Automatic Monitoring of Memory Hierarchies in Threaded Applications with AMEBA*

Edmond Kereku, Michael Gerndt

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

Abstract. In this paper we present an approach to online automatic monitoring of memory hierarchies in threaded applications. Our environment consists of a monitoring system and an automatic performance analysis tool. The EPC monitoring system, uses static instrumentation of the source code and information from the hardware counters to generate performance data for selected code regions and data structures. The monitor supports threaded applications by providing per-thread performance data or by aggregating it. It also provides a monitoring requests API for the performance tools. Our tool AMEBA performs an online automatic search for cache and thread-related ASL properties in the code.

1 Introduction

With SMP architectures being more and more a commodity computing resource and with paradigms like OpenMP, the programming of threaded applications became a task that almost any average programmer can master. A great number of HPC applications originated this way. But this is where the easy part ends. While debugging of such an application is not as easy as writing it, obtaining performance and scalability from it could really be a daunting task.

The memory bottleneck problems for example, already present in serial programs, obtain a greater significance in threaded applications. False sharing and data locality are added to the traditional problems such as high cache miss rate. Usual profiling and tracing tools can't manage anymore to find the new problems, what is needed is more extensive and complex monitoring and analysis.

AMEBA[1] (Automatic Monitoring Environment for Bottleneck Analysis) is our approach to automatic analysis of threaded applications in SMPs. By using the EPC[2] monitoring environment, our tool is able to automatically search for complex cache problems (expressed in ASL[3]) in serial and OpenMP regions. The search can even be refined to single data structures in the region. EPC can either use simulation or the available hardware counters in the processors to provide the required performance data. Particularly interesting for monitoring of threads and data structures is the port of EPC to Itanium-based SMPs and especially in ccNUMA architectures, such as SGI Altix 3700[4].

* This work is performed in the context of the projects EP-CACHE and Periscope, funded by the German Federal Ministry of Education and Research(BMBF)

The rest of this document is organized as follows: Section 2 reveals the support of EPC for threaded applications. Section 3 is an overview of AMEBA and Section 4 introduces new ASL properties related to threads and cache problems.

2 Monitoring threads with the EPC Monitor

The EPC environment support for threaded applications includes the instrumentation of OpenMP regions, the thread-specific configuration of monitoring resources in runtime, and the ability to deliver thread-related performance data.

Our Fortran 90 instrumenter[5] called *f90inst* instruments sequential regions, taking into account multiple exits from regions, as well as OpenMP constructs based on the work done by Mohr et al. in POMP[6].

Performance tools such as AMEBA access EPC through a well defined monitoring API called the *Monitoring Request Interface* (MRI)[7]. MRI allows the tool to specify monitoring requests in terms of *Runtime Information* (what to measure), code regions, and *Active Objects* which can be threads, nodes etc.

In order to configure single threads upon an MRI Request, EPC has a thread management utility for storing thread-related information. With its help the monitor knows how many threads are used in the monitored application, and which thread executed a call to the monitoring library.

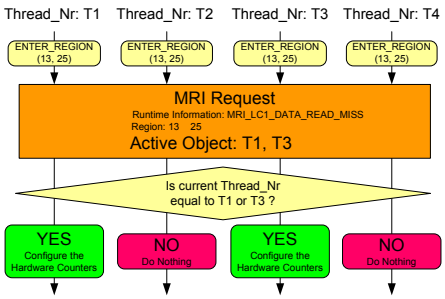

Fig. 1. Configuration of monitoring for multi-threaded applications

Figure 1 shows how the configuration for thread-related monitoring works. After a thread entered the monitoring library and a monitoring request is pending for the current region, first EPC determines which thread has entered the monitoring library. If the thread id matches one of the threads specified in the MRI Request, the hardware resources are accordingly configured, otherwise the application immediately continues the execution.

3 The AMEBA automatic performance analysis tool

Our analyzer AMEBA performs an automated iterative search for performance problems specified with ASL which is executed while the application is run-

ning. The search process is iterative in the sense that AMEBA starts with a set of potential performance properties, performs an experiment, evaluates the hypotheses based on the measured data, and then refines the hypotheses. The refinement can either be towards more specific performance properties or towards subregions and data structures of the already tested region.

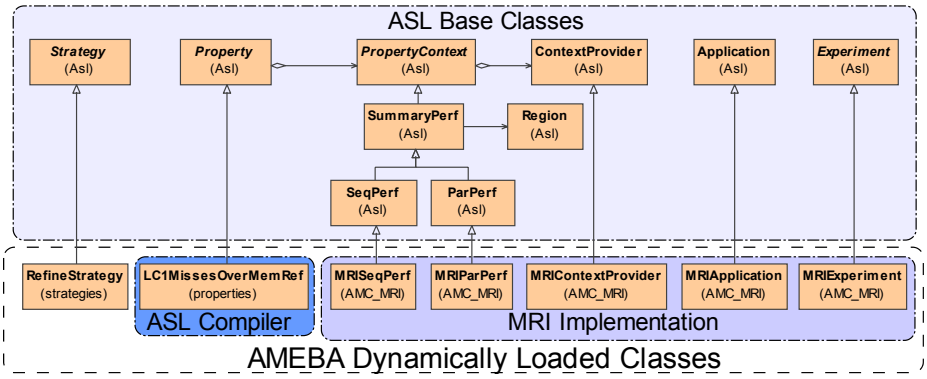

Fig. 2. AMEBA Classes derived from the ASL data model

AMEBA is built over the model shown in Figure 2. This model includes a set of fixed base classes which are implemented in the tool and a set of derivative classes which are implemented separately for a specific monitoring environment, hardware system or application domain. Those classes are dynamically loaded at runtime, which makes AMEBA a very versatile and extensible tool.

The base classes includes the *Strategy* which specifies the search process, the *Property* which describes the performance problems, the *PropertyContext* which holds the performance data needed to evaluate the property, and the *ContextProvider* which sets the connection with the monitoring system. The classes derived from the PropertyContext contain summaries of the performance data specific to sequential (*SeqPerf*) or parallel (*ParPerf*) regions, and specific to the monitoring environment (*MRI(Seq/Par)Perf*).

4 Threads and Cache-related ASL Properties

As stated before, AMEBA uses the APART Specification Language (ASL) to describe the performance problems[1] in terms of condition, confidence, and severity. Consider for example the following ASL Property:

[1] Actually in the original ASL specification, a property constitutes a performance problem if its severity exceeds a given threshold. In AMEBA we provide the threshold as parameter to the property, therefore we interchangeably use the terms property and problem when referring to ASL Properties.

```
property UnbalancedLC1DataMissRateInThreads(ParPerf pp, float t){
 let
  miss_rate(int tid)=pp.lc1_miss[tid]/pp.mem_ref[tid];
  mean=sum(miss_rate(0),...,miss_rate(pp.nrThreads))/pp.nrThreads;
  max = max(miss_rate(0),...,miss_rate(pp.nrThreads));
  min = min(miss_rate(0),...,miss_rate(pp.nrThreads));
 in
  condition: max(max-mean, mean-min) > t;
  confidence: 1;
  severity: max(max-mean, mean-min) * pp.parT[0]; }
```

where *pp* is summary of performance data for a parallel region, *parT[0]* is the execution time of the master thread. This property specifies that there is a problem in parallel regions if the L1 cache miss rate in a thread deviates from the mean value achieved over the threads beyond a given percentage *t*. Furthermore it specifies that the problem is more *severe* if it is found in regions where most of the execution time is spent. We also are 100% *confident* about the existence of the problem in regions where the condition is true because our measurements are based in precise counter values and are not statistical values.

If a performance problem expressed by an ASL property is the refinement of another performance problem, we organize them in an hierarchical order. In AMEBA we implemented search strategies which take this organization into account. For example `UnbalancedLC1DataMissRateInThreads` is actually a refinement of the problem specified by `LC1DataMissRateInThreads`, which holds if the miss rate in one of the threads is higher than a threshold. Similar properties exist for the L2 and L3 caches and refinements are provided for write and read misses.

An interesting group of properties can be defined for Itanium-based ccNuma architectures such as SGI Altix 3700. By using the Itanium-specific events `DATA_EAR_CACHE_LAT4, LAT8, LAT16 .. LAT4096` we can identify data locality problems. One property of this group is `RemoteMemAccessInThreads`.

References

1. E. Kereku, M. Gerndt: The EP-Cache Automatic Monitoring System. (In: Proceedings of Parallel and Distributed Computing and Systems, Phoenix 05)
2. E. Kereku et al.: A Data Structure Oriented Monitoring Environment for Fortran OpenMP Programs. In: Proceedings of Euro-Paar 04, Pisa. (2004) 133–140
3. T. Fahringer et al.: Knowledge Specification for Automatic Performance Analysis. Technical report, APART Working Group (2001)
4. SGI Altix 3700's homepage: http://www.sgi.com/products/servers/altix/3000.
5. M. Gerndt, E. Kereku: Selective Instrumentation and Monitoring. (In: Proceedings of 11th Workshop on Compilers for Parallel Computers, 2004) 61–74
6. B. Mohr, A. Malony, S. Shende, F. Wolf: Design and Prototype of a Performance Tool Interface for OpenMP. Journal of Supercomputing **23** (2002) 105–128
7. E. Kereku, M. Gerndt: The Monitoring Request Interface (MRI). (In: Proceedings of IPDPS 06, Rhodos Island) to appear.