The Monitoring Request Interface (MRI)

Edmond Kereku and Michael Gerndt

Institut für Informatik
Technische Universität München
{kereku, gerndt}@in.tum.de

Abstract

In this paper we present MRI, a high level interface for selective monitoring of code regions and data structures in single and multiprocessor environments. MRI keeps transparent the available monitoring resources from the performance analysis tools and can electively generate monitoring results as online profile information, or as postmortem traces. MRI is the first step toward a standard monitoring interface which can be used by a broad range of performance analysis tools, from profiler tools, trace producers and visualizers, up to complex automatic performance analyzers. We also present an implementation of MRI for SMPs which transparently use a simulation backend and a PAPI backend to obtain performance data.

Keywords: Performance Analysis, Monitoring, Cache, Instrumentation, Interface, C API.

1 Introduction

Performance analysis for high-performance systems is based on monitoring the dynamic behavior of the application during its execution. The information produced during runtime is either stored in files and inspected after the program terminated (offline analysis) or delivered to analysis tools while the application is still running (online analysis). The monitoring for current performance analysis tools is achieved in a very tool-specific way.

In contrast to this approach, we suggest to define a standard interface, the Monitoring Request Interface (MRI), which permits the performance analysis tool to request the available information from the monitors. The most important advantage of this approach is to keep the monitoring resources transparent from the performance tool. Further development of hardware counters or simulators used to collect performance data, does not require any changes in the performance tool.

Another advantage of MRI is, that it allows selective monitoring for individual code regions. For monitoring memory hierarchies, it allows one to focus the monitoring on accesses to individual data structures. With the latest developments in hardware counters it is now possible to only collect events from a restricted memory address range[4] and MRI makes this available to performance analysis tools.

MRI also supports monitoring in multiprocessor environments. From simple SMPs to thousand-processor teraflop machines, multiprocessors are the de-facto running environment of HPC applications.

MRI can be divided into four interfaces.

The Monitor Configuration Interface which serves to online configure the monitoring resources.

The Data Retrieval Interface. This interface provides means to deliver the determined information at runtime to the analysis tool. If the analysis is performed in an offline approach, an archiver is required that collects the runtime information and stores it in a file for later processing.

The Application Control Interface which controls the execution of the monitored application.

The Publication Interface which copes with the diversity of underlying hardware and software sensors available for monitoring. This interface publishes at runtime what information can be obtained from the monitoring system and in which form or aggregation.

The overall design of the application monitor and its interaction with performance analysis tools is illustrated in Figure 1. Performance analysis tools can submit information requests to the application monitor via the Monitoring Request Interface of the information producer. The producer configures the sensors, aggregates their data and delivers the information to the analysis tool.

Because the configuration of monitoring resources happens online and there are requests and result types for almost any known performance data type, MRI can basically be used by any kind of performance analysis tool. This no matter whether the tool performs postmortem or online analysis and no matter what the nature of the tool is. We

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

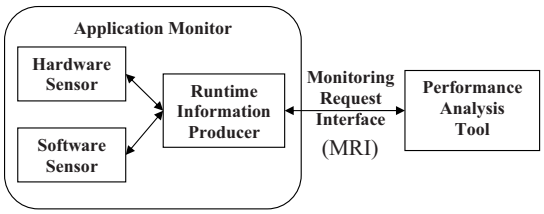

Figure 1. Overall design of the monitoring infrastructure.

think of MRI as the first step toward a standard interface used by a broad range of tools starting from simple profilers, trace generators and visualizers, GUIs up to complex automatic performance analyzers.

We implemented MRI for the EPC Monitor[3], an environment for monitoring cache hierarchies in SMP systems. In the context of EP-Cache[1] project[7] we also implemented performance tools which use MRI. The tools include a trace archiver which produces traces in Vampir-Trace-Format (VTF3), a GUI[8] and an automatic analysis tool called AMEBA[2].

The rest of the paper is organized as follows. Section 2 shows the terms and definition used in MRI. Sections 3, 4, 5 and 6 explain respectively the monitoring, results delivery, application control and publishing interfaces. Finally the Section 7 provides some details in our implementation of MRI.

2 Terms and Definitions

MRI is a C interface for submitting monitoring requests in terms of performance events for specific code regions or data structures. When used in multiprocessor environments the requests should be also able to specify aggregations, threads, processes or compute nodes. This section explains some of the terms and definitions we use in the rest of this paper as well as their C representations in MRI.

Runtime information is any information gathered during the execution of an application. Some monitoring systems use the terms event or metric. These can be events such as cache misses, CPU cycles, etc. or software sensor events such as execution time. MRI representation for this term is an enumerator called `MRI_Runtime_Information_Spec`.

A *(code) region* is a single entry block of statements. We interchangeably use the terms region and code region in the rest of this document pointing to the same concept. Examples of regions are:

Program units such as procedures, functions, and the main program.

Other sequential regions such as sequential loops and vector statements.

Parallel regions such as parallel loops, parallel sections, and master regions.

`MRI_Region_Spec` is the MRI representation of a code region. The structure includes two elements: `MRI_Region_Type` identifying whether the region is a sequential loop, routine, etc. and `MRI_Region_ID` identifying the region's location in the source code in terms of a `file_ID`, an unique id for a source file, and a `line_NR`, the region's first line number in the source file.

`MRI_Data_Structure_Spec` specifies the data structures to be monitored. A similar region id as for code regions is used to identify the data structure. This id however does not indicate the line number where the data structure is defined or used in the code. Instead the id indicates the program unit where the data structure is declared. This information together with the data structure's name (also included in the `MRI_Data_Structure_Spec`), uniquely identifies the data structure.

With the C data structures explained above here, it is possible to specify measurement requests for code regions and data structures. But MRI also supports requests for multiple processor environments such as shared memory or distributed systems. The performance tools using MRI should be able to request information for a single thread or process as well as aggregated information such as for example, *Sum over all threads* or *Minimum over the threads*. The following definitions serve to achieve this goal.

An *active object* is an entity performing some computation. We also use the term *region instance* by which we mean the execution of a region by an active object.

Active objects are represented in MRI by a structure called `MRI_Active_Object_Spec`. This structure includes the active object's type which can be a thread, process or computing node. It also includes an id for specifying the thread id (if the active object is a thread), the process id (if the active object is a process), and so on.

An *aggregation* combines runtime information from multiple instances of a region. These might be region instances in the same active object or in different active objects of the same kind. `MRI_Aggregation_Spec` defines the aggregation in MRI. Aggregations have an operation and a target. Operation indicates how to aggregate the runtime information. The target specifies the kind of the active object where the aggregation takes place. Examples are:

"SUM" and "PER_THREAD" specifies the sum of a runtime information for all executed instances of a region for a specific thread.

"MAX" and "PER_PROCESS" specifies the maximum of a runtime information achieved in any executed in-

[1]The work presented in this paper is mainly performed in the context of the EP-Cache project, funded by the German Federal Ministry of Education and Research (BMBF), and the APART working group

stance of a region in all threads of a process.

"MAX" and "PER_THREAD" specifies the maximum of a runtime information achieved in any instance of a region for a single specified thread.

Aggregation and active object are complementary. This means that if "PER_THREAD" is specified as aggregation target for example, MRI expects the specified active object to be a thread.

3 Specification and Management of Monitoring Requests

MRI provides routines for submitting and deleting monitoring requests. The communication mode is asynchronous. Immediately after a request submission, the control is returned to the performance tool which is using MRI. No action is undertaken from the MRI at this time. The monitoring only starts after one of the application control routines (see Section 5) is called. The resulting information, partial or complete depending on the state of the application's execution, is available via MRI as long as the request exists. Once the request is deleted via `MRI_Request_Delete()`, the resulting information is no longer accessible for the request.

The data structures introduced in section 2 are used as part of the parameters for the MRI requests. We say "part" because the requests accept tables of regions, active objects, data structures etc. Here is the definition of such a table for `MRI_Region_Spec`:

```
typedef struct {
    MRI_Region_Spec *region;
    int nr_Elements;
} MRI_Region_Table;
```

The other tables are similar to this one. The concept of tables of objects is useful for specifying more than one active object or code region per request.

In MRI there are four types of requests: aggregated, histogram, trace, and profile requests.

3.1 Aggregated Requests

```
MRI_Request_ID MRI_Aggregate_Request(
    MRI_Runtime_Information_Spec *ri,
    MRI_Region_Table *regions,
    MRI_Active_Object_Table *ao
    MRI_Data_Structure_Table *variables,
    MRI_Aggregation_Spec *aggregation)
```

This routine defines an aggregation request. The tool specifies one or more code regions, none or one data structure, one or more active objects, as well as an aggregation. Whether a data structure can be specified or not, depends on the requested runtime information. For example, if the execution time is specified as runtime information, it doesn't make sense specifying a variable. On the other hand, if the number of level one cache misses is requested, specifying a data structure restricts the cache miss count to the virtual address of the specified data structure. The routine returns `MRI_Request_ID`, an identification number for the request. This is used later to retrieve the results of the request or to delete the request.

The information is aggregated according to the aggregation and active object parameters. If multiple regions are specified, the information is additionally aggregated over all specified regions.

The number of measurements for a code regions involving hardware counters is limited by the actual number of available counters. If more requests are added than counters are available, the request will return `MRI_Error`. This does not apply for runtime information such as `EXECUTION_TIME` or `NUMBER_OF_INSTANCES` which doesn't require any hardware counters.

3.2 Histogram Requests

```
MRI_Request_ID MRI_Histogram_Request(
    MRI_Runtime_Information_Spec *ri,
    MRI_Region_Table *regions,
    MRI_Active_Object_Table *ao,
    MRI_Data_Structure_Table *variables,
    int *histogram_elements_nr)
```

The histogram request is especially designed for future hardware monitors or monitoring systems which provide detailed information for example about the access behavior of a code region in the memory occupied by a data structure. The EP-Cache hardware monitor[7], for example, can deliver this information. An MRI *histogram* contains aggregated values for a subrange of the requested address range. Each entry specifies a value plus a list of the data structure mapped to this subrange. Figure 2 depicts an histogram with 25 bins about the level one cache read misses for a data structure of a simple program.

As it can be seen from the request's signatures, the first four parameters are the same for aggregated and histogram requests. For the sake of conformity we designed it that way for all the MRI requests. While the first four parameter specify what is to be measured and exactly where, the next parameters hold information about the special kind of request.

`histogram_elements_nr` for instance specifies the number of bins for the requested histogram. Assigning a meaningful value to this parameter helps to understand and analyze the histogram. Consider again Figure 2. If the data structure represented in the histogram is an array of 25 x

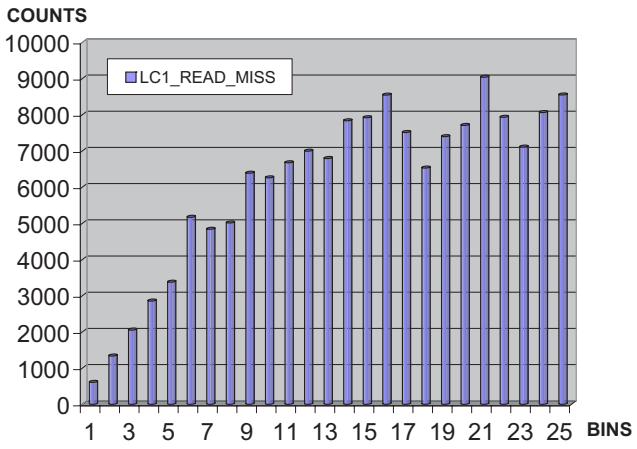

Figure 2. The histogram of level 1 cache read misses for a two dimensional array (100x100) in *gauss*, a program for solving linear equations, $Ax = b$, using Gauss elimination without pivoting.

25 elements, then each bin of the histogram represents the access behavior of an array's row (or column depending on the programming language and compiler).

3.3 Trace Requests

```
MRI_Request_ID MRI_Trace_Request(
    MRI_Runtime_Information_Spec *ri,
    MRI_Region_Table *regions,
    MRI_Active_Object_Table *ao,
    MRI_Data_Structure_Table *variables)
```

Collecting, analyzing, and visualizing trace information of an application is one of the most common ways of performing performance analysis today. As we committed to build a measurement interface that would provide support for almost any monitoring system, we had to also support trace information. Therefore we provided the `MRI_Trace_Request`. As it can be noted from the request's signature, only the 4 parameters specifying what to measure and where to measure it are available. For each instance of the specified regions as well as their subregions and for each specified active object a trace record will be generated which contains the measured metric's value, e.g., for each instance of region you get a trace record with the number of L1 cache misses.

We also defined a special runtime information for exclusive use with trace requests. This is called `MRI_REGION_EVENT`. A trace request with this runtime information will trigger the generation of trace records each time a region is entered or exited. This is necessary for visualizing a timeline of the application for example with Vampir[10].

3.4 Profile Requests

```
MRI_Request_ID MRI_Profile_Request(
    MRI_Runtime_Information_Spec *ri,
    MRI_Region_Table *regions,
    MRI_Active_Object_Table *ao,
    MRI_Data_Structure_Table *variables)
```

This request provides profile information about the application. That is for example a time profile or the profile of L1 cache misses for the executed regions. The information per region is summed over the specified active objects. If no active object was specified then every executed instance of the regions is counted. This request generates another kind of histogram, but this time the runtime information is distributed between the regions. For each region there is one bin in the histogram.

Figure 3 shows the visualized profile information from a MRI profile request. The required runtime information is `MRI_Execution_Time`. The left window of Kcachegrind[1] shows the list of regions together with the execution time in percentage. The profile information can be combined with the application's static information to produce a call tree as shown in the right window of Kcachegrind.

4 Data Retrieval Interface

After the submission of the measurement requests by using the request functions and after the application or a part of it was executed, the tool using MRI should be able to ask whether the performance results are ready and get the results for further analysis.

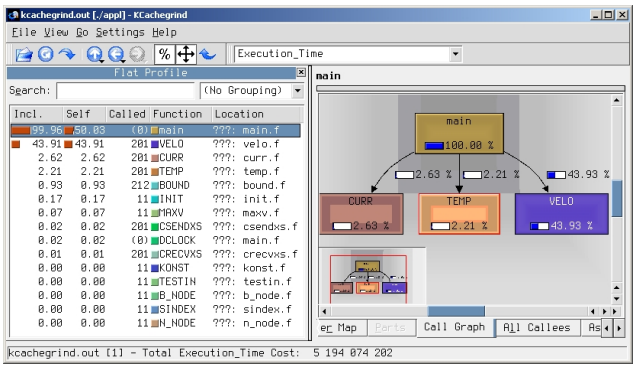

Figure 3. Profile information produced by MRI and visualized with Kcachegrind[1]

The delivery interface includes two synchronization routines, one of them is blocking and the other one a non blocking routine. `MRI_Wait()` as the name suggests, block the execution of the performance tool until either the results for the submitted requests are there or the application finishes its execution. The second routine, `MRI_No_Wait()` returns the status of the monitored application. The status tells whether the application is in execution, terminated, or stopped waiting for the results to be retrieved and maybe for new measurement requests.

The performance data for a request can be retrieved through the routine `MRI_Get_Data()`. This routine return results for aggregation, histogram and for profile requests based on the id of the request. It is up to the performance tool to cast the chunk of memory delivered by the routine to the proper result according to the request's kind. MRI specifies a result type for each kind of request.

4.1 Aggregation request results

`MRI_Aggregate_Request_Results` defines the result type for the aggregate request. The result contains the number of counted events and an eventual active object which is called *focus active object*. This is optional for aggregations that implies a minimum or maximum value. If for example the minimum of L1 cache misses over threads was requested, the focus active object indicates the thread where the minimum was achieved.

4.2 Histogram request results

`MRI_Histogram_Request_Results` is the result type for the histogram request. Each histogram bin contains the number of counted events together with a list of data structures. If a data structure was specified in the histogram request, each bin of the histogram will contain the same name of the specified data structure. In this case each bin correspond to a part of the data structure. If no data structure was specified in the request, then the whole address space of the application is monitored. In this case every data structure for which runtime information was counted appears in the histogram. Depending on the size of the data structure, it can happen that there are several bins of the histogram corresponding to one (large) data structure as well as multiple data structures for a single bin.

4.3 Profile request results

`MRI_Histogram_Profile_Results` is the type for the profile request. This is a table of all the monitored code regions together with the gathered runtime information for each region.

4.4 Trace request results

The results delivery routines shown at the beginning of this section are not available for the trace requests. While we can calculate the amount of information delivered by the aggregate, histogram, or profile request prior to results delivery, this is impossible for trace requests. We don't know how many instances of a region will be executed in one thread as we don't know how many events will be measured in a monitoring session. To cope with this dynamically growing amount of data, the data delivery interface provides a special mechanism. The monitored data is stored in a buffer in the form of trace records. Each record contains among other things the runtime information, a code region where the event was measured together with an eventual data structure, an active object, and a timestamp.

```
typedef struct MRI_Trace_Element_Spec{
  MRI_Runtime_Information_Spec event_type;
  MRI_Region_Spec region;
  MRI_Active_Object_Spec active_object;
  MRI_Data_Structure_Spec data_structure;
  unsigned long long timestamp;
  unsigned long long data;
} MRI_Trace_Element_Spec;
```

To retrieve the data, the tool using MRI, has to first register a call back function using the MRI routine `MRI_Register_Call_Back_Function()`. During the monitoring process, if the trace buffer is full, the registered call back function will be called and the tool can retrieve the data. The application execution stops until the buffer is again empty. The same procedure is used to retrieve the information at the end of the monitoring process.

There are two routines providing the necessary functionality for emptying the trace buffer. The first routine, `MRI_Get_First_Trace_Record()` returns the trace record at the top of the buffer. The second routine, `MRI_Get_Next_Trace_Record()` incrementally delivers the next record until the trace buffer is empty.

5 Application Control

The control over the execution of the monitored application is an important part of MRI. The application and the analysis tool using MRI synchronize with each other at the beginning of the execution. The tool poses its demands of monitoring in form of MRI requests and determines whether the application will start running until a predefined breakpoint or until a stop command is given. MRI provides two routines for running the stopped application.

One of them is `MRI_Start()`. The consequence of its call is for the application and the monitoring system to

continue execution until the application terminates or the MRI_Stop() routine is called from the tool. The second start routine poses a breakpoint to the application's execution. MRI_Start_Stop() takes as a parameter a normal MRI_Region. Once the end of this region is reached, the application stops and waits for further commands from the tool. We emphasize that only serial code regions can be used as breakpoints. This way we are sure that all active objects stop their execution after a breakpoint.

MRI also provide the means to load and finish the application. The respective routines are MRI_Application_Load() and MRI_Application_Finish(). This functionality is used if we want to restart the application several times while not changing the measurement requests. Such a situation can occur in case of a saturation of hardware counters for example. Instead of making sure that no more requests are submitted as the monitoring resources can handle at a time, the tool can just rerun the application until all the request are honored.

Part of the application control are also the initialization and finalization of the interface. MRI_Init() and MRI_Finalize() make sure that the tool waits for the application to load at the beginning of the monitoring process and that the application is also finished when the tool finish monitoring.

6 Publication Interface

We thought of MRI as an interface that should work with a large variety of monitoring resources in different computers. A performance tool prior to using MRI to submit measurement requests in a specific system, usually need to exactly know what resources the system offers in the sense of hardware and software sensors and what is provided by the monitoring infrastructure in terms of runtime information and aggregations. We supplied the publication interface as part of MRI to provide the tool with this kind of information. The following is provided by the publication interface:

6.1 Available hardware resources

This is provided in the form of: number of CPU(s), memory hierarchy, number of available hardware counters, etc. For trace requests including a timestamp or for requests evolving time measurements is especially important to know whether the times supplied by the monitor are expressed in seconds, microseconds or nanoseconds. This is specified by the MRI_Time_Entity member of hardware information.

MRI_Get_System_Information() from publication interface returns information about the architecture of the machine and the monitoring system. This is provided in form of a structure called MRI_HW_Info including detailed info about CPU, memory hierarchy, available hardware counters etc.

6.2 Available runtime information

This is a list of metrics than are provided by the hardware and software sensors of the monitoring system. Dependent on the design and ability of hardware counters available in the system this could vary from a couple of events to more than a hundred of them.

Get_Available_Runtime_Information() returns a table of runtime information available by the monitoring system.

6.3 Available active objects

This shows in which computational resources the application will execute, i.e., the number of threads, processes, compute nodes, etc. The aimed systems can vary from single processor workstations to SMP nodes, to thousand processor machines.

Get_Available_Active_Objects() returns the list of active objects which will execute the application.

6.4 Available aggregations

The available aggregations depend on the monitoring system. Usually they are specific to the required runtime information. As MRI would likely be used by a tool rather than directly by a user, it is important that we provide this quite intuitive information.

For example it does make sense to aggregate the runtime information MRI_LC1_DATA_READ_MISS as a minimum over threads, meaning that the thread is required where the minimum of level 1 cache read misses occur and how much this minimum is. On the other side it doesn't make any sense to aggregate the runtime information MRI_REGION_EVENT which indicates the start of a region and is only used in trace requests.

Get_Available_Aggregations() provides a list of available aggregations for a given runtime information.

7 Implementation of EPC Monitoring System

We built a monitoring system called EPC which implements the Monitoring Request Interface. EPC was first designed for a new hardware monitor developed in TU-München[7]. The counters on our hardware monitor can be configured to observe the whole or only a precise address range of the memory. We built a simulator called

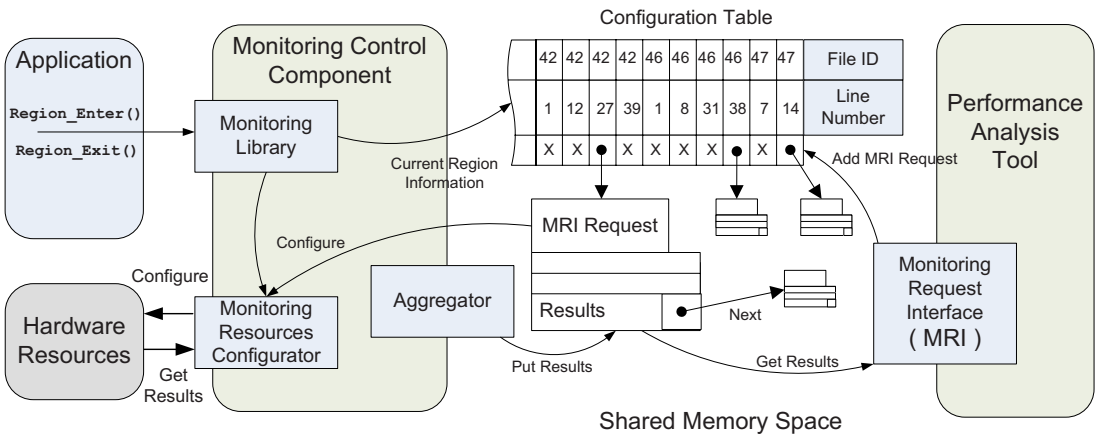

Figure 4. A more detailed view of EPC monitoring system revealing some internal functionality.

SMART[9] to simulate SMP nodes with our hardware monitor integrated in each of the node's processors.

We also implemented a PAPI[6] backend for EPC which allows us to perform measurements using the hardware counters available in architectures that are supported by PAPI. Recently we developed an Itanium specific implementation thus being able to monitor data structures in the program using the Itanium's hardware counters.

EPC supports Fortran 95 OpenMP programs. The monitoring environment requires source code instrumentation to insert monitoring library calls in the code. We use a Fortran 95 instrumenter[5] based on NAGWare f95 compiler front-end to instrument the code regions and data structures. The instrumenter also generates static information about the application such as which code regions or data structures were instrumented.

Our monitoring system is structured into two processes, the application process and the performance tool process. These processes communicate via shared memory segments of System V's Inter Process Communication (IPC). Accordingly, the system is implemented as two libraries. Figure 4 reveals the whole EPC system with some implementation details and functionality.

The first process is the performance tool (at the right in Figure 4) linked with the library which implements the Monitoring Request Interface. This component creates a Configuration Table in shared memory starting from code region entries in the application's static information. It also starts and terminates the monitored application. MRI requests issued by the tool are saved as shared memory segments and attached to the proper entry of Configuration Table according to the code region specified in the request.

The second process is the monitored application linked with the Monitoring Library, and the back-end which can be either the simulator or PAPI. The Monitoring Control Component (MCC) is the implementation of the monitoring library. The monitoring library passes the control to MCC at each region enter and exit which at its turn looks at whether there are appended MRI requests in the Configuration Table. If this is the case, the Monitoring Resources Configurator breaks down the MRI Requests to simpler PAPI or simulator Events and uses the Resources Configurator to configure the simulator or the hardware monitor.

Once the monitoring is finished, the results are either saved in the trace buffer, or if the MRI Request is an aggregated request, the Aggregator component aggregates the results and save them in the shared memory space.

7.1 Monitoring Scenario

Figure 5 depicts the interaction between the performance analysis tool, the application and the monitoring system. This helps the understanding of how the monitoring with EPC works.

The very first statement executed by the application after entering the main region is an instrumented call that initializes the EPC monitor. The application is blocked during initialization until the analysis tool specifies MRI requests and releases the application. The EPC initializes all monitoring sensors and then signals the tool that the application is ready for monitoring.

Once the tool finishes generating its initial MRI requests[2], it specifies a breakpoint where the program should stop (usually the end of a region) and the EPC is notified to start the application's execution.

At the entry and exit points of each instrumented region, the region instrumenter inserted a call to the monitoring li-

[2]We say initial because the tool can make other requests at any time. The synchronization between EPC and the tool allows interruption of the program for getting partial results or for making new requests. This is particulary useful if the tool can make new decisions (followed by new requests) based on the existing results gathered until the present state of application's execution.

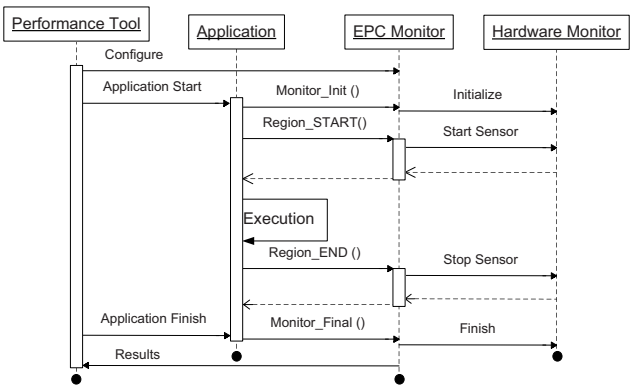

Figure 5. Monitoring Scenario - UML Sequence diagram.

brary. When the control flow enters the library, EPC looks up the configuration table for an MRI request that is appended to the current region. If such a request exists, symbolic data structure information will be translated into virtual addresses if necessary and the Hardware monitor will be configured accordingly. After that, the control is returned to the application.

At the end of the monitored region, EPC stops the hardware monitor, retrieves the results, aggregates them if required, and transfers the results to the space reserved for them in shared memory. When the end of a region corresponds with the specified monitoring breakpoint, the tool is notified to retrieve the results by leaving the application blocked.

There are two[3] possibilities for terminating the monitoring. One is that the application terminates. As showed in Figure 5 the application calls `Monitor_Final()`. This call is instrumented in all exit points of the application. After wrapping up all ongoing activities, EPC signalizes the tool to retrieve the final results and terminates the application.

Another possibility is that the tool decides to terminate the application. Another synchronization event between the application and the tool takes place in this case and the tool sets a termination flag in the EPC. On the next region enter or exit, the EPC realizes that the termination flag is set and calls `Monitor_Final()`. Everything continues afterwards as in the first case.

8 Summary and Outlook

MRI is the first step toward a standard monitoring interface. We think such an interface can improve our ability to build better performance tools. By allowing online and offline monitoring, MRI is flexible enough to provide all the performance data needed by almost any existing performance tools. Because it is transparent, MRI can help the performance tool developers to concentrate their efforts on the quality of the analysis and optimization rather than on how to obtain the performance data. MRI also support monitoring in multiprocessor environments being thus available in basically any architecture.

MRI is an ongoing work. We are currently working on extending the runtime information offered by MRI in order to support more monitoring resources and we are also studying possible extensions or changes in the interface for an improved support of distributed systems and the Grid.

[3]For the sake of simplicity both cases of monitoring termination are merged together in the Figure 5

References

[1] KCachegrind - Profiling Visualization), 2005. http://sourceforge.net/projects/kcachegrind/.

[2] Edmond Kereku, Michael Gerndt. The EP-Cache Automatic Monitoring System. In *Proceedings of Parallel and Distributed Computing and Systems, Phoenix AZ*, pages 39–44, November 2005.

[3] Edmond Kereku, Tianchao Li, Michael Gerndt, Josef Weidendorfer. A Data Structure Oriented Monitoring Environment for Fortran OpenMP Programs. In *Proceedings of Euro-Paar 04, Pisa*, pages 133–140, 2004.

[4] Intel Corporation. *Intel® Itanium™ Processor Reference Manual for software Development*. Intel Press, 2002.

[5] Michael Gerndt, Edmond Kereku. Selective Instrumentation and Monitoring. In *Proceedings of 11th Workshop on Compilers for Parallel Computers (CPC 04)*, pages 61–74. Shaker Verlag, 2004.

[6] Shirley Browne and Jack Dongarra and Nathan Garner and George Ho and Phil Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[7] T. Brandes, H. Schwamborn, M. Gerndt, J. Jeitner, E. Kereku, W. Karl, M. Schulz, J. Tao, H. Brunst, W.E. Nagel, R. Neumann, R. Mller-Pfefferkorn, B. Trenkler, H.-C. Hoppe. Monitoring Cache Behavior on Parallel SMP Architectures and Related Programming Tools. *Future Generation Computer Systems*, 20, 2005.

[8] Tianchao Li, Michael Gerndt. Cockpit: An Extensible GUI Platform for Performance Tools. In *Proceedings of Euro-Paar 05*, 2005.

[9] Tianchao Li, Michael Gerndt. SMART: A Simulation Tool for Monitoring Cache Access Behavior on SMPs. In *IEEE Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2005.

[10] W. E. Nagel and A. Arnold and M. Weber and H. C. Hoppe and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12(1):69–80, 1996.