Minimal Invasive Monitoring

Daniel Mahrenholz
University of Magdeburg
Universitätsplatz 2
D-39106 Magdeburg, Germany
mahrenho@ivs.cs.uni-magdeburg.de

Abstract

Using software monitors to measure embedded or real-time applications is a difficult task because of their impact on code size and execution time. Our approach for a minimal invasive monitor shows how to minimize the impact by adapting the monitor to the monitoring task. We present techniques to reduce the monitor overhead to the most necessary.

1. Introduction

Besides simulation and analytical modeling, measurement is one of the three evaluation techniques available for software systems. To measure the properties of a software system you have to know about its internal state. To make this internal state visible to the outside world monitoring systems are used.

There are three types of monitoring systems: software, hardware and hybrid systems. Software monitors are software extensions to an existing system. In contrast hardware monitors log the states of the different hardware buses to track the software's behavior. Hybrid monitors take advantage of both methods - they use software extensions to generate special signal patterns on the observed buses and external hardware devices to recognize and record the events. Both approaches have their pros and cons. Software monitors on the one hand are cheap and highly flexible but they perturb the monitored system. Hardware monitors on the other hand do not necessary interfere with the target system. But these systems highly depend on the target processor and thus are expensive and inflexible compared to a software monitor.

So why should we use software sensors, especially in a real time environment? We can not measure whether a system is real time capable or not, i.e. if it will meet its deadlines in the future. This is subject to the system design. But we can assist in making statements about the time behavior of the system. If an algorithm is real time capable it is not said that it works with a particularly hardware. It can fail only because the hardware is to slow to run the algorithm in a given time. But it can also fail because the software monitor introduces to much overhead. So the software extension have to be as small as possible. There is another reason why a monitor extension should be minimal. If you try to monitor systems used in the deeply embedded market, it is likely to find systems with very limited memory resources. Memory sizes of 8 KB are not unusual. And if you consider an operating system with an application with a total size of 5 KB there are only 3 KB left. So the monitoring system has to be minimal to fit into the system at all. Even if the remaining memory is too small to store a reasonable large buffer to keep the measured values, the software monitor can provide an external hardware monitor with information about the internal behavior of the system.

2. The Nature of Software Sensors

2.1. Trigger a Sensor

Before a sensor can log an event it has to be triggered. As we focus on object-oriented languages we also take advantage of their possibilities. We have two types of sensors, block and expression level sensors. Expression level sensors are inserted to log the pass of a single code position or the state of a variable at this point. Block level sensors log entries into and/or exits from basic code blocks (e.g. functions).

Expression level sensors are simple calls to a logging function inserted as expression statements or by extending simple expressions to compound expressions. Block level sensors in contrast are inserted as variables instantiating a class with a scope local to the block. As they were class instances they have to be initialized at the beginning of their scope und destructed at the end. So it is guaranteed that the class constructor is called whenever the block is entered

and the destructor when it is left. So we are able to monitor entries and exits to a block separately.

By using an optimizing compiler these sensors can be fully inlined so that even the block level sensors do not need extra space in the data segment for the class instance.

2.2. Sensor Parts

Sensors consist of three parts: measurement, data storage and sensor management. The measurement part collects the data of interest. The storage part decides how and where to store the gathered information. At last the management part connects the sensor with the monitoring system. Mostly these three parts can not be fully separated – especially the collection and storage of interesting data often go hand in hand. But logically they do different jobs, so take a closer look on them.

The primary job of a sensor is to collect data about the system's current state. The system behavior can then later be described in terms of state changes by combining the data collected by all the sensors during the system's runtime. So it is clear that we focus on logging state changes in the system – something that influences the placement and design of sensors, that both can be optimized.

The measured values have to be made visible to the outside world. To do so there are two possibilities. First to store them to an external device every time they get measured or second to cache them in the local memory and to store them later as a whole. We use the second one as it involves the smallest communication overhead. If we are going to use an external hardware monitor, we switch to the first possibility. In this case we write to a special memory location and catch all writes with the external monitor.

The management part of a sensor interacts with the software monitor to synchronize its work with the other sensors especially when writing into the global trace buffer. It also ensures that the global trace buffer does not overrun.

3. Sensor Optimization

3.1. An Example

Figure 1 shows a small part of the PURE[1] [8] class hierarchy. As an example we show the steps needed to monitor the context switches of a PURE system. In the example system we have two possible reasons for a context switch, a direct or indirect[2] call to `Coroutine::resume` (cooperative switch) or a preemptive switch caused by a timer event.

[1] PURE is an object-oriented operating system family that mainly targets the area of deeply embedded systems. It is a development of our group.

[2] With indirect calls we mean all calls to functions that itself calls the target function (in)directly.

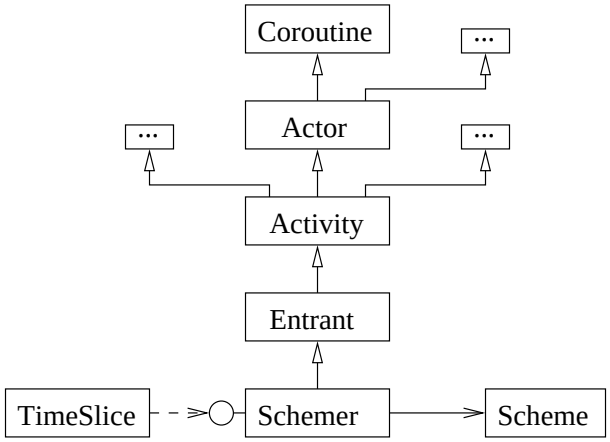

Figure 1. Partial PURE class hierarchy

Our goal now is to measure the time spend in the different classes in order to find inefficient implementations and to verify the expected runtime behavior. We do this by inserting block level sensors into all methods calling `Coroutine::resume` directly or indirectly. Block level sensors are triggered on all entries to or exits from a function.

To do this instrumentation we use the aspect-oriented programming techniques[7] provided by our aspect weaver suite. The specification of such an instrumentation is quite simple. We first place a block level sensor in `Coroutine::resume`. Second we search for all call paths leading to this function and instrument the passed functions accordingly. This is done by recursively searching for functions that are already instrumented. The result of the instrumentation process are the instrumented source code and a description of the planned sensors. In a second step this description is used to generate the source code for all sensors. This separation enables us to optimize the sensors without rerunning the whole instrumentation process.

Figure 2 depicts the placement and relationship of the inserted sensors. If not specified in the instrumentation process all sensors are full-featured and log directly into the global event trace.

3.2. Optimization

The basic instrumentation and sensor generation process provides a ready-to-run system. But this basic instrumentation can be optimized in several ways. First we can strip down the sensors so that they only store significant values that could not be reconstructed afterwards and remove all functional parts that are never used. If we know (e.g. from a previous run) that a sensor is not used, we can remove it completely. Second we can move parts of the sensors out of the critical execution path to minimize their impact on the

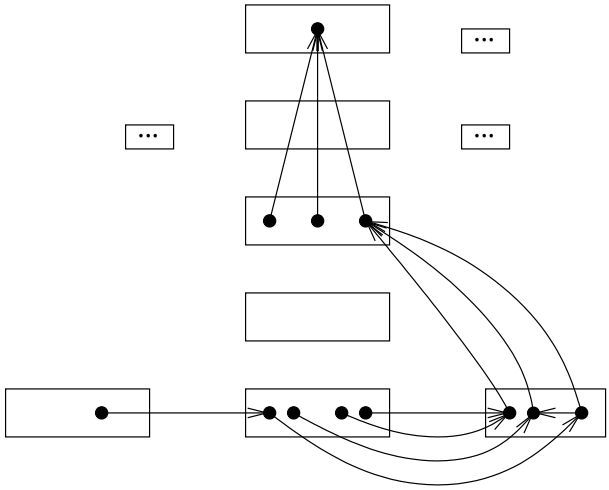

Figure 2. Initial sensor placement and call graph

measured execution time.

Another optimization possibility that comes into mind is whether to inline the sensor code or to call it as a function. Inlined sensors have smaller execution times. Sensors called as functions instead have a smaller code size but need extra execution time to prepare and perform the call. Another disadvantage is that they can hardly be optimized by the following techniques. So we focus an inlined sensors that can highly be optimized for size and speed. It is even possible so make them smaller than their counterparts.

3.3. Strip Redundant Sensor Information

The first and easiest way to optimize a sensor is the removal of redundant storage operations. As we use the JUWEL format[2] as our native output format to support various external applications we have to provide a complete JUWEL style record for every event occurring. But we are not forced to fill in each field. We only have to ensure that all fields can be filled out afterwards (e.g. by an external postprocessor). So we can leave out all fields that are not used, have not changed since the last event or that can be reconstructed by the help of other information (e.g. the call graph of the instrumented system).

The removal of redundant storage operations can easily be done by hand during the instrumentation process or afterwards. In our case we do not need the `Parameter` or `ProcessorID` field so we configure them out. A second thing we do not need are the boundary checks that protect us from overrunning the global trace buffer. As we have enough memory to hold a fairly large trace buffer we can also configure these sensor parts out. If the available memory is tight we have to check the buffer regularly to save it to an external storage if it gets to full. Normally this check is done every time we log an event. This behavior is safe but far from optimal so it is subject to the second optimization strategy.

Field	Size (Bit)
EventID	32
Timestamp	64
ProcessorID	32
ThreadID	32
Parameter	32

Table 1. JUWEL event record

The usage of the remaining three fields can be further optimized. First we minimize storage of the actual thread ID. We analyse the whole system to find all write accesses to the memory location that holds the ID of the currently running thread (*life pointer*) and set the first sensor before and thereafter to store its value. All other sensors need not to store it, because the thread ID can be reconstructed. If the ID is not encapsulated into a class the automated analysis can get very tricky. In this cases it is often better to specify all position were it changes manually.

Next we take a look at the time stamp field. It is 64 bit wide to support high resolution timers (e.g. processor cycle counters). Suppose we have a fast processor running at a clock speed of 500 MHz the lower 32 bit of the counter will overrun after 8.6 seconds. So in most cases the lower 32 bit are enough to recognize each overrun and so to reconstruct the exact time stamp value. This saves a memory access operation on 32 bit processors and even more on smaller hardware. For all processor types and clock speeds specific optimizations can be found in the same way.

The last interesting field is the event ID. Figure 2 depicts the part of the system's call graph containing our sensors. Figure 3 shows the call graph in a second version denoting split and join points. It is clear that there are several dependencies between these sensors. For instance, if you pass sensor 8 the next three have to be 5, 3 and 1. It is also clear that by knowing these dependencies some sensor IDs can be guessed afterwards so they do not need to be stored. In the example we have only instrumented nodes. But it is also possible to have nodes without sensors. They are treated like nodes with unidentified sensors. If a call is only performed under certain conditions we assume a call to a virtual sink node if the condition is not met.

Our goal now is to find a minimal set of identified sensors so that we are able to identify each possible path through the graph. The difficulty of finding such a set increases with the number of sensors, split und join points and soon reaches a level at which it can not be properly done by hand so an

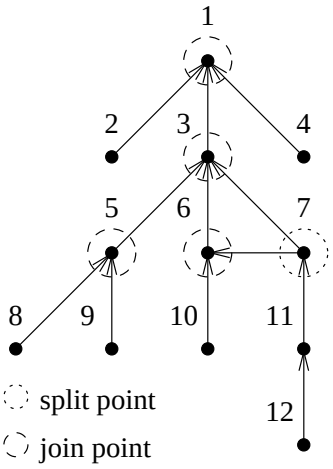

Figure 3. Call graph

automated system is needed (see section 4.2).

To test a set of sensor configurations you can use the following method. First go through the list of split and join points and remove the selected node temporary from the graph. Second test if all paths in the resulting sub-graphs that lead to the selected node (for a join point) or from it (for a split point) can be clearly identified by the chosen sensor configuration. If the resulting sub-graphs include other split or join points apply this method recursively.

3.4. Move Sensor Parts

After we have stripped down the sensors as much as possible we now try to move necessary parts to other locations to reduce the sensors' impact on time critical code sections. Another goal is to concentrate sensor parts so that their time utilization can be measured using our own techniques. This can help to reduce the measurement error caused by the sensors. Two questions remain: which parts can be moved and where shall they be placed?

Nearly all parts of a sensor can be moved. But the management part is particularly suitable because it has little to do with the measurement in principle. It synchronizes the sensor with others when accessing the global trace buffer, prevents the trace buffer from overruns and so on. Especially synchronization with other sensors can result in an unpredictable time behavior. The boundary checks "only" costs us time. Both things are not very welcome to perform a highly precise measurement. Our solution to this problem is relatively simple. We extend the sensor with a small buffer to keep the measured values and transfer them later into the global trace buffer. This requires additional memory space for the buffers and processor cycles for the temporary writes. But if it is possible to move the management parts of all sensors in a measured region to the outermost sensors (leaves in figure 3) or completely out of the region you get a highly precise measurement because all sensors are as small as possible and have predictable execution times.

Second there is the question where to insert the removed parts of a sensor. The best solution is to insert the management and transfer code into the class destructor of the outermost sensor or into the program code that is called whenever the program exits. Our little example works with this solution but in most cases it will not. The crucial problem in this situation are loops around sensors that causes the sensor to be triggered multiple times. If there is a small upper limit for the loop count this problem can be solved by assigning a larger buffer to the sensor to cache several events. But this is not satisfying at all. So our general solution to this problem is to insert the management code into the last sensor that would be passed before the loop closes.

There is another crucial point when using cached values. Depending on how sensor parts are moved around, the order of events in the resulting log file can be changed. In general this is not a problem because events can be reordered with their time-stamps. But if you use a slow timer several events may have the same time-stamp. In this case it helps to consult the call graph to order the events according to their unique ID.

At last another possibility to move sensor parts to has to be mentioned. From our point it is hypothetical because we never tried it. But if you are using an external hardware monitor, parts of or even the whole sensor can be moved to the device connected to the hardware monitor. If you move sensors partly you get an hybrid system. The remaining sensor parts would then perform write operations that the hardware monitor would recognize. The moved parts would then store the values caught by the hardware monitor. If you move the whole sensor the monitored system would get instrumented with zero size sensors (e.g. debugging symbols). This would enable the hardware debugger to trigger the external sensor code. But without access to the target system's memory it would be really difficult to monitor things like the currently running thread.

4. Automated Instrumentation and Sensor Optimization

After the discussion of the general optimization strategies we now discuss how this can be automated.

4.1. Using AOP

In general monitoring has nothing to do with the actual tasks of a system. It is a technical requirement to the system that is only needed temporary. And even if it is needed, the

first question always is where and not how to monitor somethings. Speaking in terms of the aspect-oriented programming (AOP)[1, 5] this is called an *aspect* of the system. So we model a monitoring aspect and merge it with the system. This is called *aspect weaving*. Figure 4 shows the whole instrumentation process. The *aspect program* also allows us to specify special sensor configurations and strategies to move sensor parts in the system.

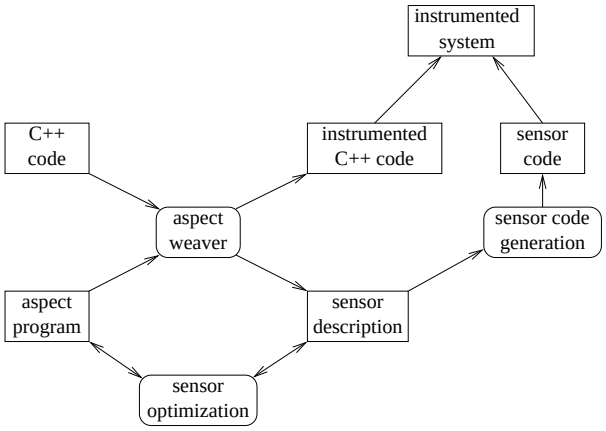

Figure 4. Instrumentation process using AOP

4.2. Identify Redundant Sensor Information

As explained in section 3.3 the removal of redundant storage operations can not very well be automated. For all fields but the sensor identifier and time-stamp the problem is to find all instructions that write to the memory positions storing the values the sensors uses to fill out the event records. For a well structured system that respects the principle of data encapsulation this can be done easily. But C++ does not force you to do so. So this could result in a highly expensive data flow analysis.

The removal of sensor identifiers can be fully automated. The best way is to use the brute force method that tests all possible settings. The disadvantage of this approach is the exponential complexity of the solution. With some algorithmic optimizations and lots of computational power it is possible to repress the problem but you will always find a number of sensors that is to large. You could for instance try to find call chains without split and join points and remove the identifiers from the inner nodes because they are clearly identified by the first and last sensor of the chain. This also reduces the total number of sensors to optimize. You can also try to find solutions for different numbers of identified sensors using a bisection method. If this does not help you can use heuristics to minimize the number. You get a sufficient solution by identifying all sensor before and behind split and join points. But it is unlikely that you get a minimal solution this way.

4.3. Moving Sensor Parts

The basic strategy to move necessary sensor parts is to search the call graph (which is a directed graph) for all possible paths and to delay execution of parts of sensors found along these paths as long as possible. This works in three phases. First we identify entry, exit, join and split points of the system. Second we reduce the call graph to the parts containing our sensors. And third we move the code parts along the paths.

Entry and exit points are defined by the system. Entry points are nodes were a new control flow starts. These are the starting point (function) of a system, interrupt service functions and the very first function called after thread or process creation. Exit points are those function that end a control flow. These are functions that jump out of the system, destruct a thread or process or do things like stopping the processor forever. Join points are nodes that have more than one incoming edge, split points respectively have more than one outgoing edge. The initial entry and exit points have to be specified for the target system.

To reduce the call graph to the interesting sub-graph we first remove all isolated nodes and sub-graphs that do not contain any sensors. Thereafter we look for all nodes that have no sensors and only incoming or outgoing edges. They lie on the border of the call graph and were never passed on a path between two sensors. We apply these steps as long as we can reduce the graph. If we remove a node marked as an entry or exit point the connected nodes become new entry respectively exit points.

The last phase requires a cycle-free graph so we first search for all cycles that itself do not contain other cycles. If we find one we separate it from the rest, duplicate the nodes that belong to both parts and remove all edges leading to the last node of the cycle. We assume the node where the cycle close the first node of the cycle. Figure 5 should give an idea of this process. As the result we get a set of cycle free call graph fragments.

For each fragment we search all contained sensors and move their management parts contrary to the edge direction along the path. We then place the new management code in the furthermost sensor possible. If we pass a node marked as join point along the path we have to follow all edges leading to this node. This means we possibly have to insert the moved management code into several sensors. But this is not a problem because all these sensors share a special management code that is capable of storing a group of cached event records at once. Simultaneously we change the sensor types. The sensor we remove the code from is changed

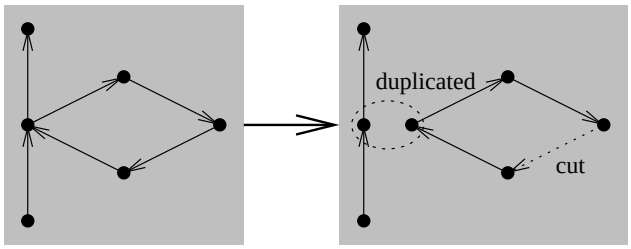

Figure 5. Removal of call graph cycles

to use a temporary memory for its values. The sensor we place the code in is changed to store the values of the first sensor's temporary memory. Figure 6 shows where sensor parts were removed and inserted. But keep in mind that not the actual code is moved. Its functionality is now adduced by another piece of code at a different place.

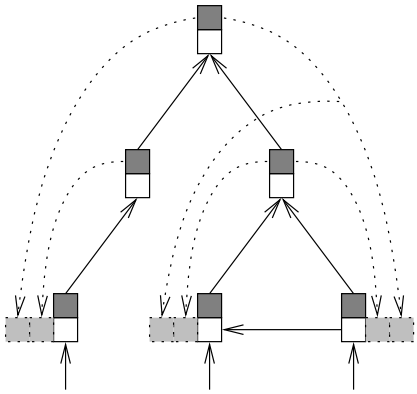

Figure 6. Movement of sensor parts

4.4. Using Monitoring Results

Remember the example in section 3.1. There you can see that we inserted several sensors into functions that lie on paths leading to `Coroutine::resume`. We did this because we do not know which paths will be used. But we will know it if we run the instrumented system and analyse the resulting trace log. Although a sensor is not used according to the trace log it is not guaranteed to be redundant. But it gives an idea of what parts of the system were really used. So this assists in removing redundant sensors completely. This has to be done before all other optimizations because it can strongly influence them. And if you decide to remove all unused sensors this can be done fully automated.

Someone could argue that finding unused sensors could also be done by an static analysis. This is true in many cases. But there are also some difficulties connected to this approach. There are several code sections (e.g. interrupt handlers) that are only reached in reaction to an external event or when processing certain inputs. So you have to consider all these external factors in the analysis process.

5. Results

We discussed a lot of optimization possibilities in the last sections. Now we take a look at what this means in practice. Our testbed is a Pentium 133 MHz with 60 ns EDO-RAM. This is not a typical embedded platform but it provides a clock cycle counter to run a highly precise measurement.

We start with the example of section 3.1. We added 12 sensors and the runtime extensions for the software monitor. All sensors are at function level and generate an event on entry and exit of the function. So the sensors generate 24 unique events. Table 2 shows the initial sizes for the system.

Part	Code	Data	Sum
communication interface	778	104	882
event generation/storage	479	28	507
sensors	2496	0	2496
overall	3753	132	3885

Table 2. Initial sizes (in byte) of the monitoring extensions

In section 3.3 we discussed how to remove redundant parts of the sensors. Table 3 shows the savings we could get from leaving out different sensor parts. We can not remove all fields from all sensors. But the sum of all sensors drops from 2496 to 426 bytes (on average 35.5 bytes per event).

Parts	Code saving (byte)
boundary checks	32
optional parameter	10
processor ID	10
thread ID	6
event ID	9
time-stamp (upper 32 bit)	5

Table 3. Savings from removal of redundant sensor parts (normal sensors)

Our next step in section 3.4 was to delay execution of sensor functions as long as possible. In this step we also changed the basic behavior of the sensors. We moved the management part away from most sensors and modified them to cache their values in a fixed memory location. This reduces the size of a full sensor to 53 bytes. In contrast

all sensors that now get the job to store the cached values grow by 10 bytes at each case. The new caching sensors can also be stripped down. Table 4 shows the savings we can achieve. The minimal sensor we can create now is 7 bytes small. The code size needed for all sensors now drops to 344 bytes which means an average sensor size of 28.7 bytes.

Parts	Code saving (Byte)
optional parameter	10
process ID	10
thread ID	10
event ID	10
time-stamp (upper 32 bit)	6

Table 4. Savings from removal of redundant sensor parts (caching sensors)

With the complete set of optimization described in the previous sections, we were able to reduce the overhead introduced by the software sensors from 2496 down to 344 bytes. This is a reduction of about 86%. Table 5 gives an overview of the final results.

Two things have to be mentioned at this point. First the effects of the optimization on the sensors' execution times and second the cache effects that appear. The execution time of a sensor depends much more on the target hardware as its size does. In our test case a full functional, not optimized sensor had an average execution time of 60 cycles (450 ns). The minimal stripped down sensor took 20 cycles (150 ns) on average. At last the smallest possible sensor took 8 cycles (60 ns) to run. This is exactly the access time of the memory we used. So it seems that our sensors are more affected by the speed of the memory than by the speed of the processor. As we mentioned before you have to take care of cache effects. When triggering a sensor for the first time its execution time can be 2 (largest sensor) to 10 (smallest sensor) times higher as it is later. There is almost nothing you can do about this because it is impossible in most cases to put the sensor code and data into the cache without affecting other parts of the system.

6 Related Work

The monitoring of software systems using hardware, software and hybrid systems[12] has been studied for years. Many works dealt with the techniques used to instrument a system at compile or runtime in various use cases. Dynamic instrumentation systems such as Paradyn[3], KernInst[10] or the system presented in [4] insert the instrumentation code into running binaries. Other binary rewriters like

	not optimized	stripped down	fully optimized
Full sensor	104	104	53
Minimal sensor	104	32	7
Example (sensors)	2496	426	344
Example (per event)	104	35.5	28.7

Table 5. Summary: possible sensor sizes (byte)

EEL[6] are used to instrument a system before it is started. They all share the same problems. To call the sensor code they have to replace code sections to perform a call or trap to it. And without access to the source code they can not gain much knowledge about the internal state of the monitored system. That's why we focus on source code instrumentation like the TAU[9] or JEWEL[2] system. We differ from them in two ways. First that we use an aspect-oriented approach to perform the instrumentation (s. section 3) as well as transformation networks[7] to model and specify the monitoring task, special sensor requirements etc. And second that we use generated software sensors to adapt them to the monitoring task to be as small and fast as possible and to ensure a predictable runtime if necessary. [11] discusses the effects of software sensors on distributed real-time system and how to deal with them. They present a method for deterministic observations of real-time systems.

7. Conclusion

Software monitors are a cheap and highly flexible way to monitor the behavior of software. But their perturbation of the target system makes them unsuitable or even impossible for embedded and real time systems.

By using aspect-oriented programming techniques to describe monitoring tasks and system internals and a set of automated optimization tools we make it possible to adapt the monitor to the task. Our work shows that the combination of a fine grained instrumentation with several optimization strategies can generate a unique software monitor minimal in size and execution time. This enables us to perform software (supported) monitoring in areas dominated by hardware solutions.

References

[1] K. Czarnecki. *Generative Programming: Principles and Techniques of Software Engeneering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Technische Universität Illmenau, Germany, 1998.

[2] Martin Gergeleit. *Automatic Instrumentation of Object-Oriented Programs*. Technical report, German National Research Center for Information Technology, 1994.

[3] Jeffrey K. Hollingsworth et al. *MDL: A Language and Compiler for Dynamic Program Instrumentation*. Technical report, Computer Sciences Department, University of Maryland; Computer Sciences Department, University of Wisconsin, May 1997.

[4] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceeding of Scalable High Performance Computing Conference*, 1994.

[5] Gregor Kiczales et al. *Aspect-Oriented Programming*. Technical report, Xerox Palo Alto Research Center, 1997.

[6] J.R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*, June 1995.

[7] Daniel Mahrenholz. Aspektorientierte Realisierung eines generischen Systemmonitors. Master's thesis, University of Magdeburg, Germany, 2000.

[8] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. *Design Rationale of the* PURE *Object-Oriented Embedded Operating System*. In *Proceeding of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (Dipes '98)*, volume 5/6, Paderborn, Germany, October 1998. ISBN 0-7923-8614-0.

[9] S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable profiling and tracing for parallel scientific applications using C++. In *Proceedings of ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT '98)*, August 1995.

[10] Ariel Tamches and Barton P. Miller. *Fine-Grained Dynamic Instrumentation of Commodity Operation System Kernels*. Technical report, Computer Sciences Department, University of Wisconsin, 1998.

[11] Henrik Thane. Design for deterministic monitoring of distributed real-time systems. Technical report, Mälardalen Real-Time Research Centre, 2000.

[12] J. Tsai, Y. Bi, S. Yang, and R. Smuth. *Distributed Real-Time Systems, Monitoring, Visualization, Debugging and Analysis*. Wiley, 1996.