Modeling and Detecting Performance Problems for Distributed and Parallel Programs with JavaPSL *†

Thomas Fahringer and Clóvis Seragiotto Júnior

Institute for Software Science, University of Vienna
Liechtensteinstrasse 22, A-1090, Vienna, Austria
[tf,clovis]@par.univie.ac.at

Abstract

*In this paper we present JavaPSL, a **P**erformance **S**pecification **L**anguage that can be used for a systematic and portable specification of large classes of experiment-related data and performance properties for distributed and parallel programs. Performance properties are described in a generic and normalized way, thus interpretation and comparison of performance properties is largely alleviated. Moreover, JavaPSL provides meta-properties in order to describe new properties based on existing ones and to relate properties to each other.*

JavaPSL uses Java and its powerful mechanisms, in particular, polymorphism, abstract classes, and reflection to describe experiment-related data and performance properties. JavaPSL can also be considered as a performance information interface based on which sophisticated performance tools can be built or other tools can access performance data in a portable way.

We have implemented a prototype performance tool that uses JavaPSL to automatically detect performance bottlenecks for MPI, OpenMP, and mixed OpenMP and MPI programs. Several experiments with realistic codes demonstrate the usefulness of JavaPSL.

Keywords: Performance analysis, performance specification language, distributed and parallel systems, mixed OpenMP and MPI programs

1 Introduction

Performance tuning commonly involves a cyclic and very ad-hoc process of measuring and analyzing performance data, identifying and possibly eliminating performance problems in slow progression. A large variety of performance tools have been created over the last decade, which includes performance instrumentation, measurement, profiling, and tracing tools. These tools commonly provide a vast amount of performance data, focus on specific program and machine behavior, do not relate performance information back to the input program, leave the interpretation of performance data to the user, or do not guide the programmer to essential performance problems.

Although there are several well-known formats to describe tracing and profiling information such as SDDF (Pablo Self-Describing Data Format) [3], ALOG [9] or Vampir Tracefile Format [17] there does not exist a generic way to describe performance problems, which is important to build comprehensive performance tools that can be more easily adjusted for new programming languages and target architectures. Most existing performance tools are limited to performance overheads (e.g. execution and communication times, cache misses, etc.) but do not provide higher level performance information such as performance properties (e.g. scaling behavior, load imbalance, etc.). Tools frequently hard-code performance information,

*This research is partially supported by the Austrian Science Fund as part of Aurora Project under contract SFBF1104, and the ESPRIT IV Working Group on Automatic Performance Analysis: Resources and Tools funded under Contract No. 29488

which is awkward to be reused by other tools or to be extended for novel programming paradigms and target machines. Additionally, performance information that is not normalized is difficult to interpret or to be compared against other performance data. For instance, absolute number of cache misses for a specific code region has very little meaning without being compared against cache hit counts for the same region.

In our opinion a generic specification language for experiment-related data (e.g. information about program versions, code regions, target machine, and profiling or tracing information) and performance properties can be used as a standard performance information interface to describe wide classes of performance problems for a large variety of programming languages and target architectures. Such a language can be used to build higher-level performance analysis technology that accesses experiment-related data or performance properties in order to compute, for instance, new performance properties. Finally, a generic specification language can also be used by other tools such as compilers or transformation tools to access performance information in a portable way.

In this article we describe JavaPSL, a generic performance specification language for modeling experiment-related data and performance properties of distributed and parallel programs. Performance properties characterize a specific negative performance behavior of a program and are defined over experiment-related data. Note that JavaPSL is not a tool, but intended to be used as a standard performance information interface that can be used to model a large variety of performance information, to build sophisticated performance tools (e.g. to provide automatic bottleneck analysis), and to enable portable access to performance information.

JavaPSL uses powerful Java mechanisms, in particular, polymorphism, abstract classes, and reflection, to describe performance properties. Moreover, JavaPSL provides meta-properties (defined as Java abstract classes) so as to describe new properties based on existing ones and to relate properties to each other. Performance properties are related to the code regions that cause them through experiment-related data. JavaPSL filters and statistics classes can be used to restrict performance analysis to specific experiment-related data, and to compute statistics based on arbitrary sets of performance values.

We will present several examples that show how to model performance properties of an application, including non-scalability, load imbalance, inefficiency, uneven work distribution, and various overheads such as synchronization, communication, unidentified, loss of parallelism, control of parallelism, late sender, and cache misses.

Figure 1 shows a design of a performance tool that tries to automatically find all performance bottlenecks of a program by using JavaPSL. The program files are input to the performance tool. An experiment decision system requests performance data from possibly several performance profiling/tracing/prediction tools, which is then stored by JavaPSL classes describing experiment-related data. Based on this data and pre-defined property specifications, a bottleneck analysis system computes a set of performance properties, which are stored together with experiment-related data as JavaPSL classes. A cyclic search process for performance bottlenecks is invoked, during which the bottleneck analysis system computes, examines and stores performance properties and initiates additional performance experiments through the experiment decision system. If all bottlenecks – specified in the performance property specification repository – are found or a time limit is reached the search is stopped and performance bottlenecks can be visualized or further examined. Arbitrary external performance tools can be used to provide experiment-related data. Moreover, not only performance tool builders but also the user should be given the possibility to add new properties and change or delete existing ones in the performance property specification repository. JavaPSL is used as a standard interface to represent and to provide portable access to experiment-related data and arbitrary performance properties.

This paper focuses on JavaPSL as a performance specification language and its flexibility to describe large classes of experiment-related data and performance properties. We have implemented a prototype performance tool that uses JavaPSL in order to detect performance bottlenecks for MPI, OpenMP and mixed OpenMP and MPI programs. Experiments with several realistic codes will be shown to demonstrate the usefulness of JavaPSL.

The organization of this article is as follows. Related work is presented in Section 2. The next section describes the execution model for which JavaPSL has been mostly used so far. Section 4 describes the experiment-related data as well as filters and statistics that can be applied to them. Performance properties are described in the following section, which includes simple and meta-properties. Several performance properties are examined for MPI and mixed OpenMP/MPI codes in Section 6. Conclusions and future work are discussed in Section 7.

2 Related Work

The use of specification languages in the context of automatic performance analysis tools is a relatively new approach.

Kappa-pi [1] employs a post-mortem performance analysis to search for performance bottlenecks based on trace files of PVM applications. Patterns are tried to be detected in trace files that match performance-problems patterns in a knowledge

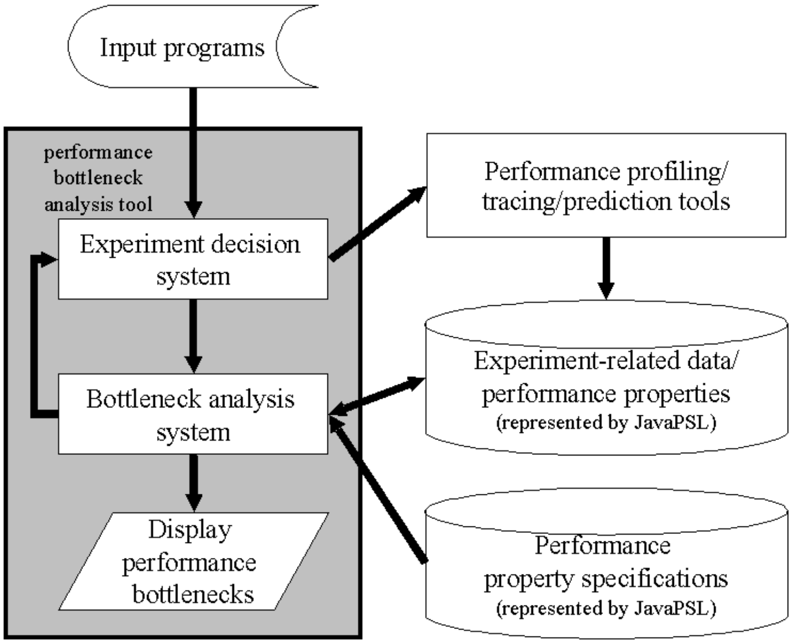

Figure 1. Design of a performance tool that automatically tries to find all performance bottlenecks by using JavaPSL

base which cannot be modified by the user.

Paradyn [14] performs an automatic online analysis and is based on dynamic monitoring of sequential or message-passing applications. Among its several components is the Performance Consultant, which automatically searches for performance problems *while* the application is running. Even though Paradyn is highly configurable (a configuration language is provided to request specific performance data), the Performance Consultant does not allow the user to modify existing or to define new performance problems to be searched for.

Finesse [16] directs the user to the overheads found in the application, and provides guidance for eliminating such overheads. Finesse focuses on shared memory programs, and its performance analyzer, which tries to find performance overhead, cannot be configured by the user.

Poirot [10] is the design of a software tool which does not depend on any specific programming environment. To build such a design, Poirot's authors gathered several performance tools with the goal of formalizing performance bottlenecks of a parallel program. The search algorithm in Poirot uses a database containing predefined performance bottlenecks to be searched for.

Autopilot [2] is a software infrastructure that can be used to build adaptive parallel and distributed software (that is, it is intended to analyze the data while the program is running). It provides a set of C++ classes to create sensors (pieces of software inserted in the application to monitor its behavior), actuators (components that change the application's behavior), and decision procedures (rules that govern the changes ordered by the actuators based on the data read from the sensors).

EARL [20] is a language designed to describe event patterns of message-passing programs based on trace files. Implemented as an extension of high-level script languages (currently Tcl, Perl and Python), Earl provides abstractions that hide the trace file details.

Our work on JavaPSL has been stimulated by ASL (Apart Specification Language [8]) developed in the APART Esprit IV working group. ASL is a new language designed to formalize experiment-related data and performance properties; it uses an object-oriented model to describe experiment-related data, and functions and constraints to specify performance properties. ASL has been extended several times, which resulted in a rather complex language. It is unclear how efficient ASL can be implemented, which has yet to be done. JavaPSL overcomes most complexity introduced by ASL through simpler and at the same time more generic mechanisms. For instance, ASL concepts like templates and meta-properties are expressed with abstract classes and reflection under JavaPSL; ASL global definitions and constructions like "FORALL...SUCH THAT" are handled with JavaPSL iterators and filters. JavaPSL can be understood as the first effort to implement the underlying ideas of ASL. JavaPSL also introduces various novel performance properties not included in ASL such as non-scalability, imperfect cache behavior, inefficiency, and overhead for any execution.

3 Execution Model

JavaPSL is a generic performance specification language for sequential, parallel and distributed programs. Figure 2 shows the execution model of a generic process with its own address space in a MIMD (multiple-instructions-multiple-data) or SMPD (single-program-multiple-data) program. The process executes a program subdivided into sequential and parallel regions.

A process may dynamically spawn/fork, synchronize, and terminate threads during execution of the program. All threads of the process share the same address space. In a sequential region only one thread within the process is active (executes the code region), while in a parallel region several threads may be active and execute simultaneously. The distribution of the computational load among active threads within a parallel region is language dependent. Threads may be spawned at the beginning of the program or a parallel region, which is also language/implementation dependent.

At the end of the parallel region the active threads may be synchronized (e.g. through a barrier synchronization or join operation); moreover, all but one (which continues to execute the following region) will either be terminated or turned passive (that is, do not execute a region but remain alive, possibly being turned active in a subsequent parallel region). Active threads within the same process exchange data by using the shared memory, while threads associated with different processes need to use generic SEND and RECV operations to exchange data. These operations can be executed in both sequential and parallel code regions.

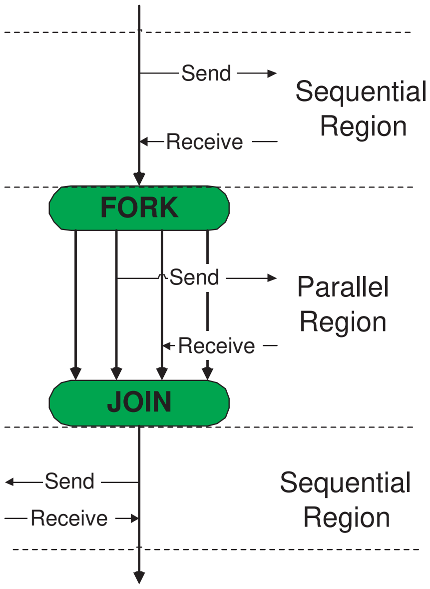

Figure 2. Execution model

4 Experiment-related Data, Filters, and Statistics

In the following we describe experiment-related data, which is used to define performance properties, and discuss the role of filters and statistics, which are important to restrict performance analysis to essential parts of experiments and to alleviate the specification of performance properties.

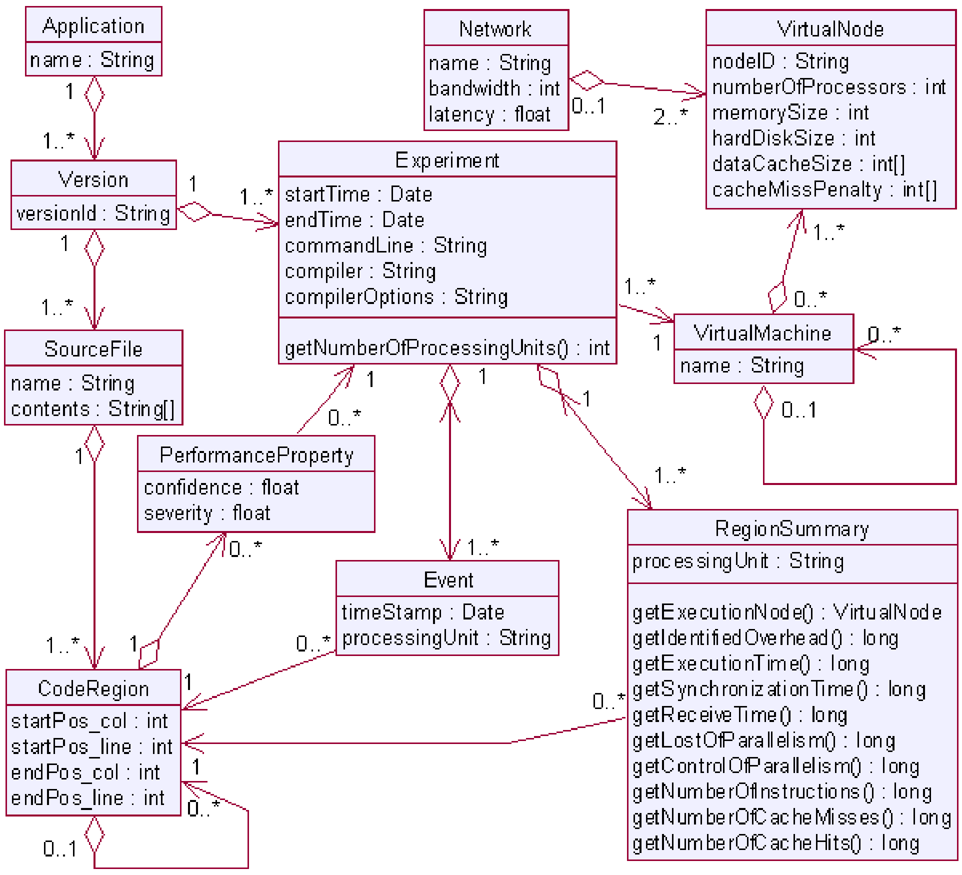

Figure 3. JavaPSL experiment-related data

4.1 Experiment-related Data

An *experiment* refers to a sequential or parallel execution of a program on a given target architecture. Every experiment is described by *experiment-related data*, which includes information about the application code, the part of a machine on which the code has been executed, and performance information. JavaPSL uses the syntax and semantic rules of the Java programming language in order to specify experiment-related data. Figure 3 visualizes the JavaPSL classes for experiment-related data by using the UML (Unified Modeling Language [18]).

An application (program) may have a number of implementations (code versions), each of them consisting of a set of source files and associated with one or several experiments. Every source file has one or several static code regions (ranging from the entire program to single statements), uniquely specified by *startPos* and *endPos* (positions where the region begins and ends in the source file).

Experiments are associated with the virtual machines on which they have been taken. The virtual machine is the part of a physical machine available to the experiment; it is described as a set of computational nodes (e.g. single-processor systems, Symetric Multiple Processor (SMP) nodes sharing a common memory, etc.) connected by a specific network.

A set of region summaries (*profile information*) describes performance information for every processing unit (process or thread) used by the experiment. Among others, region summaries currently cover execution, synchronization, communication, receive (time spent in blocking receive operations), and waiting time; loss of parallelism (time spent on replicated, unparallelized or partially unparallelized code regions); control of parallelism (time spent to control the parallelism, such as scheduling of loops).

JavaPSL also formalizes trace files by using an event class. This paper does not contain any property which incorporates events, which may change in future work.

Note that due to advanced monitoring and profiling technologies (e.g. dynamic profiling [14], hardware profiling [5], and source code profiling [19, 15]), there is basically no barrier to obtain experiment-related data for arbitrary parallel and distributed programs.

4.2 Filters and Statistics for Experiment-related Data

It order to provide flexible mechanisms to describe performance problems we commonly require filters (used to restrict the performance analysis to a subset of experiment-related data) and statistical methods (e.g. standard deviation). For this purpose we introduce a few new classes (see Figure 4), like *RegionSummaryIterator* and *Statistics*, as well as the interface *RegionSummaryFilter*. The method *regionSummaryIterator* of an Experiment instance can be used to obtain an object that iterates over all region summaries of this experiment. Optionally, a *filter* can be passed as a parameter to this method. In this case, the iterator object will return only the summaries satisfying the filter condition(s). Filters must implement the interface *RegionSummaryFilter*, whose only method verifies whether a region summary is accepted or rejected by the filter. For example, a filter that accepts only region summaries with non-zero message passing time can be encoded as follows:

```
class MyFilter implements RegionSummaryFilter {
    public boolean accept(RegionSummary rs) {
        return rs.getCommunicationTime() != 0;
    }
}
```

Therefore, in order to implement an iterator object that iterates over all region summaries with non-zero message passing time, we can use the following code:

```
iterator = experiment.regionSummaryIterator(new MyFilter());
```

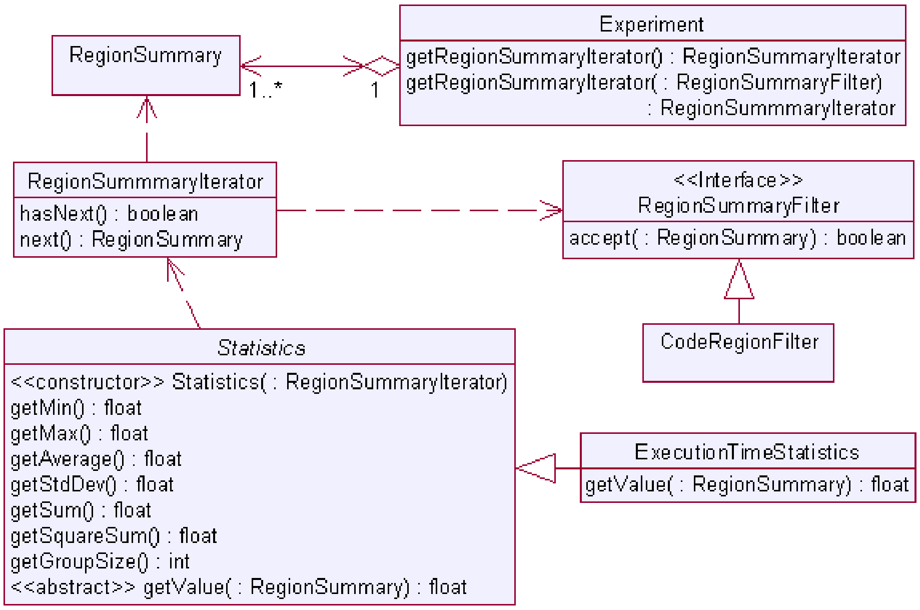

Figure 4. Statistics, iterator and filter classes

The abstract class *Statistics* provides statistical methods defined over specific performance information (e.g. communication time) of a region summary. The desired performance information must be provided by overriding the method *getValue* in a subclass of *Statistics*. Commonly used statistics and filters are predefined under JavaPSL, which includes a class *ExecutionTimeStatistics* that computes statistics over the execution time of a given iterator, and a filter *CodeRegionFilter* that accepts only summaries of a specific code region.

In what follows we present a brief example to demonstrate the usage of filters and statistics under JavaPSL:

```
1.  filter = new CodeRegionFilter(reg);
2.  iterator = experiment.regionSummaryIterator(filter);
3.  statistics = new ExecutionTimeStatistics(iterator);
4.  average = statistics.getAverage();
5.  stdDev = statistics.getStdDev();
```

Line 1 creates a filter that accepts only summaries of region *reg*, and, in line 2, a new iterator is created based on the given filter. Line 3 generates an object whose methods can be used to compute statistics over the set of values returned by the iterator. Lines 4 and 5 determine average and standard deviation for the given set of values.

5 Performance Property Specification

A performance property (e.g. load imbalance, synchronization overhead, etc.) characterizes a specific negative performance behavior of a program and is defined by three components:

- holds: boolean value that determines whether a property holds or not.
- confidence: normalized value between 0 and 1 that indicates the degree of confidence in the correctness of the value of *holds*. A confidence value 1 means that the value of holds is very likely to be correct. The closer the confidence value is to 0, the more uncertain the correctness of *holds* is.
- severity: normalized value between 0 and 1 that indicates the importance of the property. Severity value 0 means that the property has little importance whereas a severity value 1 may imply a detrimental effect on the overall performance.

JavaPSL uses syntax and semantic rules of the Java programming language in order to specify performance properties and experiment-related data. In the following sections we introduce the key concepts to specify performance properties.

5.1 The Interface *Property*

All JavaPSL performance properties implement the common interface *Property* (see Figure 5), which includes specific methods to express the hold, confidence and severity value of all properties. The value returned by *getSeverity* is undefined if the method *holds* returns *false*.

```
public interface Property {
    boolean holds();
    float getConfidence();
    float getSeverity();
}
```

5.2 Simple Properties

Many performance properties obey a generic pattern: they compute a severity value which, if greater than 0, indicates that the property holds; the confidence value is per default set to 1. In order to incorporate this generic pattern for performance properties we created an abstract class with the name *SimpleProperty*.

```
public abstract class SimpleProperty implements Property {
    protected float severity;
    public boolean holds()       { return severity > 0; }
    public float getConfidence() { return 1; }
    public float getSeverity()   { return severity; }
}
```

All current performance properties except meta-properties are defined as subclasses of *SimpleProperty*. In order to define a novel simple property, commonly only a constructor must be provided to compute a normalized severity value (between 0 and 1). In the following we describe and discuss several important simple properties.

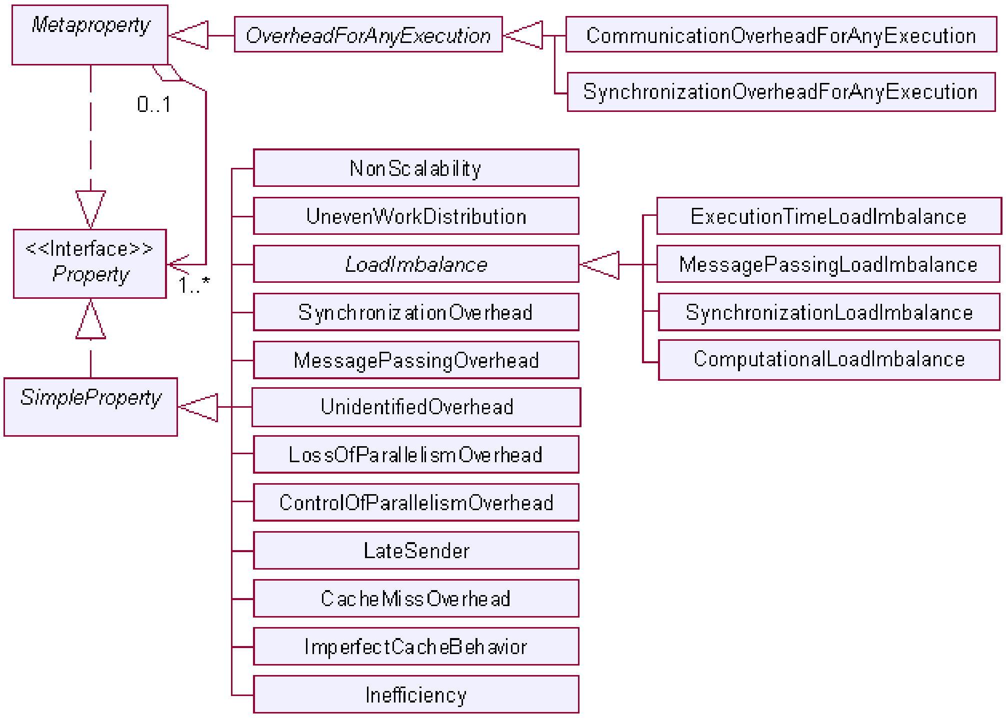

Figure 5. JavaPSL performance properties

5.2.1 Non-scalability

The scalability of a parallel application reflects the execution behavior for changing machine and problem sizes. Our definition for scalability assumes that, based on a given set of experiments, a code region scales if the efficiency is nearly the same for every experiment in the set [7].

We quantify “nearly the same” as the difference between the average and the minimum efficiency. Consequently, the property *NonScalability* can be defined as follows:

```
public class NonScalability extends SimpleProperty {

   public NonScalability(Experiment sequentialExp, Experiment[] parallelExps, CodeRegion r) {
       CodeRegionFilter f = new CodeRegionFilter(r);
       float seqTime = sequentialExp.summaryIterator(f).next().getExecutionTime();
       float minEfficiency = 1, sumEfficiencies = 0, avgEfficiency;

       for(int i = 0; i < parallelExps.length; i++) {
           Statistics st = new ExecutionTimeStatistics(parallelExps[i].summaryIterator(f));
           float parTime = st.getMax();
           float p = st.getGroupSize();
           float efficiency = seqTime / (parTime * p);
           if (efficiency > 1) efficiency = 1;
           minEfficiency = Math.min(minEfficiency, efficiency);
           sumEfficiencies += efficiency;
       }
       avgEfficiency = sumEfficiencies / parallelExps.length;
       severity = avgEfficiency - minEfficiency;
   }
}
```

The constructor of the property *NonScalability* takes as arguments the code region r which will be tested, a sequential experiment (from which the sequential execution time of r is extracted) and an array of parallel experiments (to get the parallel execution times of *r*). The efficiency is computed for each parallel experiment (lines 9 to 13), and the severity is defined by subtracting the minimum from the average efficiency (line 18).

Note that a program that scales well may still be very inefficient, which can be determined by the property *Inefficiency* (see section 5.2.2).

5.2.2 Inefficiency

Given a parallel experiment, the efficiency [13] of a code region is defined as $\frac{T_s}{q*T_p}$, where T_s is the sequential execution time, T_p the parallel execution time (maximum across all processing units), and q the number of processing units that execute the code region. As all performance properties reflect some negative performance behavior, we define an inefficiency performance property by the converse of efficiency as follows:

```
public class Inefficiency extends SimpleProperty {
    public Inefficiency(Experiment seqExp, Experiment parExp, CodeRegion r) {
        CodeRegionFilter f = new CodeRegionFilter(r);
        Statistics st = new ExecutionTimeStatistics(parExp.summaryIterator(f));
        float seqTime = seqExp.summaryIterator(f).next().getExecutionTime();
        float parTime = st.getMax();
        float ideal = seqTime / st.getGroupSize();
        severity = parTime <= ideal ? 0 : (1 - ideal / parTime);
    }
}
```

The severity of the inefficiency property varies between 0 (best-case efficiency for linear or super-linear speedup) and 1 (worst-case efficiency).

In the case where the inefficiency property holds and a high severity value is indicated, we can further explore its causes. The following sections present properties that explore some possible causes of inefficiency.

5.2.3 Overhead sources

For every individual overhead we specify a unique performance property, the severity of which is computed as the ratio of the corresponding performance value (e.g. synchronization or message passing time) and a reference value (for instance, the execution time of the region or the entire program).

In the remainder of this section and in the two following ones, we define several of these overhead performance properties: *CommunicationOverhead, SynchronizationOverhead*, *LossOfParallelismOverhead*, *ControlOfParallelismOverhead*, *UnidentifiedOverhead*, and *CacheMissOverhead*.

```
public class CommunicationOverhead extends SimpleProperty {
    public CommunicationOverhead(RegionSummary summary, RegionSummary rankBasis) {
         severity = summary.getCommunicationTime() / rankBasis.getExecutionTime();
    }
}

public class SynchronizationOverhead extends SimpleProperty {
    public SynchronizationOverhead(RegionSummary summary, RegionSummary rankBasis) {
        severity = summary.getSynchronizationTime() / rankBasis.getExecutionTime();
    }
}

public class LossOfParallelismOverhead extends SimpleProperty {
    public LossOfParallelismOverhead(RegionSummary summary, RegionSummary rankBasis) {
         severity = summary.getLossOfParallelism() / rankBasis.getExecutionTime();
    }
}

public class ControlOfParallelismOverhead extends SimpleProperty {
    public ControlOfParallelismOverhead(RegionSummary summary, RegionSummary rankBasis) {
        severity = summary.getControlOfParallelism() / rankBasis.getExecutionTime();
    }
}
```

A property *LateSender* can be defined to detect overhead at receiving processing units waiting for messages to arrive. It is often assumed that this overhead is caused by a sending processing unit that issued send operations too late.

```
public class LateSender extends SimpleProperty {
    public LateSender(RegionSummary rs, RegionSummary rankBasis) {
        severity = rs.getReceiveTime() / rankBasis.getExecutionTime();
    }
}
```

Note that the communication time used in the property *CommunicationOverhead* already includes the receiving time as used in the property *LateSender*. The property *LateSender* can therefore be used to explain the reason for the property *CommunicationOverhead*.

5.2.4 Unidentified overhead

The difference between parallel execution time of a region *r* and the corresponding ideal execution time defines the total overhead of a parallel execution of *r* and is given by $T_p(r) - \frac{T_s(r)}{q}$. Depending on the quality of the underlying performance analysis tools, a portion of the total overhead – the identified overhead – can be directly determined (e.g. by measurements). The difference between identified and total overhead is defined as unidentified overhead, for which we define a separate property.

```
public class UnidentifiedOverhead extends SimpleProperty {
    public UnidentifiedOverhead(RegionSummary seqSummary, RegionSummary parSummary,
                                RegionSummary rankBasis) {
        float seqTime = seqSummary.getExecutionTime();
        float parTime = parSummary.getExecutionTime();
        int q = parSummary.getExperiment().getNumberOfProcessingUnits();
        float unidentifiedOverhead = (parTime - seqTime / q) -
                                     parSummary.getIdentifiedOverhead();
        severity = unidentifiedOverhead / rankBasis.getExecutionTime();
    }
}
```

5.2.5 Cache Misses and Imperfect Cache Behavior

In order to compute the overhead of level-i cache misses, we multiply the number of cache misses by the time penalty for one cache miss, which is commonly described in vendor hardware manuals (for instance, a level-2 cache miss on a Pentium III Xeon causes approximately 25 cycles overhead [11]). Note that in the following two properties, the parameter *level* indicates the level of the cache architecture.

```
public class CacheMissOverhead extends SimpleProperty {
    public CacheMissOverhead(RegionSummary summary, RegionSummary rankBasis, int level) {
        severity = summary.getNumberOfCacheMisses(level) *
                   summary.getExecutionNode().getCacheMissPenalty(level) /
                   rankBasis.getExecutionTime();
    }
}
```

A separate property has been defined to reflect the cache behavior of a code region, by determining the ratio between cache misses and accesses. Note that this property is not an overhead.

```
public class ImperfectCacheBehavior extends SimpleProperty {
    public ImperfectCacheBehavior(RegionSummary summary, int level) {
        severity = summary.getNumberOfCacheMisses(level) /
                   summary.getNumberOfCacheAccesses(level);
    }
}
```

5.2.6 Load Imbalance Properties

The relative load imbalance of a set *X* of performance values (e.g. execution times or number of instructions executed), each of which measured for a specific processing unit, is defined by $\frac{Avg(X)}{Max(X)}$, where *Avg(X)* and *Max(X)* determine the average and maximum value of elements in *X*. The relative load imbalance varies between *1* (perfect load balance) and $\frac{1}{|X|}$ (worst-case load imbalance). Thus, if we normalize the severity value for load imbalance to a range between 0 and 1, we can specify the following abstract class *LoadImbalance*:

```java
public abstract class LoadImbalance extends SimpleProperty {
    protected LoadImbalance(Experiment e, CodeRegion r) {
        CodeRegionFilter filter = new CodeRegionFilter(r);
        Statistics st = new Statistics(e.summaryIterator(filter)) {
            protected float getValue(RegionSummary rs) { return getValueOfInterest(rs); }
        };
        float loadImbalance = st.getAverage() / st.getMax();
        severity = (1 - loadImbalance) / (1 - 1/st.getGroupSize());
    }
    protected abstract float getValueOfInterest(RegionSummary rs);
}
```

By using the previous class, we can now easily define specific load imbalance performance properties that are based on a set of performance values, such as execution time load imbalance, synchronization load imbalance, message-passing load imbalance or computational load imbalance. The performance values which are used to compute the load imbalance are defined by overriding the method *getValueOfInterest*.

```java
public class ExecutionTimeLoadImbalance extends LoadImbalance {
    public ExecutionTimeLoadImbalance(Experiment e, CodeRegion r) { super(e, r); }
    protected float getValueOfInterest(RegionSummary rs)          { return rs.getExecutionTime(); }
}

public class SynchronizationLoadImbalance extends LoadImbalance {
    public SynchronizationLoadImbalance(Experiment e, CodeRegion r) { super(e, r); }
    protected float getValueOfInterest(RegionSummary rs)            { return rs.getSynchronizationTime(); }
}

public class CommunicationLoadImbalance extends LoadImbalance {
    public CommunicationLoadImbalance(Experiment e, CodeRegion r) { super(e, r); }
    protected float getValueOfInterest(RegionSummary rs)          { return rs.getCommunicationTime(); }
}

public class ComputationalLoadImbalance extends LoadImbalance {
    public ComputationalLoadImbalance(Experiment e, CodeRegion r) { super(e, r); }
    protected float getValueOfInterest(RegionSummary rs)          { return rs.getNumberOfInstructions(); }
}
```

5.3 Meta-properties

A meta-property is an abstract property whose definition depends on a set of already defined properties possibly known only during the execution time (which demands the use of Java reflection capabilities). For this purpose, JavaPSL provides an abstract property *Metaproperty* which represents an arbitrary set of property instances and hides the utilization of the Java reflection library from the user. Some important public methods of *Metaproperty* (see Figure 6) are explained as follows:

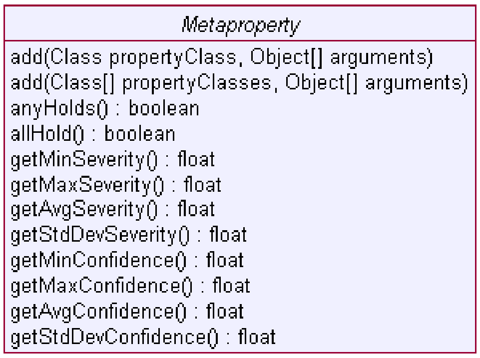

Figure 6. JavaPSL Meta-property class

- `add(Class propertyClass, Object[] arguments)`: An instance of the property *propertyClass* is created and added to the meta-property. The elements of *arguments* are used as parameters for the property constructor.

- `add(Class[] propertyClasses, Object[] arguments)`: An instance of each property in *propertyClasses* is created and added to the meta-property.

- `boolean allHold()`: Determines if all of the properties that have been added to the meta-property hold.

- `boolean anyHolds()`: Determines if at least one of the properties that have been added to the meta-property holds.

Statistical methods are also defined. For example, the methods *getAvgSeverity* and *getStdDevSeverity* respectively compute the average and the standard deviation of the severity values among all of the holding properties added to the meta-property (see Figure 6).

In the following we define a meta-property *OverheadForAnyExecution* for a region *r*, which verifies whether a single property holds for at least one execution of *r* in an experiment. If the meta-property holds then its severity and confidence will be, respectively, set to the maximum severity and the minimum confidence over all region summaries for which the property holds.

```
public abstract class OverheadForAnyExecution extends Metaproperty {
    protected OverheadForAnyExecution(Class property, Experiment parallelExp,
                                      RegionSummary rankBasis, CodeRegion r) {
        RegionSummaryIterator it = parallelExp.regionSummaryIterator(new CodeRegionFilter(r));
        while(it.hasNext()) {
            RegionSummary parSummary = it.next();
            Object[] arguments = new Object[]{ parSummary, rankBasis };
            add(property, arguments);
        }
    }
    public boolean holds()       { return anyHolds(); }
    public float getConfidence() { return getMinConfidence(); }
    public float getSeverity()   { return getMaxSeverity(); }
}
```

Based on the meta-property *OverheadForAnyExecution*, we can now easily create concrete properties to verify if there is at least one execution of a region for which an overhead property holds:

```
public class CommunicationOverheadForAnyExecution extends OverheadForAnyExecution {
    public CommunicationOverheadForAnyExecution(Experiment parallelExp, RegionSummary rankBasis,
                                                CodeRegion r) {
        super(CommunicationOverhead.class, parallelExp, rankBasis, r);
    }
}

public class SynchronizationOverheadForAnyExecution extends OverheadForAnyExecution {
    public SynchronizationOverheadForAnyExecution(Experiment parallelExp, RegionSummary rankBasis,
                                                  CodeRegion r) {
        super(SynchronizationOverhead.class, parallelExp, rankBasis, r);
    }
}

public class LateSenderForAnyExecution extends OverheadForAnyExecution {
    public LateSenderForAnyExecution(Experiment parallelExp, RegionSummary rankBasis,
                                     CodeRegion r) {
        super(LateSender.class, parallelExp, rankBasis, r);
    }
}
```

A property such as *NonScalability* (see Section 5.2.1), which is based on the definition of efficiency, can be alternatively written by using a meta-property, since we have already defined the property *Inefficiency*:

```
public class AlternativeNonScalability extends Metaproperty {
    private float severity;
```

```
    public AlternativeNonScalability(Experiment sequentialExp, Experiment[] parallelExps, CodeRegion r)
        for(int i = 0; i < parallelExps.length; i++) {
            add(Inefficiency.class, new Object[]{sequentialExp, parallelExps[i], r});
        }
        severity = getMaxSeverity() - getAvgSeverity();
    }

    public boolean holds()       { return severity > 0; }
    public float getSeverity()   { return severity; }
    public float getConfidence() { return 1; }
}
```

Note also that a performance tool can cache the properties that are implicitly computed by a meta-property, thus avoiding redundant calculations for subsequent property computations.

6 Experimental Results

We have implemented a prototype of a performance tool outlined in Figure 1 that uses JavaPSL to determine performance properties for MPI, OpenMP, and mixed OpenMP/MPI programs. Our current prototype tries to compute every performance property defined in this paper for user-definable code regions based on the experiment-related data available. SCALEA [19] is used to instrument and profile input programs. Moreover, profile data is automatically converted to the format defined in Section 4.1.

In this section, we evaluate our prototype implementation by applying it to several applications executed on Gescher, an SMP cluster with 6 SMP nodes (connected by FastEthernet), each of which comprises 4 Intel Pentium III Xeon 700 MHz CPUs.

6.1 LAPW0

The program LAPW0 [4] calculates the effective potential of the Kohn-Sham [12] eigen-value problem, which consists of two parts, the Coulomb potential (subdivided into Coulomb-1 and Coulomb-2 sub-regions) and the exchange-correlation potential (named here XC-computation).

Implemented as a Fortran MPI code, LAPW0 has been examined for a fixed problem size with different machine configurations (1, 2, 4, and 6 SMP nodes, each of which executes 1, 2, 3 or 4 LAPW0 processes).

6.1.1 Non-scalability and Inefficiency

	entire program	Coulomb-1	Coulomb-2	XC-computation
severity	0.41	0.31	0.20	0.26

Table 1. NonScalability severity values between 0 (best case) and 1 (worst case) for LAPW0

The starting point of our bottleneck analysis is the property *NonScalability*, which has been computed based on the set of experiments described above. Table 1 shows that LAPW0 does not scale well. In order to determine the cause of this behavior we also examine the *Inefficiency* property. In Figure 7 we can observe increasing inefficiency for all code regions of interest, where the XC-computation shows the most detrimental inefficiency behavior. We also observe, in the code regions Coulomb-1 and Coulomb-2, that for specific processing unit numbers (e.g. for 12 and 18) the efficiency improves. To further examine this effect, we consider the *LoadImbalance* property.

6.1.2 Load Imbalance and Overhead Sources

The results for the *ExecutionTimeLoadImbalance* property are shown in Figure 8, where we can see that the inefficiency problems shown in Figure 7 correlate with the execution time load imbalance; furthermore, we observe that, for some problem sizes, there is an evident load imbalance problem, which can be explained by the problem size of 36 atoms used in our experiments. Since LAPW0 distributes the atoms onto the set of processing units, when using 8, 16, and 24 processing units

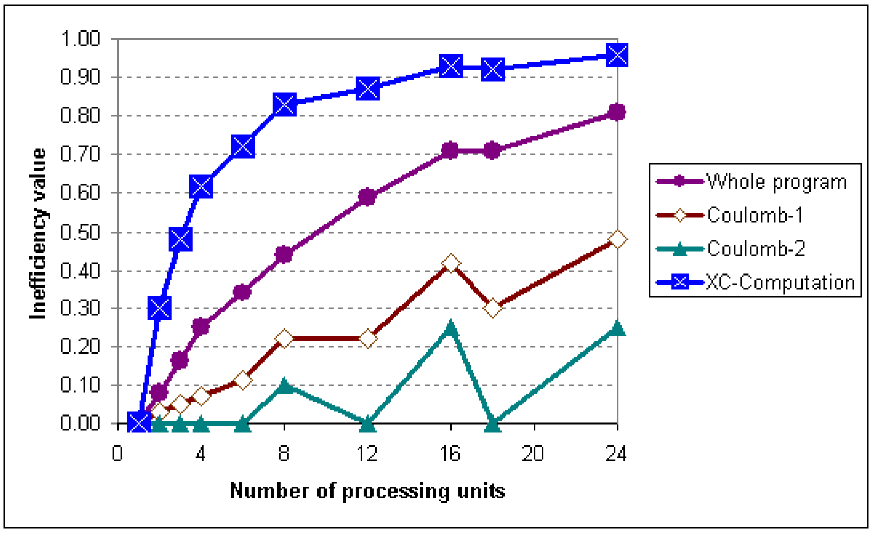

Figure 7. Inefficiency in LAPW0

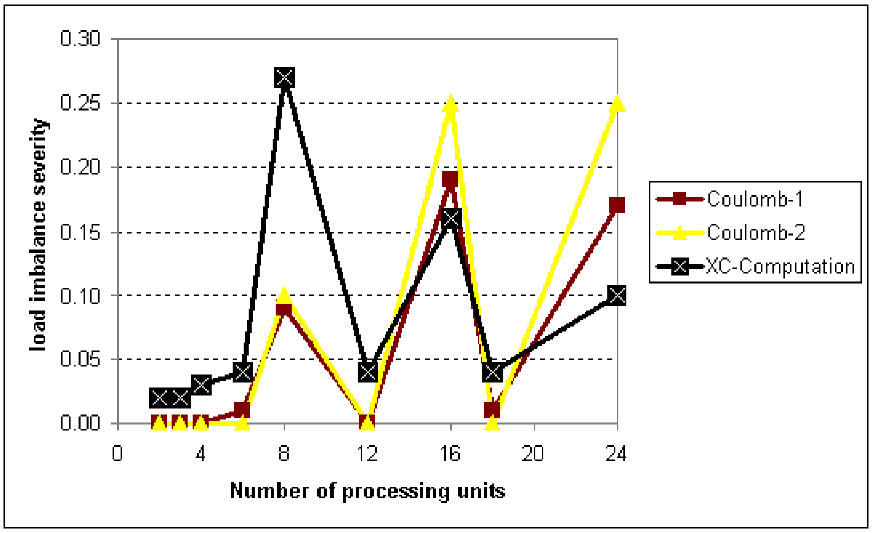

Figure 8. ExecutionTimeLoadImbalance in LAPW0

we cannot reach optimal load balance, whereas 1, 2, 4, 6, 12 and 18 processing units display a much better load balance. This effect is clearly confirmed by the *ExecutionTimeLoadImbalance* property of JavaPSL.

Next, we examine LAPW0 with respect to communication overhead. By using our properties we determine that XC-Computation (see Figure 9) implies significant communication overhead, whereas for the other code regions this property does not hold. Figure 9 shows a very irregular communication behavior depending on the number of processing units involved. Moreover, we can detect the real reason for communication overhead by examining the *LateSender* property, which is almost identical with the overall communication overhead. Again, for 8, 16 and 24 processing units, the severity value is considerably larger than for other machine configurations.

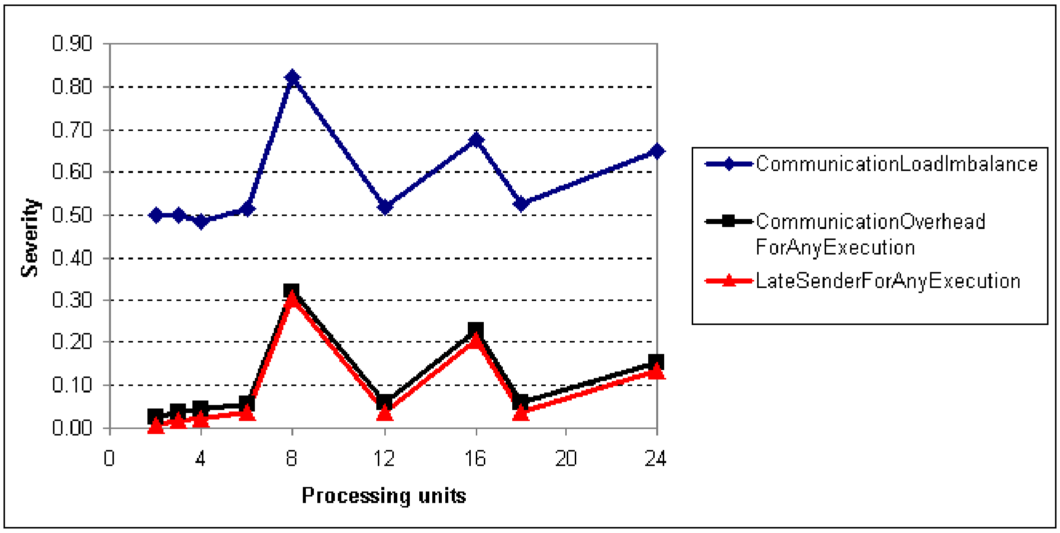

Figure 9. Communication overhead for XC-Computation in LAPW0

6.2 Backward Pricing

The backward pricing code [6] is a parallel implementation of the backward induction algorithm to compute the price of an interest rate dependent financial product, such as a variable coupon bond.

Backward pricing has been implemented as a mixed Fortran MPI and OpenMP code. For the experiments we used a single realistic problem size and different machine configurations (1, 2, 4 or 6 SMP nodes, each one executing 1, 2, 3 or 4 threads). Two code regions were analyzed: the entire program and the core OpenMP loop, which accounts for more than 90 % of the overall execution time.

Again, we start our performance analysis with the computation of the *NonScalability* property for the code regions and machine configurations mentioned above. Table 2 clearly displays that, for the machine configurations tested, the backward pricing code scales very well. Note that scalability, according to our definition, only determines whether the efficiency of several experiments is nearly the same. A good scaling behavior does not guarantee good efficiency. For this reason, we examine the *Inefficiency* property values (see Table 3). Except for the experiment with 6 SMP nodes with 4 threads each, the inefficiency property cannot be proven or is insignificant. By further examining the *controlOfParallelism* property for this machine configuration, we detect that its severity is 0.35 (compared to the execution time of the entire program). Finally, by evaluating every code region that falls into the category of control of parallelism, the performance tool detected that MPI_INIT is the predominant overhead.

	entire program	OpenMP loop
severity	0.12	0.00

Table 2. NonScalability severity values for Backward Pricing

7 Conclusions and Future Work

In this article we describe JavaPSL, a generic performance specification language for modeling experiment-related data and performance properties of distributed and parallel programs. Performance properties characterize a specific negative performance behavior of a program and are defined over experiment-related data.

JavaPSL uses powerful Java mechanisms, in particular, polymorphism, abstract classes, and reflection, to describe performance properties. Moreover, JavaPSL provides meta-properties – defined as Java abstract classes – in order to describe

Number of SMP Nodes	Number of threads	Holds?	Severity
6	4	yes	0.14
6	3	no	-.-
6	2	no	-.-
6	1	yes	0.05
4	4	no	-.-
4	3	no	-.-
4	2	no	-.-
4	1	yes	0.03
2	4	no	-.-
2	3	no	-.-
2	2	no	-.-
2	1	yes	0.01
1	4	no	-.-
1	3	no	-.-
1	2	no	-.-
1	1	no	-.-

Table 3. Inefficiency severity values for Backward Pricing

new properties based on existing ones and to relate properties among each other. Performance properties can be related to the code regions that cause them through experiment-related data. JavaPSL filter and statistics classes can be used to restrict performance analysis to specific experiment-related data, and to compute statistics based on arbitrary sets of performance values. A variety of pre-defined performance properties are supported to analyze one or several experiments, which examine, for instance, the load imbalance or the scalability behavior of a program.

We propose JavaPSL to be a standard performance information interface to model a large variety of performance information, to build sophisticated performance tools (e.g. to provide automatic bottleneck analysis), and to enable portable access to performance information. This interface is intended to be used by performance tools, compilers, program transformation systems, etc.

We have implemented a prototype performance tool that uses JavaPSL to automatically detect performance properties in realistic applications implemented as MPI, OpenMP and mixed OpenMP/MPI programs.

We are currently extending JavaPSL to enable the specification of search algorithms for performance problems and its refinement. Moreover, we improve our performance tool to include a more sophisticated search for performance properties based on JavaPSL.

References

[1] E. Luque A. Espinosa, T. Margalef. Automatic Performance Evaluation of Parallel Programs. In *IEEE Proc. of the 6th Euromicro Workshop on Parallel and Distributed Processing*. IEEE Computer Society Press, January 1998.

[2] R. Aydt and R. Ribler. Autopilot user's manual . ftp://www-pablo.cs.uiuc.edu/pub/Autopilot/Documentation/AutopilotManual.ps.gz, April 2000.

[3] R. A. AYDT. SDDF: The pablo self-describing data format. Tech. rep., Department of Computer Science, University of Illinois, April 1994.

[4] P. Blaha, K. Schwarz, and J. Luitz. WIEN97, Full-potential, linearized augmented plane wave package for calculating crystal properties. Institute of Technical Electrochemistry, Vienna University of Technology, Vienna, Austria, ISBN 3-9501031-0-4, 1999.

[5] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceeding SC'2000*, November 2000.

[6] E. Dockner and H. Moritsch. Pricing Constant Maturity Floaters with Embeeded Options Using Monte Carlo Simulation. Technical Report AuR_99-04, AURORA Technical Reports, University of Vienna, January 1999.

[7] J. Dongarra, S. Browne, G. Fagg, and E. Strohmaier. Scalability analysis. http://www.cs.utk.edu/ Edongarra/WEB-PAGES/lect15.ps, April 1999.

[8] T. Fahringer, M. Gerndt, G. Riley, and J. Träff. Knowledge Specification for Automatic Performance Analysis, Revised Version. APART Technical Report, Workpackage 2, Identification and Formalization of Knowledge, Technical Report http://www.kfa-juelich.de/apart/result.html, Research Centre Jülich, Zentralinstitut für Angewandte Mathematik (ZMG), Jülich, Germany, January 2001.

[9] W. Gropp and E. Lusk. Sowing MPICH: A case study in the dissemination of a portable environment for parallel scientific computing. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):103–114, Summer 1997.

[10] B. Robert Helm, Allen D. Malony, and Stephen F. Fickas. Capturing and automating performance diagnosis: The Poirot approach. In *Proceedings of the 9th International Symposium on Parallel Processing (IPPS'95*, pages 606–613, Los Alamitos, CA, USA, April 1995. IEEE Computer Society Press.

[11] Intel Corporation, http://support.intel.com/design/pentiumiii/manuals. *Intel Architecture Optimization, Reference Manual*, 1999.

[12] W Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev. A*, 140:1133–1138, 1965.

[13] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing:design and analysis of parallel algorithms*. Benjamin/Cummings, 1994.

[14] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37 – 46, November 1995.

[15] B. Mohr, D. Brown, and A. Malony. TAU: A portable parallel program analysis environment for pC++. In *CONPAR*, Linz, Austria, 94.

[16] N. Mukherjee, G.D. Riley, and J.R. Gurd. Finesse: A prototype feedback-guided performance enhancement system. In *Euromicro Workshop on Parallel and Distributed Processing PDP'2000, Rhodes, Greed*. IEEE Computer Society, January 2000.

[17] Pallas GmbH. *Vampir 2.0 User's Manual*, 6 1999.

[18] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison Wesley Longman, Reading, Mass., 1999.

[19] Hong-Linh Truong, Thomas Fahringer, Georg Madsen, Allen D. Malony, Hans Moritsch, and Sameer Shende. On Using SCALEA for Performance Analysis of Distributed and Parallel Programs. In *Proceeding SC'2001, Denver, USA*, November 2001.

[20] F. Wolf and B. Mohr. EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs. In *Proc. of 7th International Conference, HPCN Europe 1999*, pages 503–512, Amsterdam, The Netherlands, April 1999.