information, and often come with their own observation technique, which implies building a new observation system when designing a new tool.

Every observation system involves a monitoring granularity and a set of monitored resources [MAN 93]. The granularity is either a coarse grain or a fine grain one, but most existing tools use a fixed granularity, which cannot be changed during the monitoring process [SRI 94]. A coarse grain monitoring, realized with a small frequency, produces less intrusion to the applications and the underlying operating system. However, a Δt delay exists between the occurrence of the resource variation and the moment it is detected, which is due to the monitoring frequency. The Δt delay can be reduced by increasing the frequency to reach a fine grain monitoring that produces more reliable information, but it often leads to an important network load and is extremely CPU time consuming.

A usual way to decrease the overhead generated by a fine grain monitoring is to reduce the number of observed resources. The information accuracy is therefore the same for this subset of resources, but it is worse for the entire cluster. On the other hand, only a subset of all the resources require permanent monitoring, while others need to be observed during critical phases. This resources subset is usually specified at the beginning of the observation [SRI 94] [EUS 95] and cannot be modified later, even if one or more resources are no more pertinent for the current observation. In the same way, new pertinent resources cannot be added during the monitoring process.

Another side effect when choosing a coarse granularity is to potentially miss the occurrence of an emergency state of a resource, and let it reach a fatal value where the associated station or application crashes. In this case, increasing the monitoring frequency to a fine granularity will reduce the probability of missing the emergency state, but won't ensure detection on time.

In this paper, we propose to use self adaptable probes in a generic monitoring platform. Using a self adaptable probe is an efficient technique to reduce monitoring overhead while offering fine grain observation, which is only performed when necessary. Moreover, self adaptable probes allow to detect an emergency state for a given resource, and give a value added tool the opportunity to handle this state.

Section II presents the requirements for an adaptable monitoring platform. Section III gives the general architecture of PHOENIX, our generic monitoring platform. Section IV defines monitoring techniques using

resource can be requested by a tool registered to this resource tracing and specified using logical expressions.

B. The Network Message Server

The NMS (Network Message System) is a multicast communication system which offers a simple way to send data with message passing. An NMS client can specify the set of messages it wants to receive through a registration mechanism based on message signatures. A registration can be dynamically created, destroyed, activated or deactivated. The NMS hides the distributed environment, the underlying architecture and the operating system, and is location independent, where a transmitter doesn't have to care either about the location of the receivers, nor their numbers.

Communications between the different components of PHOENIX are made through the NMS to respect the location independent requirement [BON 95], and a tool can thus be located anywhere on the distributed system.

C. Library

The PHOENIX library provides a set of monitoring primitives to replace the application's system calls and for internal management of the collected traces.

Another set of primitives allows the management of monitoring conditions (specification, modification and logical expression). It also provides the way to retrieve traces of the system and the applications. There are three classes of primitives, *connection, subscription* and *granularity control*. The first class is used to get connected to and disconnected from the platform. Once connected, a tool has a set of primitives to subscribe to a resource trace, to activate, to deactivate and to cancel a subscription. When subscribed, a tool can dynamically modify any resource granularity.

The PHOENIX library has to be linked to both, tools and monitored applications. The only modification at the application level is done to its Makefile, which has to include this new library.

D. System Probes

The System Monitoring Agent (SMA) is a kernel module including a set of system probes which supplies monitoring information and receives requests to set or modify its monitoring requirements. It sends the collected system information to the tools according to their requests.

An instance of the SMA is inserted on each workstation kernel. It waits for a request to start monitoring. Upon reception of a request, it begins monitoring and delivering to the tool at a given frequency. When no more tracing is requested, the SMA stops the observation and the transmissions. An SMA is in charge of local resource monitoring like the load of the workstation and the total allocated memory. A monitoring condition is associated with each resource, which defines its monitoring frequency.

E. Application probes

A probe is a piece of code automatically inserted by the preprocessor into the source code of the application during the compilation step. The PHOENIX library is then added to the current application's libraries during the linking step.

A probe is always active, it can be dynamically enabled or disabled. A disabled probe executes the specified calls without executing the monitoring primitives. Each enabled probe is associated with a monitoring condition specified by a logical expression.

F. Logical Expressions

Logical expressions are stored in a file and include operating system and application monitoring conditions. Probes initialization is done at boot time for the operating system probes, and in the main() function for the applications probes.

There are three types of application probes; COMM, MEM and I/O probes (for communication, memory, and input/output respectively). Each probe has three parameters that can be used in a logical expression. The first one is the tracing frequency (*IO_FREQ, COMM_FREQ,* and *MEM_FREQ),* which defines the elapsed time between two consecutive tracing. The second parameter, which expresses the number of a resource use, is *IO_CALL_NBR* for the I/O probe, *MEM_CALL_NBR* and *COMM_CALL_NBR* for respectively the MEM and the COMM probes. The third parameter corresponds to the total volume of a resource use, thus *COMM_VOLUME, MEM_VOLUME*, and I/O *IO_VOLUME.*

In addition to the applications probes, the operating system probes include a load probe (LOAD), which gives the workstation load. The LOAD probe has only two parameters, the probe frequency (LOAD_FREQ) and the probe value (LOAD_VALUE).

Monitoring granularity of a given probe is defined by a logical expression. A logical expression includes simple expressions and logical operators (AND, OR , NOT) [BOU 00]. A simple expression is composed of a probe parameter, unary (DELTA) and binary operators (=, <, >, <=, >=, <>) and a probe value. DELTA(term) represents the magnitude of change on the value strictly generated since the last trace delivery. A typical example of a logical expression for a system memory probe is:

```
probe system_memory {
        MEM_VOLUME >= 32M
        OR
        DELTA ( MEM_VOLUME ) > 500.000 ;
        }
```

This expression sets the monitoring condition as follows: traces are delivered when the memory allocated by the operating system exceeds 32 Mbytes, **OR** when the volume of the allocated memory since the last trace delivery has exceeded 500.000 bytes.

IV. Monitoring Thresholds

Usually, a value added tool monitors a subset of one or more resources. For each resource, it can have multiple critical phases. Two thresholds are associated with each phase, a warning threshold and a hard threshold. If the resource value goes beyond the warning threshold, the tool enters into the critical phase. On the other hand, the probe enters into the emergency area, and notifies the tool when the hard threshold is exceeded. At this phase, the tool should perform an action to handle this event.

Such an action depends on the current cluster state (local operating systems, a set of applications, a global state of a resource). Getting the current cluster state usually implies to get more information from available resources. This information can be retrieved either by increasing the accuracy of some already monitored resources, or by adding new resources to the monitored subset.

Increasing the observation granularity for a given resource is done by the tool when it enters into the critical phase.

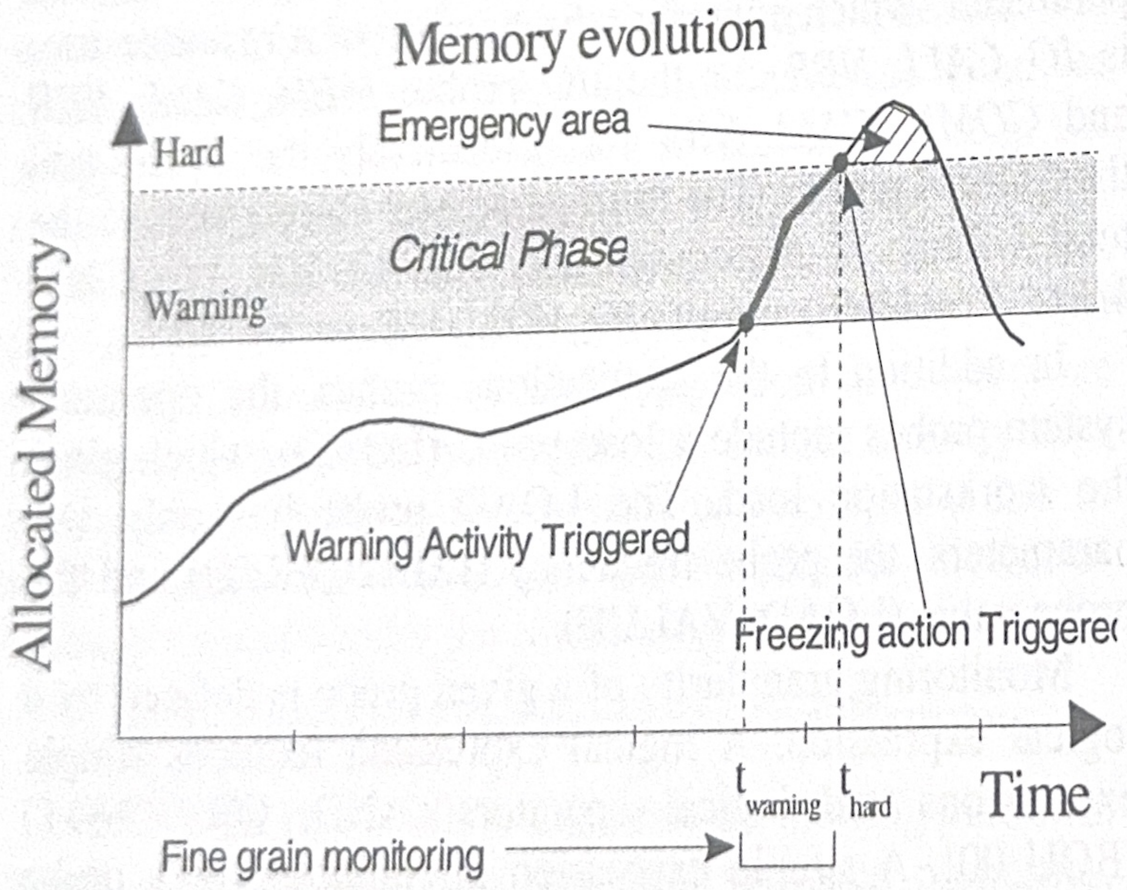

Figure 2: Variable granularity tuning

Figure 2 shows the evolution of the allocated memory by the operating system on a given workstation. In order to detect the point where the memory curve crosses the warning threshold, the following logical expression is used.

```
probe system_memory {
    MEM_VOLUME >= warning_threshold;
    }
```

At this point, the warning monitoring granularity is triggered and the tool changes the probe granularity to a finest one. A finest granularity allows the tool to have a better accuracy of the memory evolution in order to detect as soon as possible when the hard threshold has been reached. At the same time, the tool can request to monitor additional resources with a different granularity. Figure 3 shows the workstation load monitoring initiated by the tool during the critical phase. In this example, the tool requests a load monitoring when the warning activity is triggered and until the hard activity is reached.

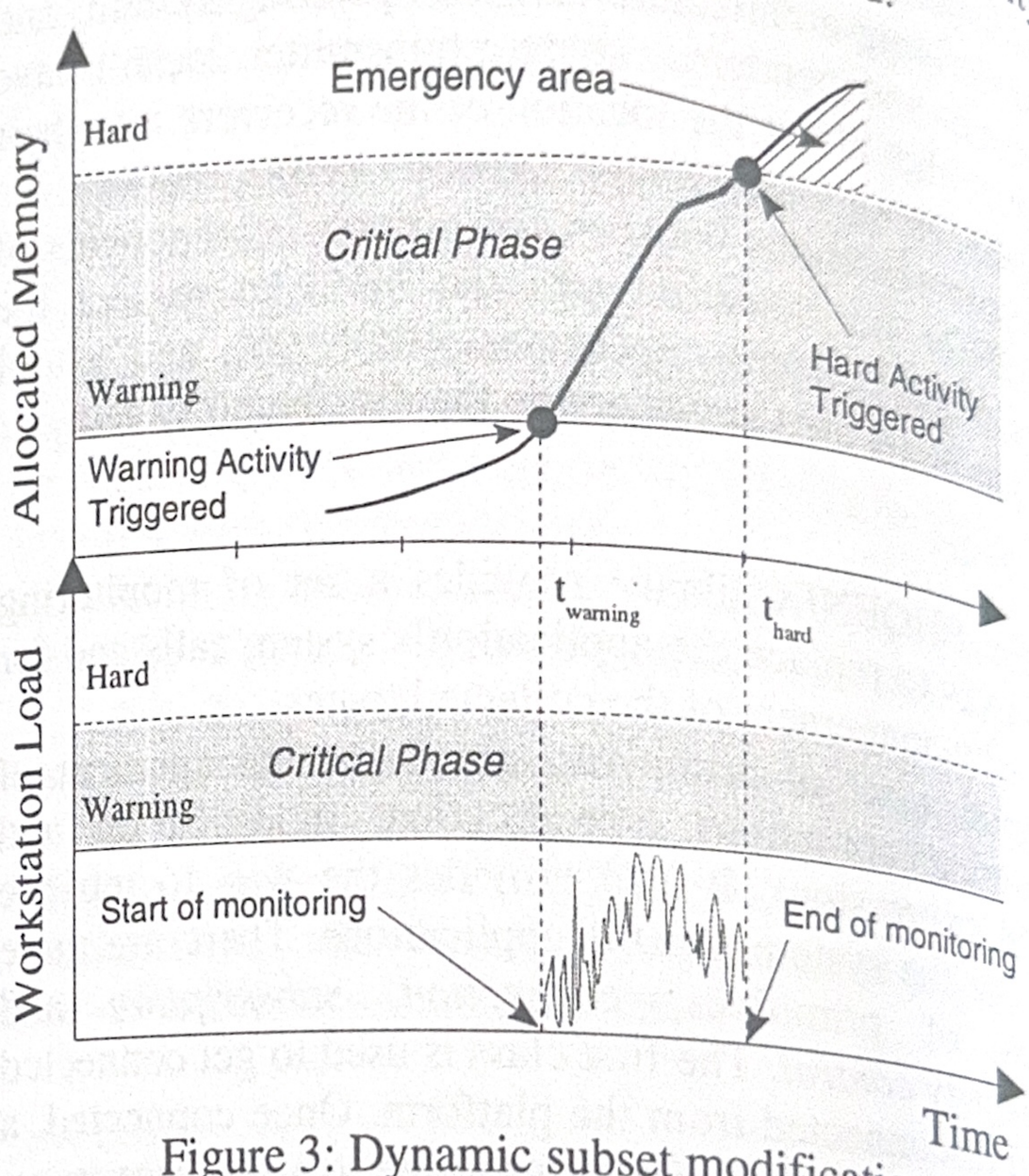

Figure 3: Dynamic subset modifications

The monitoring granularity tuning minimizes the fine grain observation interval to the strictly needed period. The dynamic modification of the monitored resources subset according to events occurrences, prevents the tool from continuously monitoring resources that are not permanently useful. Only the resources permanently useful are continuously monitored with a coarse grain. At the same time, a yo–yo behavior of a resource value (the value quickly moves up and down around the warning threshold), will significantly increase the global overhead. This is due to the induced network load (due to the granularity modification requests) as well as the CPU time needed by the probe to handle the multiple switches between fine grain and coarse grain monitoring.

V. Self Adaptable Probes

The key idea of a self adaptable probe is to handle a set of granularity controls usually performed at the tool level. A granularity control is a request issued by a tool to modify the monitoring granularity of a resource. This approach decreases the communication generated by the tool in order to control the granularity. An error tolerance is defined to handle the side effect of a yo–yo behavior

(See Section IV form more details). Moreover, a self adaptable probe gives the tool the opportunity to manage an emergency state, which occurs when a resource value exceeds the hard threshold. In this case, the resource has a high probability to reach a fatal value (specified by the tool), which will lead to an application or operating system crash (the allocated memory exceeds the amount of available virtual memory, no more space left on the file system, ...).

A self adaptable probe automatically adjusts its monitoring granularity according to the warning and hard thresholds specified by a tool. It is composed of a set of virtual probes and a freezing action to handle an emergency state. A probe can have any number of virtual probes, which can belong to several tools, but a given virtual probe belongs only to one tool. Each virtual probe has the following monitoring condition:

- Default monitoring granularity (Dmg), defined through a logical expression
- A couple of thresholds (warning threshold (Tw) and hard threshold (Th)) and a warning monitoring granularity (Wmg) defined through a logical expression,
- An acceptable error tolerance (Et) associated with the warning threshold,
- A state (READY or NOT READY),
- A freezing action (Enabled or Disabled).

Each virtual probe has its own monitoring granularity, which is independent of the monitoring granularity of the other virtual probes. Therefore multiple virtual probes can have the same critical phase (i.e. the same warning and hard thresholds), but different monitoring granularities, and different error tolerance. A virtual probe is initially at the default granularity, and is in the "Not Ready" state. As soon as its monitoring granularity (i.e. logical expression) is satisfied, it goes to the "Ready" state. Thereafter, the probe also goes to the "Ready" state, which triggers the delivery of a monitoring information and then exits. Thus, only one monitoring information is produced and delivered to the tools, even when several ones are requesting it. This reduces the communication overhead, by reducing the number of delivered information. At the same time, all the virtual probes return to the "Not Ready" state. Figure 4 presents the structure of a self adaptable probe.

The different Virtual Probes (VP) of a probe are processed sequentially, and it suffices that one VP switches to the READY state to stop the current test in order to deliver the appropriate information. A virtual probe automatically switches to the warning monitoring granularity (Wmg), when its value ((CPv), Current Probe value) reaches the warning threshold (i.e. CPv > Tw). It returns to the Dmg (Default monitoring granularity), either when its CPv exceeds the hard threshold (CPv > tw), or when it goes below the warning threshold plus the error tolerance (CPv < Tw – Et). The error tolerance is used to prevent the virtual probe from permanently switching between the default and the warning granularity when the probe has a yo–yo behavior.

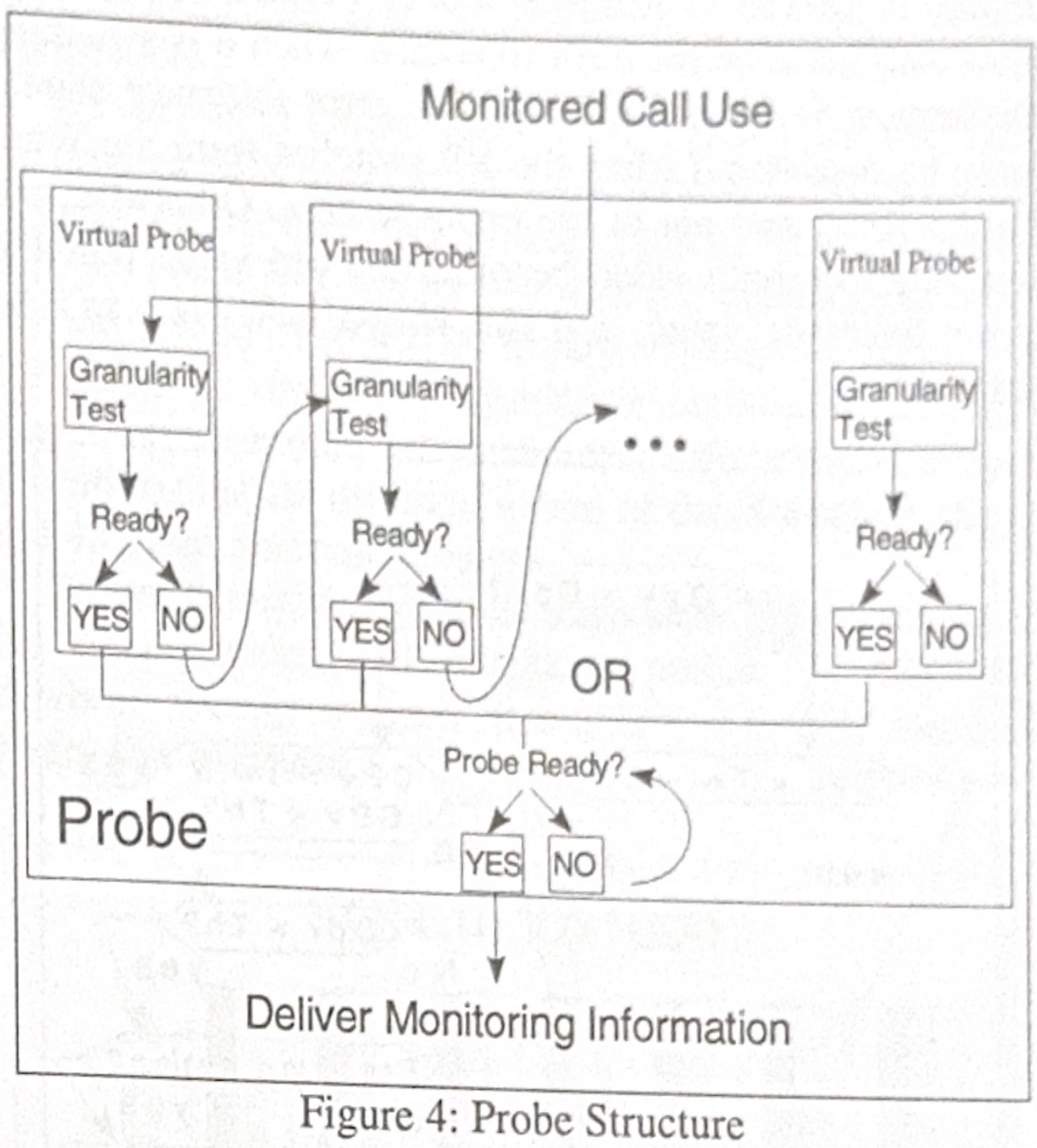

Figure 4: Probe Structure

As long as the CPv varies between the hard threshold and the warning threshold plus the error tolerance, the granularity is not modified. The choice of the tolerance value can significantly influence the virtual probe behavior. If the tolerance value is too small, it won't be able to handle an important yo–yo effect. On the other hand, if the tolerance value is too large, the virtual probe could stay too much time in the critical phase, and could potentially never return to the default granularity. In this case, the monitoring overhead will considerably increase (due to a permanent fine grain monitoring), and a self adaptable probe becomes inefficient.

When a probe enters into the emergency state (i.e. the hard threshold is reached (CPv > Th)), it triggers the freezing action if it is set (Enabled). This action freezes the current application process, which gives the tool the possibility of handling the emergency state. A typical example of an emergency state is when an application process requests more memory than the available virtual memory on the workstation. The freezing action stops the process and notifies a load balancing tool. This gives a chance, just before that the application crashes, to migrate the process to another workstation where more memory is available.

Figure 5 gives the algorithm of the granularity test, which is performed by a virtual probe, and specifies if the virtual probe goes to the READY state or not. It defines

first the actual monitoring granularity (Dmg or Wmg), and following if the logical expression is verified or not, the probe switches or not to the READY state. The freezing action is triggered if it is set (Enabled), and if the virtual probe exceeds the hard threshold. The OPv (Old Probe value) is used to determine if the VP (Virtual Probe) takes into consideration the error tolerance, when it is switching between a Wmg and a Dmg. The error tolerance should only be considered when the VP switches from the Wmg to the Dmg, and not in the opposite case. Otherwise, the warning threshold value becomes the old value plus the error tolerance value, and the yo–yo behavior can't be handled.

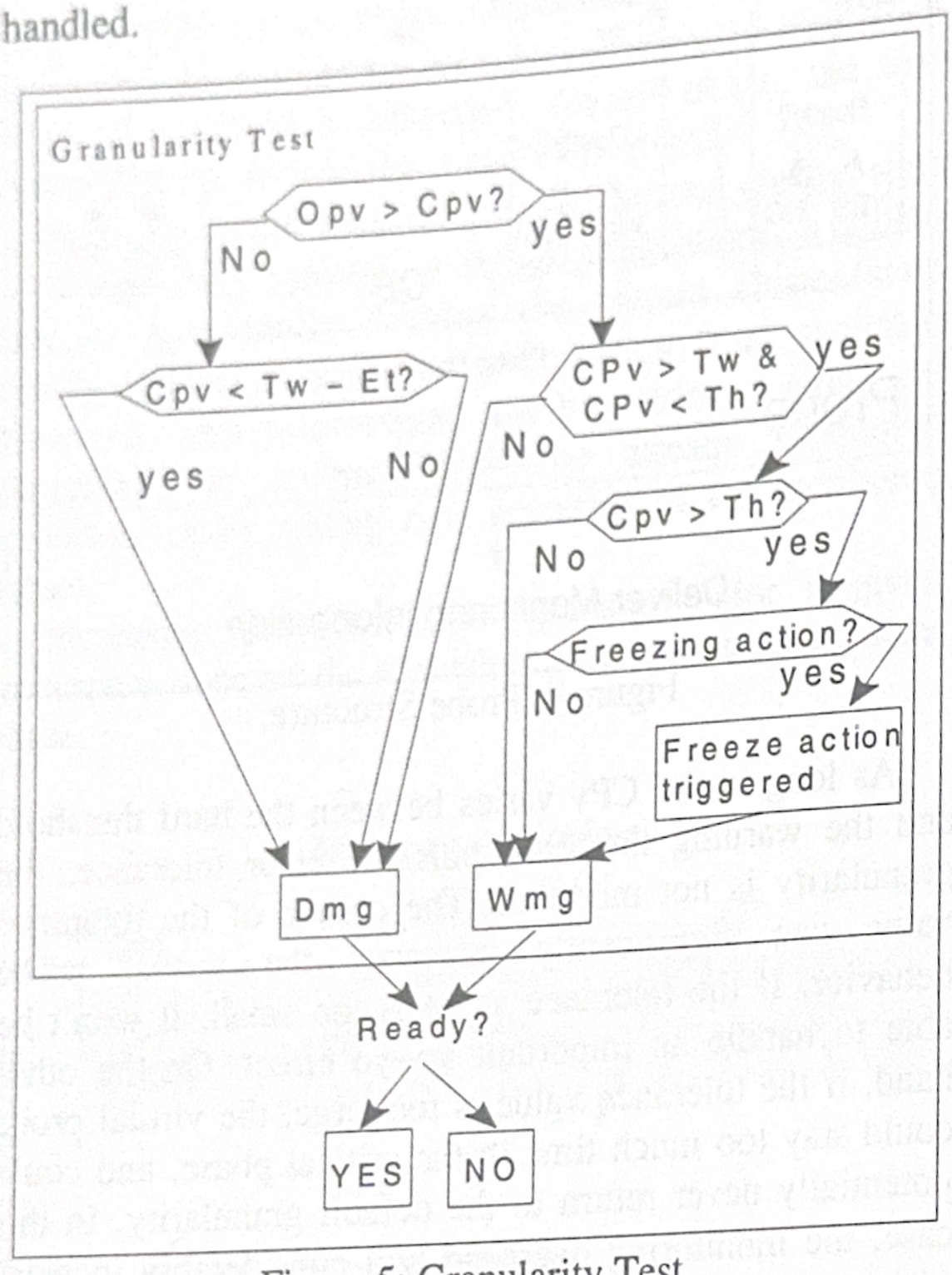

Figure 5: Granularity Test

The general algorithm of a probe is given below, and which is executed at each use of a monitored call. It executes the monitored call, and then tests if a monitor information should be delivered. Thus, it tests sequentially each virtual probe until it founds a READY one. If a virtual probe is in a READY state, the probe get into the READY state and then sends the information. If no VP is in the READY state, the probe stays in the NOT READY state and no information is delivered. At the same time, if the freezing action is enable, the probe freezes the application and notifies the tool.

```
Probe initialization {
disable all the freezing actions
Set the monitoring at the default
 granularity
}
At each monitoring call use {
 execute the call
 Get the Current Probe value (CPv)
 For each Virtual Probe (VP) {
   perform the granularity test
   if freezing action is set {
      trigger the the freezing action
   }
   if the current VP state is READY {
      set the current probe state to
         READY
      exit
   }
   go to the next VP
 }
 if the probe state is READY {
   send the monitoring information
   set all the VP and probe state to
NOT READY
 }
}
```

Probe Algorithm

The general definition of a probe is given below as well as an example of a memory probe with one virtual probe.

```
Probe resource {

 Virtual Probe {
   warning_threshold = value;
   hard_threshold = value;
   error_tolerance = value;
   freezing_action = value;

   default_monitor_condition = {
      logical_expression
   };
   warning_monitoring_granularity = {
      logical_expression
   };
 }
 .
 .
 .
 }
```

Probe definition

```
Probe Memory {
 Virtual Probe {
   warning_threshold = 55MB;
   hard_threshold = 60MB;
   error_tolerance = 2MB;
   freezing_action = ON;

   default_monitor_condition = {
     DELTA(MEM_VOLUME) = 4MB;

   };
   warning_monitoring_granularity = {
     DELTA(MEM_CALL_NBR) = 1;

   };
 }
}
```

Example of a probe definition

In this example, the default monitoring condition is defined as follow; an information is required at each time that the memory volume exceeds 4MB the current volume. Besides, when the virtual probe enters the critical phase (Tw > 55MB), a monitored information is required at each use. The error tolerance is set to 2MB, and allow to handle a yo-yo behavior of 7MB between 53MB to 60MB not included. The freezing action is triggered when the allocated memory exceeds 60MB.

VI. Performance

Some available results from other value added tools varies between 9.5% and 91%: RECPLAY produces an average overhead of 91% [RON 99], whereas GatoStar decreases the response time from 14% to 20% [FOL 94], and TRAPPER induces only a total overhead of 9.5%. At the same time, the intrusion of the PHOENIX version composed of non-adaptable probes that just performs tracing resources and transmitting the information at the requested granularity, is less than 5% [BOU 00].

A simulation of the new version of the PHOENIX plat-form using self adaptable probes has been developed. To simulate a real environment, we plugged a management tool into the platform. This tool was developed using the PHOENIX library. Its goal is to receive all the traces and to display them. It was executed on a different host than the applications processes (a 266 Mhz Pentium II). Application's processes executed on a network (100 Mbits/s switched Ethernet) of 3 x 500 Mhz Pentium III workstations.

Performance has been measured using the Red-Black SOR [LU 97] developed under the TreadMarks [AMZ 96] distributed shared memory (DSM), which provides the abstraction of a globally shared memory. TreadMarks supports parallel computing on networks of workstations and runs at user-level on Unix systems. TreadMarks produces a lot of system calls due to its virtual memory management. Both TreadMarks and the Red-Black SOR have been recompiled to automatically include PHOENIX self adaptable probes.

The Red-Black Successive Over Relaxation (Red-Black SOR) is a method to solve partial differential equations by iterating over a two dimensional shared array. It takes as parameters, the size of the two dimensional array, the number of iteration to execute, the list if workstations and the number of processes on each workstation.

We measured the execution time (system and user) for the original SOR application, for a version linked with the original PHOENIX library, and another version linked with the PHOENIX library that uses self adaptable probes. For the original PHOENIX library version, we measured the execution time for a systematic monitoring (at each usage of a monitored resource). We measured the execution time for different values of the dimension array, and different iteration numbers.

Figure 6 shows a good average slow down factor for the tracing using self adaptable probes. In fact, the overhead average dropped from 2.72% to 0.40%, which is a very low overhead.

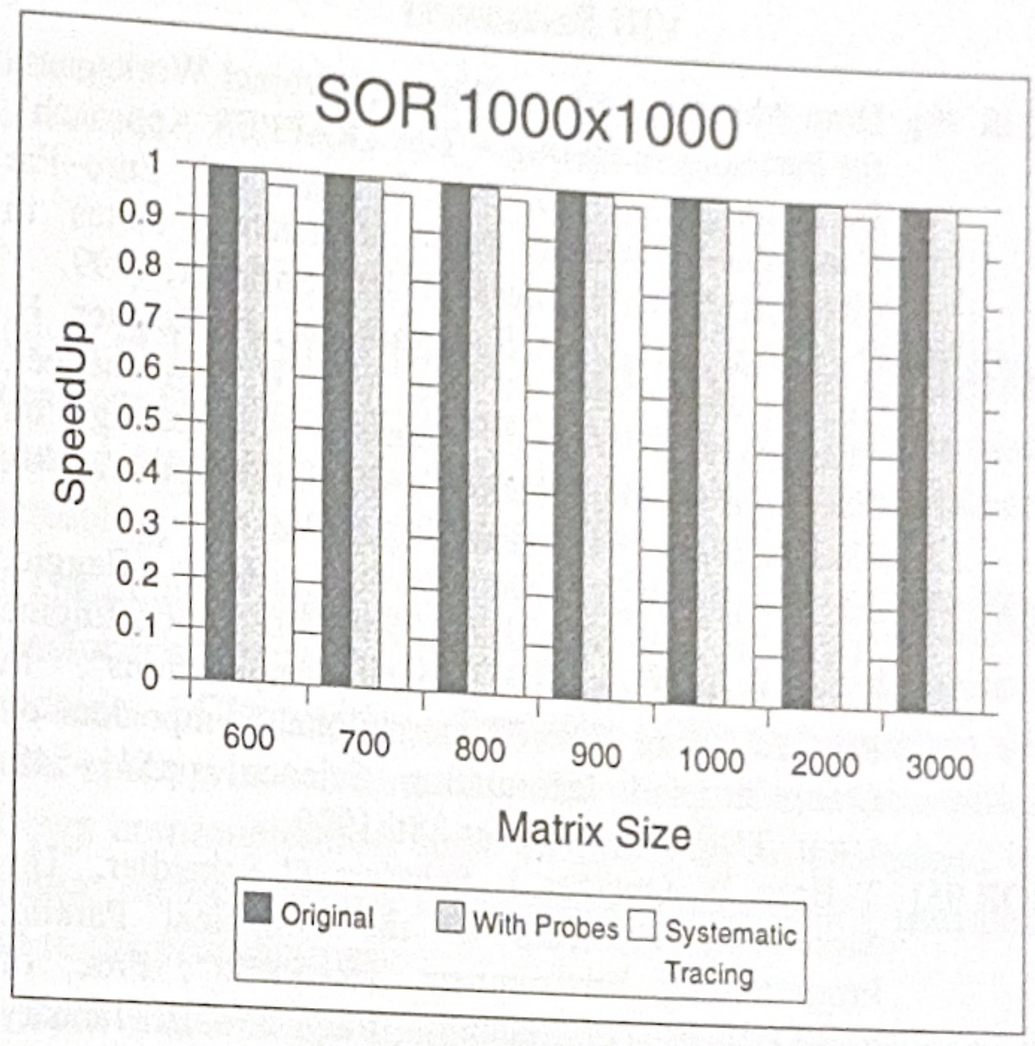

Figure 6: SpeedUp Diagram

As the adaptable probes reduce the amount of the generated communication without inducing further processing since the treatment is simply transferred from the tool level to the probe level, the performance of the version of PHOENIX with self adaptable probes is significantly improved. In the worst case, it is similar to the old version of PHOENIX, which is less than 5%. Another important advantage of this approach is the possibility of handling an emergency state, which usually ends up with a system or application crash.

VII. Conclusion

In this paper, we propose to use self adaptable probes in a generic monitoring platform. This approach allows to significantly reduce the overhead generated by the monitoring platform as it uses fine grain monitoring only when it is strictly necessary.

The granularity tuning is handled by the self adaptable probes rather than by the tools, which further reduces the granularity control between the tools and the probes, and thus the overhead. Moreover, the self adaptable probes ensure that the tool will be able to process an emergency state of a given resource using a reliable detection mechanism.

As a future work, we plan to develop a set of value added tools on top of the PHOENIX platform, and especially exploit its emergency state capability to build an efficient load balancing tool. We also expect to be able to automatically generate self adaptable probes definitions from high level distributed application specifications.

VIII. References

[AHR 99] Dino Ahr and Andreas Bäcker, "Project Workspaces for Parallel Computing – The TRAPPER Approach", Proceedings of the 5th International Euro–Par Conference, volume 1685 of Lecture Notes in Computer Science, pages 98–107. Springer, 1999.

[AMZ 96] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations", IEEE Computer, Vol. 29, No. 2, pp. 18–28 (February 1996).

[BON 95] Xavier Bonnaire, Daniel Prun, Aline Baggio, "Intrusion Free Monitoring: An Observation Engine for Message Server Based Applications", In Proceedings of the 10th nternational Symposium on Computer and Information Science, pp541–548, Izmir, TURKEY, October 28th1995.

[BOR 95] T. Born, W. Obelöer, L. Schäfers, C. Scheidler, "The Monitoring Facilities of the Graphical Parallel Programming Environment TRAPPER", Proc. of EUROMICRO '95, Sanremo, Italy, 25–27th January 1995, IEEE CS Press.

[EUS 95] Alan Eustace, Amitabh Srivastava, "ATOM: A Flexible Interface for Building High Performance Program Analysis Tools", USENIX Winter 1995: 303–314.

[BOU 00] Boutros Saab Céline, Bonnaire Xavier, Folliot Bertil, "A flexible Monitoring Platform to Build Cluster Management Services", IEEE International Conference on Cluster Computing Cluster2000, November 28 – December 2, 2000, Saxony, Germany.

[FAU 99] Faugère Jean–Charles, Folliot Bertil, Boutros Saab Céline, "Execution platform for high consuming parallel application s: a case study for Grobner basis", International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications on July 5 – 7, 1999 at Tokyo University.

[FOL 94] Bertil Folliot and Pierre Sens, "GatoStar: A Fault Tolerant Sharing Facility for Parallel Applications", Lecture Notes in Computer Science 852, Pages 581–598, Octobre 1994.

[LU 97] Lu Honghui, Dwarkadas Sandhya, Cox Alan, and Zwaenepoel Willy, "Quantifying the Performance Differences Between PVM and TreadMarks", Journal of Parallel and Distributed Computation, Vol. 43, No. 2, pp. 65–78, June 1997.

[MAN 93] Masoud Mansouri–Samani and Morris Sloman, "Monitoring Distributed System (A Survey)", Imperial College Research Report No. DOC92/23, April 1993.

[MIL 95] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchitapadam and Tia Newhall, "The Paradyn Parallel Performance Measurement Tools", Special Issue on performance Evalaution Tools for Parallel and Distributed Computer Systems, Pages 37–46, November 1995.

[NET 92] Robert H. B. Netzer, Barton P. Miller, "Optimal Tracing and Replay for Debugging Message Passing Parallel Programs", Supercomputing ' 92, November 1992, Minneapolis.

[NOE 94] Roger J. Noe, "Pablo Instrumentation Environment User's Guide",University of Illinois, Urbana, Illinois 61801, April 1994.

[RON 99] Michiel Ronsse and Koen De Bosschere, "RecPlay : a Fully Integrated Practical Record/Replay System", Universiteit Gent, Belgium 1999.

[RON 97] Michiel Ronsse, Koen De Bosschere, "Work in progress: An On–the–fly Data Trace Detector for Recplay, a Record/Replay System for Parallel Programs", In the 16th ACM Symposium on Operating System Principles (Work in progress), Octobre 1997.

[SRI 94] Amitabh Srivastava, Alan Eustace, "ATOM - A System for Building Customized Program Analysis Tools", PLDI 1994: 196–205

[TEL 94] Gerard Tel, "Introduction to Distributed Algorithms", Cambridge University Press, 1994.

[ZHO 93] S. Zhou and X. Zheng and J. Wang and P. Delisle, "Utopia: a load sharing facility for large, heterogeneous distributed computer systems", "Software–practice and experience", 1993, Volume 23, Numéro 12, Pages 1305\1336.