SCALEA: A Performance Analysis Tool for Parallel Programs[†]

Hong-Linh Truong[‡], Thomas Fahringer[§]

Institute for Software Science, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria

SUMMARY

Many existing performance analysis tools lack the flexibility to control instrumentation and performance measurement for code regions and performance metrics of interest. Performance analysis is commonly restricted to single experiments.

In this paper we present SCALEA, which is a performance instrumentation, measurement, analysis, and visualization tool for parallel programs that supports post-mortem performance analysis. SCALEA currently focuses on performance analysis for OpenMP, MPI, HPF, and mixed parallel programs. It computes a variety of performance metrics based on a novel classification of overhead. SCALEA also supports multi-experiment performance analysis that allows to compare and to evaluate the performance outcome of several experiments. A highly flexible instrumentation and measurement system is provided which can be controlled by command-line options and program directives. SCALEA can be interfaced by external tools through the provision of a full Fortran90 OpenMP/MPI/HPF frontend that allows to instrument an abstract syntax tree at a very high-level with C-function calls and to generate source code. A graphical user interface is provided to view a large variety of performance metrics at the level of arbitrary code regions, threads, processes, and computational nodes for single- and multi-experiment.

KEY WORDS: *performance analysis, instrumentation, performance overheads, visualization, parallel programs*

[†]This research is supported by the Austrian Science Fund as part of the Aurora Project under contract SFBF1104.
[‡]E-mail: truong@par.univie.ac.at
[§]E-mail: tf@par.univie.ac.at

1. INTRODUCTION

The evolution of parallel and distributed architectures and programming paradigms for performance-oriented program development challenge the state of technology for performance tools. Coupling different programming paradigms such as message passing and shared memory programming for hybrid cluster computing (e.g. SMP clusters) is one example for high demands on performance analysis tools that are capable to cope with applications for multiple programming models and target architectures. Performance tools must be able to observe performance problems at all levels of a system while relating low-level behavior to the application program.

One of the challenges faced by performance analysis tools is the selection of an appropriate level of measurement and analysis detail to systematically identify the origins of performance problems. On the one hand, a programmer may be well aware about which code sections and performance metrics should be in the center of performance analysis. On the other hand, tools can automatically divide a program into code regions, determine whether those regions cause performance problems and map code region to a given class of performance overheads. A combination of user directions and tools functionality can provide a simple and efficient mechanism to control the level of measurement and analysis. The difference between ideal and measured execution time is commonly defined as the overhead implied by parallelizing a program. A classification of performance overheads will provide the user with a detailed understanding of where and how performance was lost. Most existing tools, however, lack a classification of performance overheads and support limited measurement and instrumentation features, thus a systematic performance analysis is severely hampered.

In this paper we describe SCALEA, a performance instrumentation, measurement, and analysis system for distributed and parallel architectures that currently focuses on OpenMP [34], MPI [15], HPF [18], and mixed programming paradigms such as OpenMP/MPI. SCALEA seeks to explain the performance behavior of each program by computing a variety of performance metrics based on a novel classification of performance overheads for shared and distributed memory parallel programs which includes data movement, synchronization, control of parallelism, additional computation, loss of parallelism, and unidentified overheads. In order to determine overheads, SCALEA divides the program sources into code regions (ranging from entire program units to single statements) and finds out what performance problems occur in those regions. A highly flexible instrumentation and measurement system is provided which can precisely be controlled by program directives and command-line options. In the center of SCALEA's performance analysis is a novel dynamic code region call graph (DRG) which reflects the dynamic relationship between code regions and their subregions and enables a detailed overhead analysis for every code region. SCALEA combines source code and hardware-profiling in a single system, which broadens the performance aspects that can be examined substantially. A combination of command-line options and program directives enables the programmer to precisely control instrumentation, measurement, and analysis so that performance metrics can be computed for user-selected or pre-defined code regions. Moreover, SCALEA supports a high-level interface to traverse an abstract syntax tree (AST), to locate arbitrary code regions, and to mark them for instrumentation. Thus the SCALEA instrumentation and overhead analysis engine can be used by external tools as well.

A data repository is employed in order to store performance data and information about performance experiments which alleviates the association of performance information with experiments and the source code. SCALEA also supports multi-experiment performance analysis that allows to examine and compare the performance outcome of different program executions. A sophisticated visualization engine is provided to view the performance of programs at the level of arbitrary code regions, threads, processes, and computational nodes (e.g. single-processor systems, Symmetric Multiple Processor (SMP) nodes sharing a common memory) for single- and multi-experiment.

The rest of this paper is organized as follows: Section 2 presents an overview of SCALEA. In Section 3 we present a classification of performance overheads. The next section outlines the various instrumentation mechanisms offered by SCALEA. Section 5 presents the dynamic code region call graph. The performance data repository is described in the following Section. Experiments are shown in Section 7. Related work is outlined in Section 8, followed by conclusions in Section 9.

2. SCALEA OVERVIEW

SCALEA is a performance instrumentation, measurement, and analysis system for distributed memory, shared memory, and mixed parallel programs. Figure 1 shows the architecture of SCALEA which consists of several components: SCALEA Instrumentation System (SIS), SCALEA Runtime System, SCALEA Performance Data Repository, and SCALEA Performance Analysis & Visualization System. All components provide well-defined interfaces thus they can easily be used by external tools as well.

SIS uses the front-end and unparser of the VFC compiler [4]. SIS supports automatic instrumentation of MPI, OpenMP, HPF, and mixed OpenMP/MPI programs. The user can select (by directives or command-line options) code regions (loops, subroutines, OpenMP regions, MPI Communications, etc) and performance metrics (timing, HW-parameters, and performance overheads) of interest. Moreover, SIS offers an interface for other tools to traverse and annotate the AST at a high level in order to specify code regions for which performance metrics should be obtained. SIS also generates an *instrumentation description file* to relate all gathered performance data back to the input program.

The SCALEA runtime system supports profiling and tracing for parallel and distributed programs, and sensors and sensor managers for capturing and managing performance data of individual computing nodes of parallel and distributed machines. The SCALEA profiling and tracing library collects timing, event, and counter information, as well as hardware parameters (determined through an interface with the PAPI library [6]).

The SCALEA performance analysis and visualization module analyzes the raw performance data which is collected post-mortem and stored in the performance data repository. It computes all user-requested performance metrics, and visualizes them together with the input program. Besides single-experiment analysis, SCALEA also supports multi-experiment performance analysis. The visualization engine provides a rich set of displays for various metrics in isolation or together with the source code.

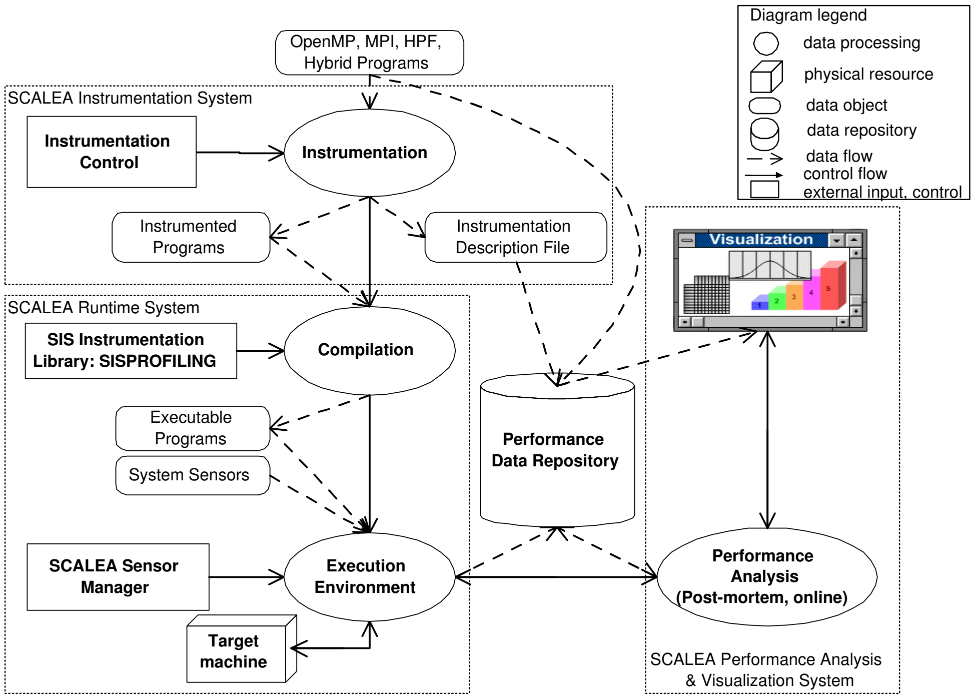


Figure 1. Architecture of SCALEA.

The SCALEA performance data repository holds relevant information about the experiments conducted which includes raw performance data and metrics, source code, machine information, etc. In the following sections, we provide a more detailed overview of SCALEA.

3. CLASSIFICATION OF TEMPORAL OVERHEADS

According to Amdahl's law [1], theoretically the best sequential algorithm takes time T_s to finish the program, and T_p is the time required to execute the parallel version with p processors. The temporal overhead of a parallel program is defined by $T_o = T_p - T_s/p$ and reflects the difference between achieved and optimal parallelization. T_o can be divided into T_i and T_u such that $T_o = T_i + T_u$, where T_i is the overhead that can be identified and T_u is the overhead fraction which could not be analyzed in detail. In theory T_o cannot be negative, which implies that the speedup T_s/T_p cannot exceed p [20]. However, in practice it occurs that temporal overhead can become negative due to super linear speedup of applications. This effect is commonly caused by an increased available cache size.

In previous work [32], we presented a preliminary and very coarse grain classification of performance overheads which has been stimulated by [7]. Figure 2 shows our novel and substantially refined overhead classification which includes:

- *Data movement* shown in Fig. 2(b) corresponds to any data transfer within local memory (e.g. cache misses and page faults), file I/O, communication (e.g. point to point or collective communication), and remote memory access (e.g. put and get). Note that the overhead *Communication of Accumulate Operation* has been stimulated by the *MPI_Accumulate* construct which is employed to move and combine (through reduction operations) data at remote sites via remote memory access.
- *Synchronization* (e.g. barriers and locks) shown in Fig. 2(c) is used to coordinate processes and threads when accessing data, maintaining consistent computations and data, etc. We subdivided the synchronization overhead into single- and multi-address space overheads. A single-address space overhead corresponds to a synchronization inside a single process on parallel systems, for instance any kind of OpenMP synchronization falls into this category, whereas multi-address space synchronization has been stimulated by MPI [15] synchronization, remote memory locks, barriers among different processes, etc.
- *Control of parallelism* (e.g. fork/join operations and loop scheduling) shown in Fig. 2(d) is used to control and manage the parallelism of a program which is commonly caused by code inserted by the compiler (e.g. runtime library) or by the programmer (e.g. to implement data redistribution).
- *Additional computation* (see Fig. 2(e)) reflects any change of the original sequential program including algorithmic or compiler changes to increase parallelism (e.g. by eliminating data dependences) or data locality (e.g. through changing data access patterns). Moreover, requests for processing unit identifications or for the number of threads to execute a code region may also imply additional computation overhead.
- *Loss of parallelism* (see Fig. 2(f)) is due to imperfect parallelization of a program. Loss of parallelism is further classified into: unparallelized code (executed by only one processor), replicated code (executed by all processors), and partially parallelized code (executed by more than one but not all processors).
- *Unidentified overhead* corresponds to the overhead that is not covered by the above categories.

4. SCALEA Instrumentation System (SIS)

SIS provides the user with three alternatives to control instrumentation which includes command-line options, SIS directives, and a high-level instrumentation library combined with an OpenMP/MPI/HPF frontend and unparser. All of these alternatives support the specification of performance metrics and code regions of interest for which SCALEA automatically generates instrumentation code and determines the desired performance values during or after program execution. In the remainder of this paper we assume that a code region refers to a single-entry single-exit code region. A large variety of predefined mnemonics

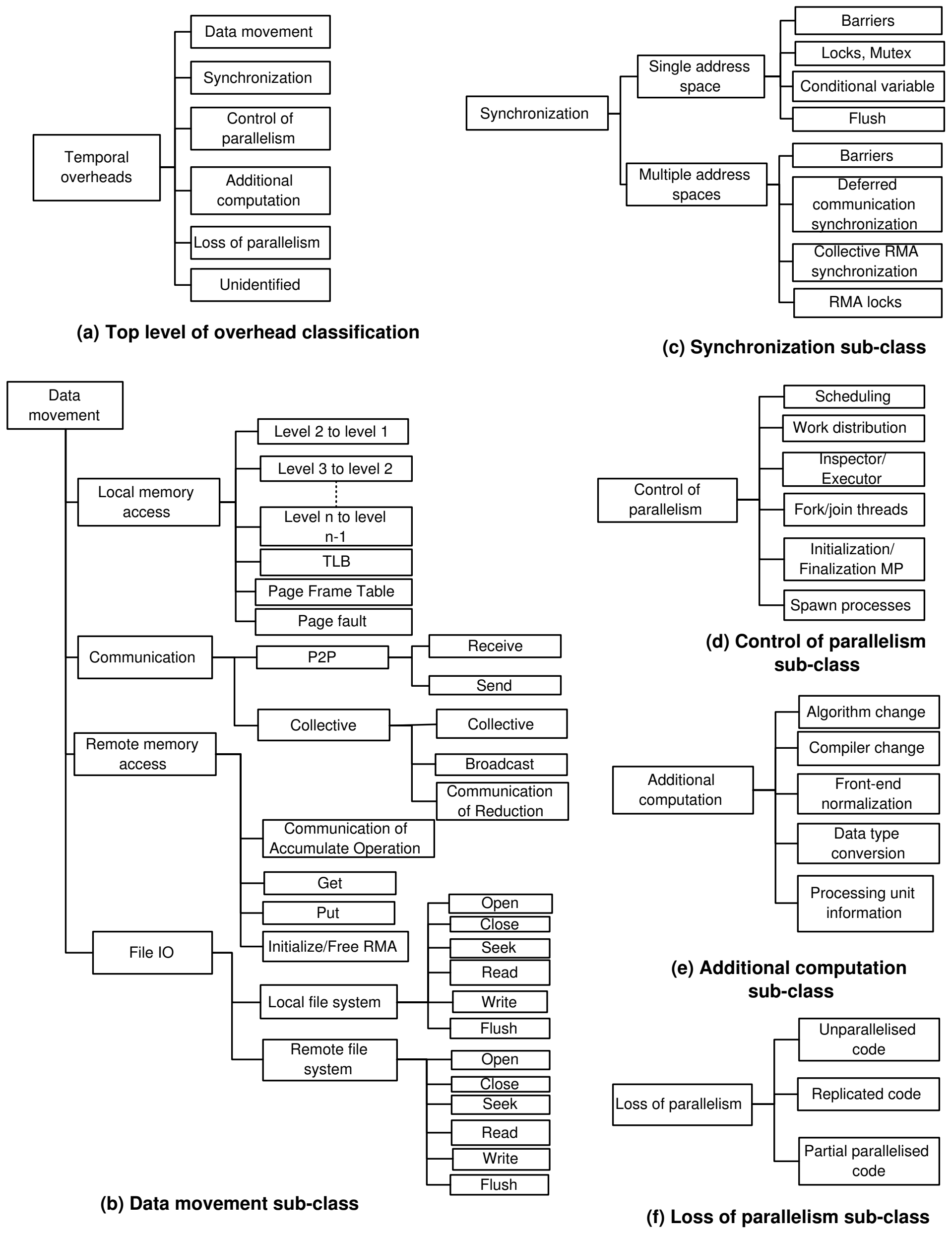


Figure 2. Classification of temporal overheads.

are provided by SIS for instrumentation purposes. The current implementation of SCALEA supports 49 code region and 30 performance metric mnemonics:

- *code region mnemonics*: arbitrary code regions, loops, outermost loops, procedures, I/O statements, HPF INDEPENDENT loops, HPF redistribution, OpenMP parallel loops, OpenMP sections, OpenMP critical, MPI send, receive, and barrier statements, etc.
- *performance metric mnemonics*: wall clock time, cpu time, communication overhead, cache misses, barrier time, synchronization, scheduling, compiler overhead, unparallelized code overhead, HW-parameters, etc. See also Fig. 2 for a classification of performance overheads considered by SCALEA.

4.1. Instrumentation

The user can specify arbitrary code regions ranging from entire program units to single statements and name (to associate performance data with code regions) these regions by using the following directive:

!SIS$ CR *region_name* **BEGIN**
    code region
!SIS$ END CR

In order to specify a set of code regions $R = \{r_1, ..., r_n\}$ in an enclosing region r and performance metrics which should be computed for every region in R, SIS offers the following directive:

!SIS$ CR *region_name* [,*cr_mnem-list*] [**PMETRIC** *perf_mnem-list*] **BEGIN**
    code region r that includes all regions in R
!SIS$ END CR

The code region r defines the *scope* of the directive. Note that every (code) region in R is a sub-region of r but r may contain sub-regions that are not in R.

The code region (*cr_mnem-list*) and performance metric (*perf_mnem-list*) mnemonics are indicated as a list of mnemonics separated by commas. One of the code region mnemonics (CR_A) refers to arbitrary code regions. Note that the above specified directive allows the user to indicate either only code region mnemonics or performance metric mnemonics, or a combination of both. If in a SIS directive d only code region mnemonics are indicated, then SIS is instrumenting all code regions that correspond to these mnemonics inside of the scope of d. The instrumentation is done for a set of default performance metrics which can be overwritten by command-line options or at runtime. This option can be particularly useful if the user knows that only a few code regions can cause critical performance problems whose performance overheads are unknown. Only if the mnemonic CR_A is included in the list of code region mnemonics of a directive d, then instrumentation for arbitrary code regions inside of the scope of d will be conducted.

If only performance metric mnemonics are indicated in a directive d then SIS is instrumenting those code regions that have an impact on the specified metrics. This option is useful if a user is interested in specific performance metrics but does not know which code regions may cause these overheads. For instance, compilers often substantially restructure OpenMP (e.g. implicit synchronization) and HPF (e.g. implicit data redistribution) codes which is not visible at the input code. In order to find the associated compiler overhead, the programmer could specify some control of parallelism overhead mnemonics in a SIS directive.

If both code regions and performance metrics are defined in a directive *d*, then SIS is instrumenting these code regions for the indicated performance metrics in the scope of *d*. Feasibility checks are conducted by SIS, for instance, to determine whether the programmer is asking for OpenMP overheads in HPF code regions. For these cases, SIS outputs appropriate warnings. If neither code region nor overhead mnemonics are indicated then the directive is simply ignored.

All previous directives are called local directives as the scope of these directives is restricted to a part of a program unit (main program, subroutines or functions). The scope of a directive can be extended to the full program unit by using the following syntax:

!SIS$ CR [*cr_mnem-list*] [**PMETRIC** *perf_mnem-list*]

A global directive *d* collects performance metrics – indicated in the `PMETRIC` part of *d* – for all code regions – specified in the `CR` part of *d* – in the program unit which contains *d*. A local directive implies the request for performance information restricted to the scope of *d*. There can be nested directives with arbitrary combinations of global and local directives. If different performance metrics are requested for a specific code region by several nested directives, then the union of these metrics is determined. In addition, SIS supports command-line options to instrument specific code regions for well-defined performance metrics in the entire application (across all program units).

Moreover, SIS provides directives to control tracing/profiling. The directives MEASURE ENABLE and MEASURE DISABLE allow the programmer to turn on and off tracing/profiling of a specific code region.

!SIS$ MEASURE DISABLE
code region
!SIS$ MEASURE ENABLE

SIS also provides an interface that can be used by other tools to exploit SCALEA's instrumentation features. We have developed a C-library to traverse the AST and to mark arbitrary code regions for instrumentation. For each code region, the user can specify the performance metrics of interest. Based on the annotated AST, SIS automatically generates an instrumented source code.

In the following example we demonstrate some of the directives as mentioned above by showing a fraction of an application code of Section 7.

```
d1:      !SIS$ CR PMETRIC ODATA_SEND, ODATA_RECV, ODATA_COL
         call MPI_BCAST(nx, 1,MPI_INTEGER, mpi_master,MPI_COMM_WORLD,mpi_err)
         ...
d2:      !SIS$ CR comp_main, CR_A, CR_S PMETRIC WTIME, L2_TCM BEGIN
             ...
d3:          !SIS$ CR init_comp BEGIN
                 dj=real(nx,b8)/real(nodes_row,b8)
```

```
                ...
d4:         !SIS$ END CR
                ...
d5:         !SIS$ MEASURE DISABLE
                call bc(psi,i1,i2,j1,j2)
d6:         !SIS$ MEASURE ENABLE
                ...
                call do_force(i1,i2,j1,j2)
                ...
d7:     !SIS$ END CR
```

Directive d_1 is a global directive which instructs SIS to instrument all send, receive and collective communication statements in this program unit. Directives d_2 (begin) and d_7 (end) define a specific code region with the name *comp_main*. Within code region *comp_main*, SCALEA will determine wall clock times (*WTIME*) and the total number of L2 cache misses (*L2_TCM*) for all arbitrary code regions (based on mnemonic *CR_A*) and subroutine calls (mnemonic *CR_S*) as specified in d_2. Directives d_3 and d_4 specify an arbitrary code region with the name *init_comp*. No instrumentation as well as measurement is done for the code region between directives d_5 and d_6.

4.2. Instrumentation Description File (IDF)

A crucial aspect of performance analysis is to relate performance information back to the original input program. When instrumenting a program, SIS generates an *instrumentation description file* (IDF) which correlates profiling, trace and overhead information with the corresponding code regions. The IDF maintains for every instrumented code region a variety of information (see Table I).

IDF Entry	**Description**
id	code region identifier
type	code region type
file	source file identifier
unit	identifier of the program unit that encloses this region
line_start	line number where this region starts
column_start	column number where this region starts
line_end	line number where this region ends
column_end	column number where this region ends
performance_metric_mnemonics	mnemonics representing performance metrics which will be collected or computed for this region
aux	auxiliary information

Table I. Content of an entry in the instrumentation description file (IDF)

A code region type describes the type of the code region, for example, entire program unit, outermost loop, read statement, OpenMP SECTION, OpenMP parallel loop, MPI barrier, etc.

The program unit corresponds to a subroutine or function which encloses the code region. The performance data of a code region stored in a separate repository is associated with the code region description via the code region identifier.

5. DYNAMIC CODE REGION CALL GRAPH

Every program consists of a set of *code regions* which can range from single statements to entire program units. A code region can be, respectively, entered and exited by multiple entry and exit control flow nodes (points) (see Figure 3). In most cases, however, code regions are single-entry-single-exit code regions.

In order to measure the execution behavior of a code region, the instrumentation system has to detect all *entry* and *exit* nodes of the code region and insert probes at these nodes. Basically, this task can be done with the support of a compiler or guided through manual insertion of directives. Figure 3 shows an example of a code region with its entry and exit nodes; each node represents a statement in the program. To select an arbitrary code region, the user, respectively, marks two statements as the entry and exit statements – which are at the same time entry and exit nodes – of the code region (e.g., by using SIS directives). Through the compiler analysis, SIS then automatically tries to determine all entry and exit nodes of the code region. The instrumentation tries to detect all of these nodes and automatically inserts probes before and after all entry and exit nodes, respectively. Note that code regions can overlap each other, however, SIS at this point does not support instrumentation of overlapped code regions. The current implementation of SIS supports mainly instrumentation of single-entry multiple-exit code regions. We are enhancing SIS to cover also multiple-entry multiple-exit code regions.

5.1. Dynamic code region call graph

SIS has a set of predefined (static) code regions which are classified into common (e.g. program, procedure, loop, function call, statement) and programming paradigm-specific code regions (MPI_calls, HPF INDEPENDENT loops, OpenMP parallel regions, loops, and sections, etc.). Moreover, SIS provides directives to define arbitrary code regions (see Section 4) in the input program.

An instrumented (static) code region can be called multiple times (each time one activation of the code region is executed) during runtime of a program. One activation of an instrumented code region is associated with a *code region stack trace* which is our extension version of the program stack trace for functions [16] to arbitrary code regions :

> A *code region stack trace* of a program at an instant of time is the sequence of instrumented code regions that are active at that time.

A code region stack trace C denoted by $C = (c_{r_{m_1}} \rightarrow c_{r_{m_2}} \rightarrow \cdots \rightarrow c_{r_{m_k}})$ starts with an activation $c_{r_{m_1}}$ of the root code region r_{m_1}, followed by an activation $c_{r_{m_2}}$ of code region r_{m_2} called inside the activation $c_{r_{m_1}}$, followed by an activation of code region r_{m_3} called inside the activation $c_{r_{m_2}}$, and so on. The above definition implies that activations of the same code

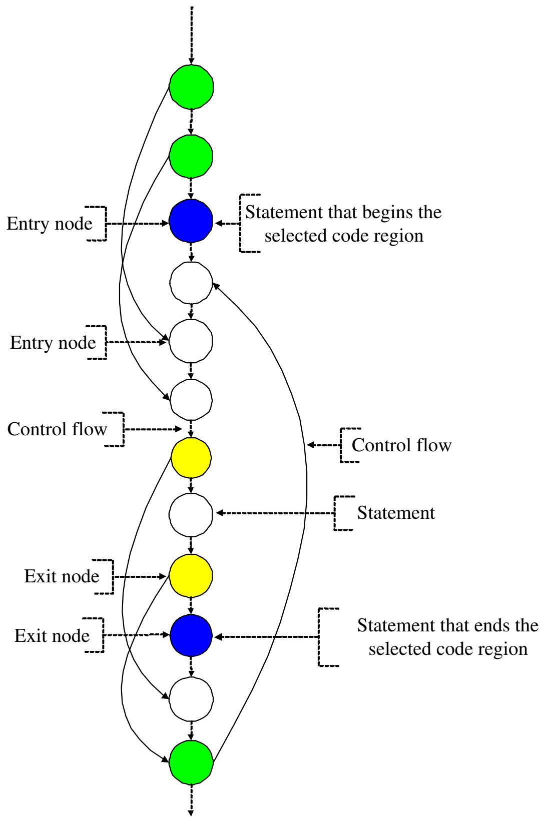

Entry node
Statement that begins the selected code region
Entry node
Control flow
Control flow
Statement
Exit node
Exit node
Statement that ends the selected code region

Figure 3. A code region with several entry and exit nodes.

region may appear in the same code region stack trace or in various code region stack traces. For example, a program Q which has a set of code regions $R = \{r_1, r_2, r_3, r_4\}$ may have three different code region stack traces at different times as follows:

- $c^1_{r_1} \rightarrow c^1_{r_3} \rightarrow c^1_{r_4}$,
- $c^1_{r_1} \rightarrow c^1_{r_2} \rightarrow c^3_{r_3} \rightarrow c^4_{r_4}$, and
- $c^1_{r_1} \rightarrow c^3_{r_2} \rightarrow c^4_{r_3} \rightarrow c^5_{r_4}$

where r_1 is the root code region and $c^l_{r_i} \rightarrow c^k_{r_j}$ means that activation numbered k of code region r_j is called directly by activation numbered l of code region r_i. We then define the equivalence relation for code region stack traces:

Let $p^{n_k}_{r_{m_k}}$ be the activation numbered n_k of code region r_{m_k} and $q^{n'_{k'}}_{r'_{m_{k'}}}$ be the activation numbered $n'_{k'}$ of code region $r'_{m_{k'}}$. $C_p = (p^{n_1}_{r_{m_1}} \rightarrow p^{n_2}_{r_{m_2}} \rightarrow \cdots \rightarrow p^{n_k}_{r_{m_k}})$

is the code region stack trace which leads to activation $p^{n_k}_{r_{m_k}}$ and $C_q = (q^{n'_1}_{r'_{m_1}} \rightarrow q^{n'_2}_{r'_{m_2}} \rightarrow \cdots \rightarrow q^{n'_{k'}}_{r'_{m_{k'}}})$ is the code region stack trace which leads to activation $q^{n'_{k'}}_{r'_{m_{k'}}}$. The stack trace C_p is called equivalent with C_q iff both stack traces satisfy the following conditions: (i) $k = k'$, $k \geqslant 1$ and (ii) for all $1 \leqslant i \leqslant k$, $r_{m_i} \equiv r'_{m_i}$.

To clarify the above definition, we back to the abovementioned example. The equivalence relation holds for the code region stack trace of activation $c^4_{r_4}$ and $c^5_{r_4}$. However, the code region stack trace of activation $c^1_{r_4}$ and $c^4_{r_4}$ do not fulfill the conditions of the equivalence relation. The key idea is that if activations have equivalent code region stack traces, we can compact them into a record of the data structure representing the calling behavior of code regions while still can associate their performance metrics with their calling context. For example, instead of storing information of activation $c^4_{r_4}$ and $c^5_{r_4}$ in two separate records, we save their information into one record.

We then extend the code region stack trace into the context of the parallel program with the assumption that the parallel program has multiple processes, each of them contains several threads of computation, and it is executed in set of computational nodes at the runtime. Each code region stack trace is associated with a thread on which activations inside the code region stack trace are executed. We can define a new data structure called dynamic code region call graph (DRG) which is used for recording the calling behavior and performance metrics of code regions:

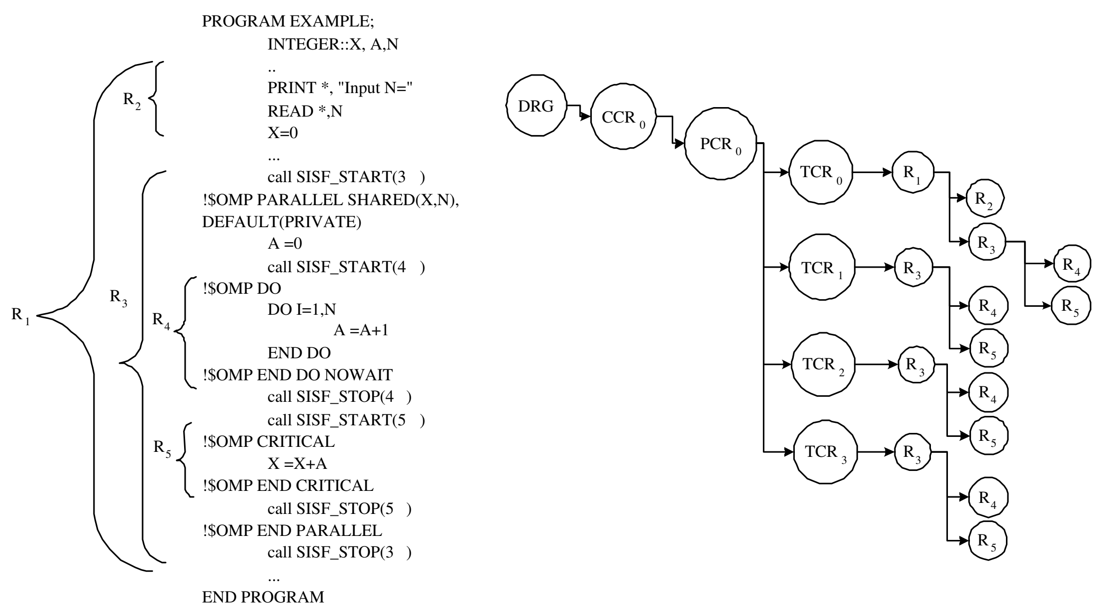
A dynamic code region call graph (DRG) of a program Q with a set of code regions $R = \{r_1, r_2, ..., r_n\}$ is defined by a directed flow graph $G = (N, E, s)$ with a set of nodes N and a set of edges E. A node $n \in N$ represents a set of activations of a code region $r_k \in R$ which is executed at least once during runtime of Q. The equivalence relation holds for all code region stack traces of all activations in n. An edge $(n_1, n_2) \in E$ is a pair of $n_1, n_2 \in N$ where n_1 and n_2 are a set of activations of code region r_p and r_q, respectively and activations in n_2 is called directly by activations in n_1. The set of activations of the first code region executed during execution of Q is defined by s.

For example, Figure 4 shows an excerpt of an OpenMP code together with its associated DRG.

The DRG is stimulated by the idea of the calling context tree (CCT)[2]. However, the DRG differs CCT in several aspects:

- A node in the CCT represents activations at procedure-level whereas in the DRG a node is defined as set of activations of an arbitrary code region (e.g. function, loop, statement).
- The DRG provides the context of parallel program. Calling context associates with not only call path of code regions but also computational calling environments (information about computational nodes, processes, threads).

The DRG is used as a key data structure to conduct a detailed performance overhead analysis under SCALEA. Notice that the generic timing overhead of a code region r with n explicitly



```
PROGRAM EXAMPLE;
         INTEGER::X, A,N
         ..
         PRINT *, "Input N="
         READ *,N
         X=0
         ...
         call SISF_START(3  )
!$OMP PARALLEL SHARED(X,N),
DEFAULT(PRIVATE)
         A =0
         call SISF_START(4  )
!$OMP DO
         DO I=1,N
                  A =A+1
         END DO
!$OMP END DO NOWAIT
         call SISF_STOP(4  )
         call SISF_START(5  )
!$OMP CRITICAL
         X =X+A
!$OMP END CRITICAL
         call SISF_STOP(5  )
!$OMP END PARALLEL
         call SISF_STOP(3  )
         ...
END PROGRAM
```

Figure 4. OpenMP code excerpt with DRG when executing with 4 threads (denoted by TCR_0–TCR_3). Code region R_1, R_2 are executed only in thread 0 whereas R_3, R_4, R_5 are executed in all threads. This program is running with 1 process (denoted by PCR_0) on a computational node (denoted by CCR_0).

instrumented sub-regions $r_1, ..., r_n$ is given by

$$T(r) = T(Start_r) + T(r_1) + ... + T(r_n) + T(Remain) + T(End_r)$$

where $T(r_i)$ is the timing overhead for an explicitly instrumented code region r_i $(1 \leq i \leq n)$. $T(Start_r)$ and $T(End_r)$ correspond to the overhead at the beginning (e.g. fork threads, redistribute data) and at the end (join threads, barrier synchronization, process reduction operation, etc.) of r. $T(Remain)$ corresponds to the code regions that have not been explicitly instrumented. However, we can easily compute $T(Remain)$ as region r is instrumented as well.

5.2. Generating and Building the Dynamic Code Region Call Graph

Calling code region r_2 inside a code region r_1 during the execution of a program establishes a parent-children relationship between executions of r_1 and r_2. The instrumentation library will capture these relationships, build nodes of the DRG and maintain them during the execution of the program. Each node in the DRG is represented by a data entry point which contains performance measurement for all activations whose code region stack traces are equivalent. An edge (n_1, n_2) of the DRG is represented by a link from the caller n_1 to the called node n_2. In our implementation, a global pointer per thread is used to maintain the current node (data entry point) of the sub-DRG of the thread. The performance measurement of the current activation will be stored into the corresponding data entry point maintained by the global pointer.

If a new activation of code region r_2 starts to be executed inside an activation of code region r_1, the instrumentation library then searches all children of the current data entry point maintained by the global pointer. If no child representing activations of r_2 is found, then a new data entry point is generated for holding information of the new activation and an edge linking the current entry point to the new data entry point is made. Otherwise, there exists a previous data entry point representing activations of r_2; code region stack traces of these activations are equivalent with that of the new activation. In this case, the existing data entry point is used to keep information of the new activation. In both cases, the global pointer is then pointed to the new or existing data entry point which holds information of the new activation. When the new activation ends its execution, the data entry point maintained by the global pointer will be updated with performance measurement. And then, the global pointer is transfered to the parent of the current data entry point. If a code region r (e.g. the code region that is executed first) is encountered that it is not a child of any other code region, an abstract code region is assigned as its parent. Every code region has a unique identifier which is included in the probe inserted by SIS and stored in the instrumentation description file.

The DRG data structure maintains the information of code regions that are instrumented and executed. Each thread of individual process will build and maintain its own sub-DRG when executing. In the post-processing phase, the DRG of the entire application will be constructed based on the individual sub-DRGs of all threads by processing the profiles/trace files that contain the performance data of threads.

6. PERFORMANCE DATA REPOSITORY

A key concept of SCALEA is to store the most important information about performance experiments including application, source code, machine information, and performance results in a data repository. The reasons for utilizing a data repository are manifold. Firstly, we need to structure the data associated with performance experiments thus performance results can always be associated with their source codes and machine description on which the experiment has been taken. Secondly, any other performance tool can store its performance data for a given application to the same repository thus providing a large potential to enable more sophisticated performance analysis. Thirdly, other tools or middleware – such as higher level performance tools [10], performance modeling and prediction tools [26], or middleware for distributed systems [12], etc. – can easily access the performance data through a well-defined interface (e.g. JDBC [35]).

Figure 5 shows the structure of the data stored in SCALEA's performance data repository. 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 portion of a machine on which the code has been executed, and performance information. An application (program) may have a number of implementations (code versions). Every implementation consists of a set of source files and

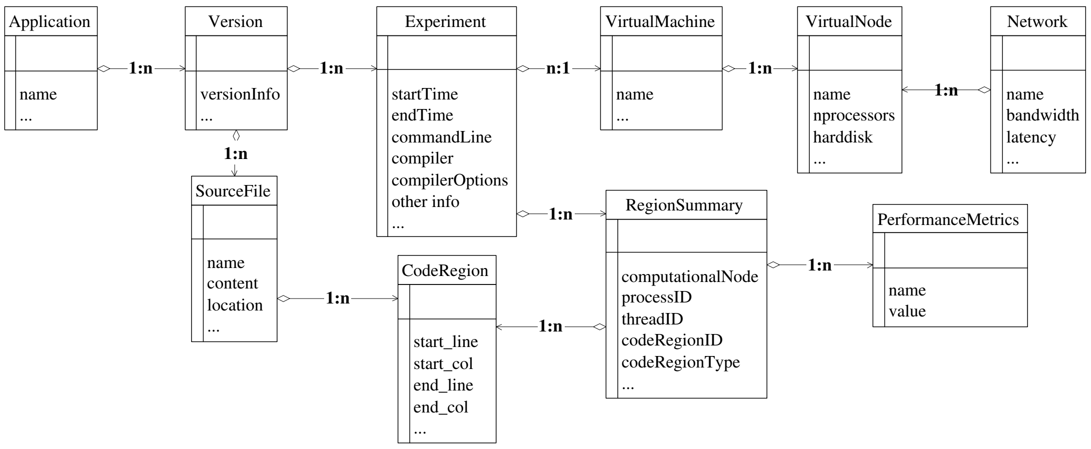


Figure 5. SCALEA Performance Data Repository.

it is associated with one or several experiments. A source file contains one or several static code regions (ranging from entire program units to single statements), each of them is uniquely specified by *startPos* and *endPos* (position – start/end line and column – where the region begins and ends in the source file). Experiments are associated with a virtual machine on which they have been taken. The virtual machine is part of a physical machine available to the experiment at the time of its execution; it is described as a set of computational nodes (e.g. single-processor systems, Symmetric Multiple Processor (SMP) nodes sharing a common memory, etc.) connected by a specific network. Specific data of physical machines such as memory capacity, peak FLOPS are also measured and stored in the data repository. A region summary refers to the performance information collected for a given code region and processing unit (process or thread) on a specific virtual node used by the experiment. The region summaries are associated with performance metrics that comprise performance overheads, timing information, and hardware parameters. Moreover, most data can be exported in XML format which further facilitates accessing performance information by other tools (e.g. compilers or runtime systems), middleware, and applications.

7. EXPERIMENTS

SCALEA as shown in Fig. 1 has been fully implemented. However, some performance overheads (see Fig. 2) are not yet supported which includes replicated code, algorithm change, compiler change, implicit barrier operations, scheduling, and communication overhead of reduction operation. Our analysis and visualization system are implemented in Java which greatly improves their portability. The performance data repository uses PostgreSQL [27] which is a public source relational database and the interface between SCALEA and the data repository is realized by Java and JDBC. SCALEA also provides well-defined interfaces for other tools to exploit SCALEA's analysis, visualization and data repository features.

SCALEA provides several analyses for single-experiment (e.g. *Load Imbalance Analysis, Inclusive/Exclusive Analysis, Metric Ratio Analysis, Overhead Analysis, Summary Analysis*) and multi-experiment (e.g. *Speedup/Improvement Analysis, Scalability Analysis, Multi-Region Analysis, Multi-Set Experiment Analysis*). In this section, we restrict the experiments shown in this section to a few selected features for post-mortem performance analysis. The experiments have been conducted on an SMP cluster with 16 SMP nodes (connected by Myrinet and Fast-Ethernet) each of which comprises 4 Intel Pentium III 700 MHz CPUs.

7.1. Inclusive/Exclusive Analysis

SCALEA's **Inclusive/Exclusive Analysis** can be used to determine the execution time or overhead intensive code regions. Each code region is related to a set of threads and processes in which the code region is executed. For each code region instance various user-selected metrics can be displayed in the inclusive/exclusive mode.

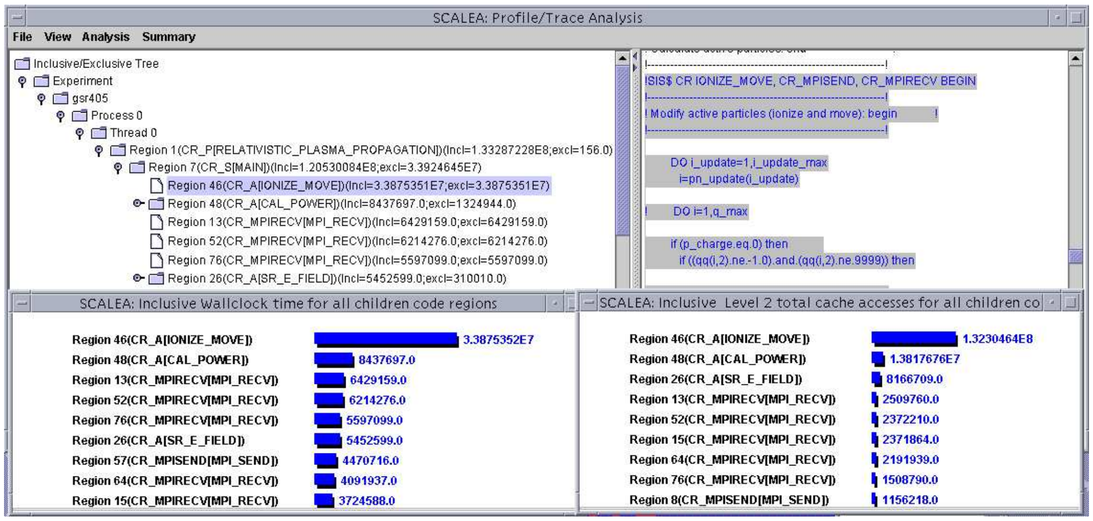
We illustrate the use of this analysis for an MPI Fortran named 3D Particle-In-Cell (3DPIC)[13] which simulates the interaction of high intensity ultrashort laser pulses with plasma in three dimensional geometry. An experiment is conducted on 3SMP nodes with 4 CPUs per node using the MPICH communication library for Fast-Ethernet 100Mbps. The problem size (3D geometry) has been fixed with 30 cells in x-direction (*nnx_glob*=30), 30 cells in y-direction (*nny_glob*=30), and 100 cells in z-direction (*nnz_glob*=100). The simulation has been done for 800 time steps (*itmax*=800).

The two lower-windows in Fig. 6 present the inclusive wallclock times and the number of L2 cache accesses for sub-regions of the subroutine *MAIN* executed by thread 0 in process 0 of SMP node gsr405. The most time consuming region is *IONIZE_MOVE* and its related source code is displayed in the upper-right window. It is shown that specific MPI routines also consume considerably wallclock. The time spent in three top time-consuming MPI routines is larger than a half of that spent in the most time-consuming code region *IONIZE_MOVE*.

7.2. Metric Ratio Analysis

SCALEA's **Metric Ratio Analysis** is used to examine various important metric ratios (e.g. cache miss ratio, system time/wall clock time, floating point instructions per second) of code region instance(s) in an experiment. SCALEA supports not only built-in metric ratios (e.g. L2 cache miss ratio, floating point instructions per second) but also user-defined metric ratios. The user just selects performance metrics availability for code regions and graphically defines the metric ratio and then SCALEA will compute the metric ratio. Metric ratios of a given code region can be compared among its instances inside a thread or a process or a computational node.

We applied this analysis to the abovementioned experiment of 3DPIC. Figure 7 shows the most critical system time/wall clock time and L2 cache misses/L2 cache accesses ratios together with the corresponding code regions. The code regions


SCALEA: Profile/Trace Analysis

File View Analysis Summary

```
Inclusive/Exclusive Tree
  Experiment
    gsr405
      Process 0
        Thread 0
          Region 1(CR_P[RELATIVISTIC_PLASMA_PROPAGATION])(Incl=1.33287228E8;excl=156.0)
            Region 7(CR_S[MAIN])(Incl=1.20530084E8;excl=3.3924645E7)
              Region 46(CR_A[IONIZE_MOVE])(Incl=3.3875351E7;excl=3.3875351E7)
              Region 48(CR_A[CAL_POWER])(Incl=8437697.0;excl=1324944.0)
              Region 13(CR_MPIRECV[MPI_RECV])(Incl=6429159.0;excl=6429159.0)
              Region 52(CR_MPIRECV[MPI_RECV])(Incl=6214276.0;excl=6214276.0)
              Region 76(CR_MPIRECV[MPI_RECV])(Incl=5597099.0;excl=5597099.0)
              Region 26(CR_A[SR_E_FIELD])(Incl=5452599.0;excl=310010.0)
```

```
!---------------------------------------------------------!
!SIS$ CR IONIZE_MOVE, CR_MPISEND, CR_MPIRECV BEGIN
!---------------------------------------------------------!
! Modify active particles (ionize and move): begin       !
!---------------------------------------------------------!

      DO i_update=1,i_update_max
        i=pn_update(i_update)

!      DO i=1,q_max

      if (p_charge.eq.0) then
        if ((qq(i,2).ne.-1.0).and.(qq(i,2).ne.9999)) then
```

SCALEA: Inclusive Wallclock time for all children code regions

Region 46(CR_A[IONIZE_MOVE])	3.3875352E7
Region 48(CR_A[CAL_POWER])	8437697.0
Region 13(CR_MPIRECV[MPI_RECV])	6429159.0
Region 52(CR_MPIRECV[MPI_RECV])	6214276.0
Region 76(CR_MPIRECV[MPI_RECV])	5597099.0
Region 26(CR_A[SR_E_FIELD])	5452599.0
Region 57(CR_MPISEND[MPI_SEND])	4470716.0
Region 64(CR_MPIRECV[MPI_RECV])	4091937.0
Region 15(CR_MPIRECV[MPI_RECV])	3724588.0

SCALEA: Inclusive Level 2 total cache accesses for all children co

Region 46(CR_A[IONIZE_MOVE])	1.3230464E8
Region 48(CR_A[CAL_POWER])	1.3817676E7
Region 26(CR_A[SR_E_FIELD])	8166709.0
Region 13(CR_MPIRECV[MPI_RECV])	2509760.0
Region 52(CR_MPIRECV[MPI_RECV])	2372210.0
Region 15(CR_MPIRECV[MPI_RECV])	2371864.0
Region 64(CR_MPIRECV[MPI_RECV])	2191939.0
Region 76(CR_MPIRECV[MPI_RECV])	1508790.0
Region 8(CR_MPISEND[MPI_SEND])	1156218.0

Figure 6. Inclusive/Exclusive analysis for sub-regions of the *MAIN* program.

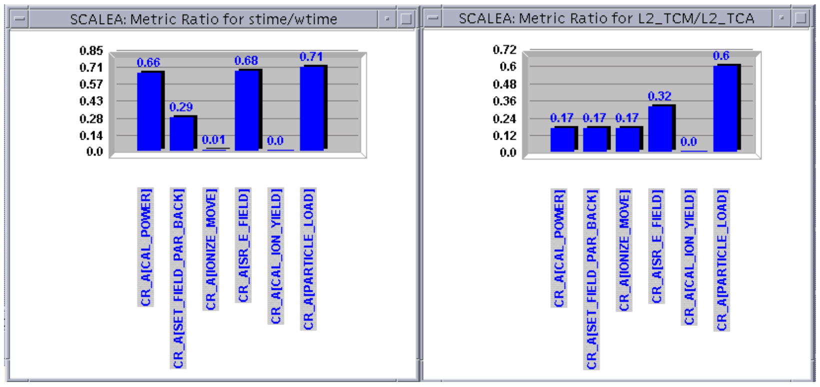


Figure 7. Metric ratios for important code regions. The left-window shows system time/wall clock time ratio and the right-window displays L2 cache misses/L2 cache accesses ratio.

CAL_POWER,*SET_FIELD_PAR_BACK*, *SR_E_FIELD*, and *PARTICLE_LOAD* imply a high system time/wall clock time ratio due to expensive MPI constructs (included in system time). Both ratios are rather low for region *IONIZE_MOVE* because this region represents the computational part without any communication (mostly user time). The code region *PARTICLE_LOAD* shows a very high L2 cache misses/L2 cache accesses ratio because it initializes all particles in the 3D volume without accessing it again (little cache reuse).

7.3. Execution Summary Analysis

SCALEA's **Execution Summary Analysis** displays a break-down of varies performance metrics (e.g execution time, overheads) for a single experiment. For each experiment, summary of a given performance metric for computational nodes is shown. The summary for computational nodes can then be broken down into that for processes.

For instance, SCALEA's **Execution Summary Analysis** has been employed to examine the impact of communication on the execution time of 3DPIC. Figure 8 and 9 depict the *execution time summary* for the experiment executed with 3 SMP nodes and with 1 SMP node (4 processors per node), respectively. In the top window of Fig. 8 each pie represents one SMP node and each pie slice value corresponds to the average value across all processes of an SMP node (min/max/average values can be selected). By clicking onto an SMP pie, SCALEA displays a detailed summary for all processes in this node in the three lower windows of Fig. 8. Each pie is broken down into time spent in MPI routines and the remainder. Clearly, with the given problem size, SCALEA indicates the dramatic increase of communication time when increasing number of SMP nodes.

7.4. Overhead Analysis for a Single Experiment

The *Overhead Analysis* is used to investigate performance overheads of an experiment based on our overhead classification.

Performance overheads of code region instances of a given experiment are computed, displayed and stored into the performance data repository. We illustrate the use of this analysis with a mixed OpenMP/MPI Fortran program that solves the 2d Stommel model [29] of an ocean circulation using a five-point stencil and Jacobi iteration.

SCALEA supports the user in the effort to examine the performance overheads for a single experiment of a given program by providing two modes for this analysis. Firstly, the *Region-to-Overhead* mode (see the "Region-to-Overhead" window in Fig. 10) allows the user to select any code region instance in the DRG for which all detected performance overheads are displayed. Secondly, the *Overhead-to-Region* mode (see the "Overhead-to-Region" window in Fig. 10) enables the user to select the performance overhead of interest, based on which SCALEA displays the corresponding code region(s) in which this overhead occurs. This selection can be limited to a specific code region instance, thread or process. For both modi the source code of a region is shown if the code region instance is selected in the DRG by a mouse click.

7.5. Multiple Experiments Analysis

Most performance tools investigate the performance for individual experiments one at a time. SCALEA goes beyond this limitation by supporting also performance analysis for multiple experiments. The user can select several experiments, code regions and performance metrics of interest whose associated data are stored in the data repository (see Figure 11). The outcome of every selected metric is then analyzed and visualized for all experiments. In this section we demonstrate multi-experiment analyses applied to LAPW0 [5] which is a material science program that calculates the effective potential of the Kohn-Sham eigen-value problem. LAPW0 has been implemented as a Fortran90 MPI code.

Multi-experiment analysis is not only based on a single set of experiments but can also be applied to compare different sets of experiments. The user can analyze the overall execution of the application across various sets of experiments; experiments are grouped based on their

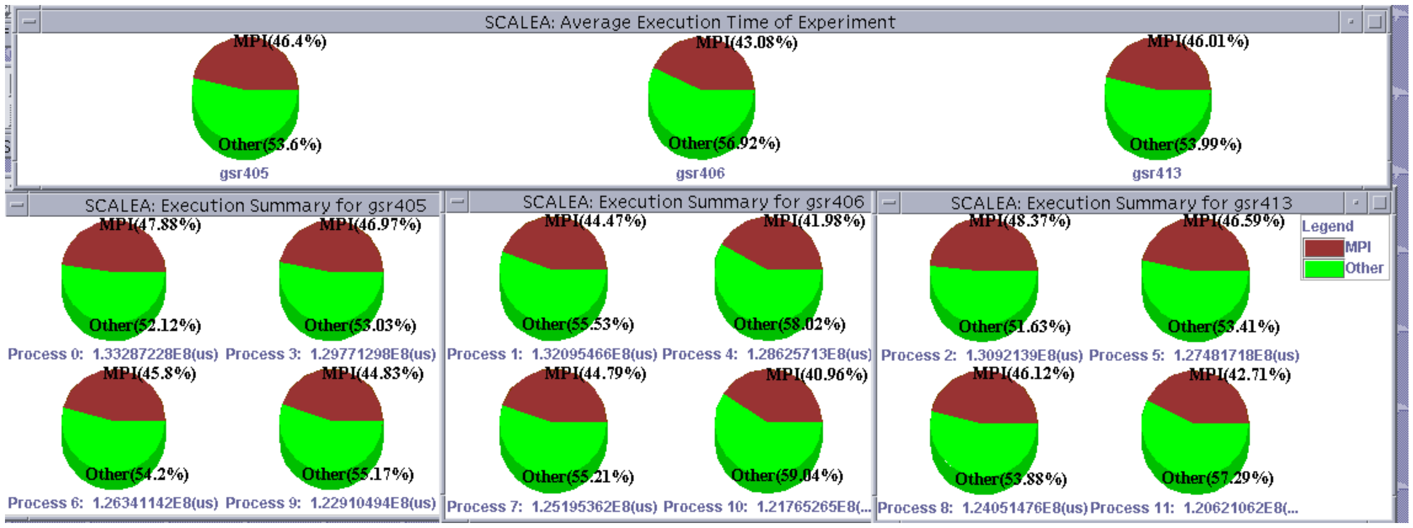

SCALEA: Average Execution Time of Experiment
MPI(46.4%)
Other(53.6%)
gsr405
MPI(43.08%)
Other(56.92%)
gsr406
MPI(46.01%)
Other(53.99%)
gsr413
SCALEA: Execution Summary for gsr405
MPI(47.88%)
Other(52.12%)
Process 0: 1.33287228E8(us)
MPI(46.97%)
Other(53.03%)
Process 3: 1.29771298E8(us)
MPI(45.8%)
Other(54.2%)
Process 6: 1.26341142E8(us)
MPI(44.83%)
Other(55.17%)
Process 9: 1.22910494E8(us)
SCALEA: Execution Summary for gsr406
MPI(44.47%)
Other(55.53%)
Process 1: 1.32095466E8(us)
MPI(41.98%)
Other(58.02%)
Process 4: 1.28625713E8(us)
MPI(44.79%)
Other(55.21%)
Process 7: 1.25195362E8(us)
MPI(40.96%)
Other(59.04%)
Process 10: 1.21765265E8(...
SCALEA: Execution Summary for gsr413
MPI(48.37%)
Other(51.63%)
Process 2: 1.3092139E8(us)
MPI(46.59%)
Other(53.41%)
Process 5: 1.27481718E8(us)
MPI(46.12%)
Other(53.88%)
Process 8: 1.24051476E8(us)
MPI(42.71%)
Other(57.29%)
Process 11: 1.20621062E8(...
Legend
MPI
Other

Figure 8. Execution time summary for an experiment with 3 SMP-nodes.

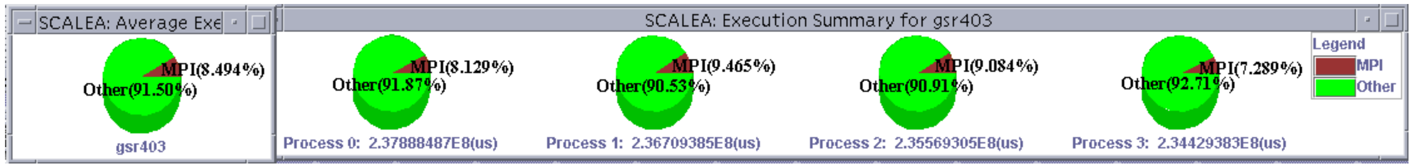
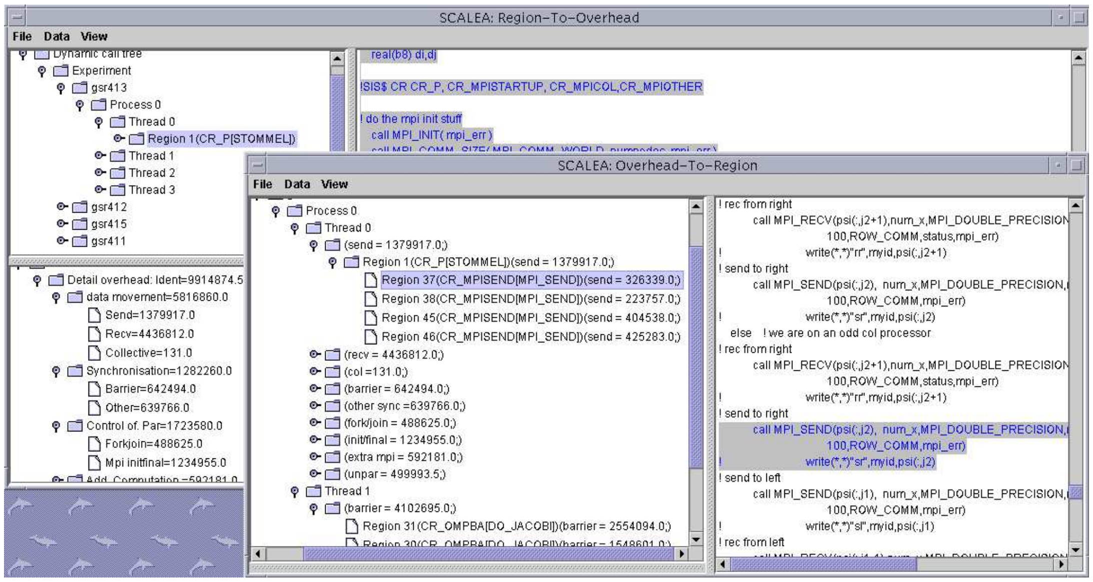


Figure 9. Execution time summary for an experiment with 1 SMP-node.



```
Dynamic call tree
  Experiment
    gsr413
      Process 0
        Thread 0
          Region 1(CR_P[STOMMEL])
        Thread 1
        Thread 2
        Thread 3
    gsr412
    gsr415
    gsr411
```

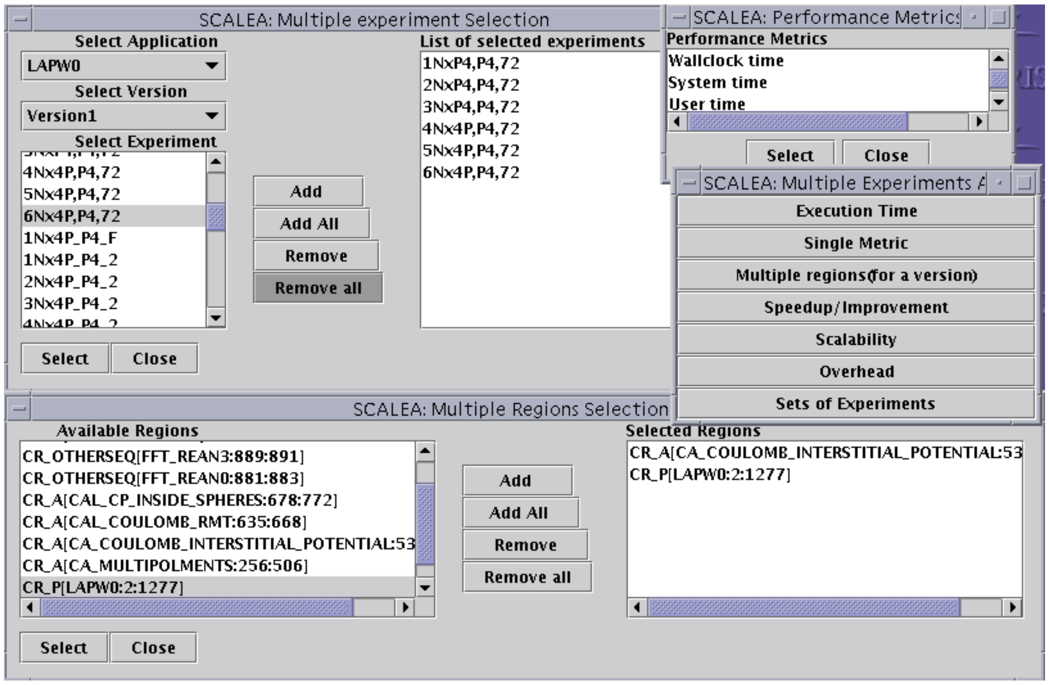
```
Detail overhead: Ident=9914874.5
  data movement=5816860.0
    Send=1379917.0
    Recv=4436812.0
    Collective=131.0
  Synchronisation=1282260.0
    Barrier=642494.0
    Other=639766.0
  Control of. Par=1723580.0
    Forkjoin=488625.0
    Mpi initfinal=1234955.0
  Add. Computation =592181.0
```

```
  real(b8) di,dj

!SIS$ CR CR_P, CR_MPISTARTUP, CR_MPICOL,CR_MPIOTHER

! do the mpi init stuff
  call MPI_INIT( mpi_err )
```

```
Process 0
  Thread 0
    (send = 1379917.0;)
      Region 1(CR_P[STOMMEL])(send = 1379917.0;)
        Region 37(CR_MPISEND[MPI_SEND])(send = 326339.0;)
        Region 38(CR_MPISEND[MPI_SEND])(send = 223757.0;)
        Region 45(CR_MPISEND[MPI_SEND])(send = 404538.0;)
        Region 46(CR_MPISEND[MPI_SEND])(send = 425283.0;)
    (recv = 4436812.0;)
    (col =131.0;)
    (barrier = 642494.0;)
    (other sync =639766.0;)
    (fork/join = 488625.0;)
    (init/final = 1234955.0;)
    (extra mpi = 592181.0;)
    (unpar = 499993.5;)
  Thread 1
    (barrier = 4102695.0;)
      Region 31(CR_OMPBA[DO_JACOBI])(barrier = 2554094.0;)
      Region 30(CR_OMPBA[DO_JACOBI])(barrier = 1548601.0;)
```

```
! rec from right
        call MPI_RECV(psi(:,j2+1),num_x,MPI_DOUBLE_PRECISION
                        100,ROW_COMM,status,mpi_err)
!                write(*,*)"rr",myid,psi(:,j2+1)
! send to right
        call MPI_SEND(psi(:,j2),  num_x,MPI_DOUBLE_PRECISION,
                        100,ROW_COMM,mpi_err)
!                write(*,*)"sr",myid,psi(:,j2)
  else   ! we are on an odd col processor
! rec from right
        call MPI_RECV(psi(:,j2+1),num_x,MPI_DOUBLE_PRECISION
                        100,ROW_COMM,status,mpi_err)
!                write(*,*)"rr",myid,psi(:,j2+1)
! send to right
        call MPI_SEND(psi(:,j2),  num_x,MPI_DOUBLE_PRECISION,
                        100,ROW_COMM,mpi_err)
!                write(*,*)"sr",myid,psi(:,j2)
! send to left
        call MPI_SEND(psi(:,j1),  num_x,MPI_DOUBLE_PRECISION,
                        100,ROW_COMM,mpi_err)
!                write(*,*)"sl",myid,psi(:,j1)
! rec from left
```

Figure 10. Region-To-Overhead and Overhead-To-Region DRG View.



Select Application: LAPW0

Select Version: Version1

Select Experiment:
3Nx4P,P4,72
4Nx4P,P4,72
5Nx4P,P4,72
6Nx4P,P4,72
1Nx4P_P4_F
1Nx4P_P4_2
2Nx4P_P4_2
3Nx4P_P4_2
4Nx4P_P4_2

List of selected experiments:
1NxP4,P4,72
2NxP4,P4,72
3NxP4,P4,72
4Nx4P,P4,72
5Nx4P,P4,72
6Nx4P,P4,72

Performance Metrics:
Wallclock time
System time
User time

Available Regions:
CR_OTHERSEQ[FFT_REAN3:889:891]
CR_OTHERSEQ[FFT_REAN0:881:883]
CR_A[CAL_CP_INSIDE_SPHERES:678:772]
CR_A[CAL_COULOMB_RMT:635:668]
CR_A[CA_COULOMB_INTERSTITIAL_POTENTIAL:53
CR_A[CA_MULTIPOLMENTS:256:506]
CR_P[LAPW0:2:1277]

Selected Regions:
CR_A[CA_COULOMB_INTERSTITIAL_POTENTIAL:53
CR_P[LAPW0:2:1277]

Figure 11. Multiple Experiments Analysis.

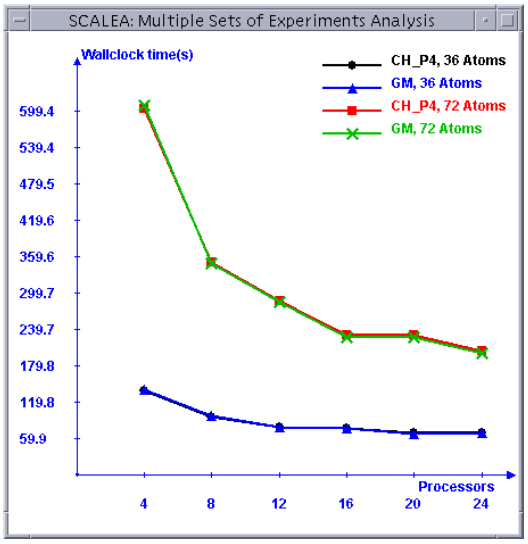
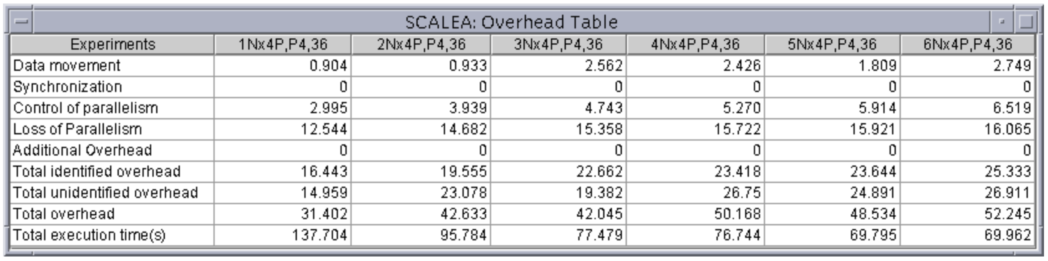


Figure 12. Execution time of LAPW0 with 36 and 72 atoms. *CH_P4, GM* means that MPICH has been used for CH_P4 (for Fast-Ethernet 100Mbps) and Myrinet, respectively.


SCALEA: Overhead Table

Experiments	1Nx4P,P4,36	2Nx4P,P4,36	3Nx4P,P4,36	4Nx4P,P4,36	5Nx4P,P4,36	6Nx4P,P4,36
Data movement	0.904	0.933	2.562	2.426	1.809	2.749
Synchronization	0	0	0	0	0	0
Control of parallelism	2.995	3.939	4.743	5.270	5.914	6.519
Loss of Parallelism	12.544	14.682	15.358	15.722	15.921	16.065
Additional Overhead	0	0	0	0	0	0
Total identified overhead	16.443	19.555	22.662	23.418	23.644	25.333
Total unidentified overhead	14.959	23.078	19.382	26.75	24.891	26.911
Total overhead	31.402	42.633	42.045	50.168	48.534	52.245
Total execution time(s)	137.704	95.784	77.479	76.744	69.795	69.962

Figure 13. Performance overheads for LAPW0.

properties (e.g. the same problem sizes and communication library). For example, we use this analysis to study the performance of LAPW0 for two problem sizes and six machine sizes with two different network configurations as shown in Fig. 12. Based on this study, we observed that changing the communication network from Fast-Ethernet by Myrinet did not actually improve the performance.

Overhead Analysis also can be applied to multiple experiments. SCALEA provides a **Performance Overhead Summary** to examine various sources of performance overheads across experiments. For example, the overhead summary for LAPW0 with problem size of 36 atoms displayed in Fig. 13 uncovers a small amount of data movement overhead but a large of overhead for loss of parallelism and unidentified overhead. As a result, instead of focusing our effort on analyzing code regions that are sources of data movement (e.g. send/receive), we study code regions that possibly cause loss of parallelism overhead.

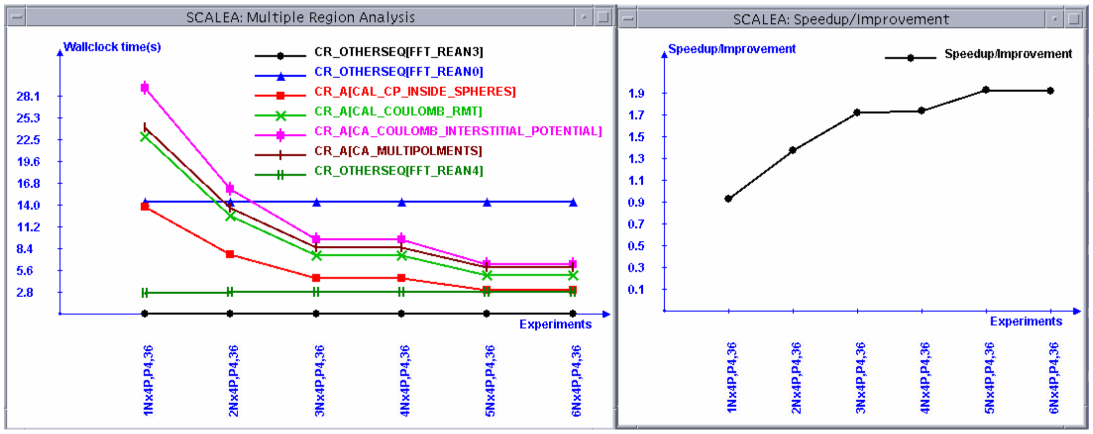


Figure 14. Execution time of computationally intensive code regions (left window) and program's speedup/improvement (right window). *1Nx4P,P4,36* means 1 SMP node with 4 processors using MPICH CH_P4 and the problem size is 36 atoms.

Multi-experiment analysis normally applied to the entire program is useful for examining the scalability of applications, however, it does not reveal the details of the scalability of code regions. In order to support studying the performance behavior of selected code regions, SCALEA provides a **Multiple Region Analysis**. For instance, the left-window of Fig. 14 visualizes the execution times for the most computational intensive code regions in LAPW0. The right-window of Fig. 14 displays the program's speedup/improvement behavior. The execution times of code regions including CAL_CP_INSIDE_SPHERES, CAL_COULOMB_RMT, CA_COULOMB_INTERSTITIAL_POTENTIAL, CA_MULTIPOLMENTS remain almost constant although the number of processors is increased from 12 to 16 and 20 to 24. In addition, code regions FFT_REAN0, FFT_REAN3, and FFT_REAN4 are executed sequentially. These code regions should therefore be subject of parallelization in order to gain performance.

In summary, we believe that multiple experiments analysis feature is very useful for scalability analysis of individual performance metrics and code regions for changing problem and machine sizes and underlying computing resources.

8. RELATED WORK

Significant work has been done by Paradyn [22, 31], TAU [21], VAMPIR [24, 14], Pablo toolkit [28], Paraver [33], and EXPERT [9].SCALEA differs from these approaches by providing a more flexible mechanism to control instrumentation of code regions and performance metrics of interest. Although Paradyn enables dynamic insertion of probes into a running code, its analysis is limited to procedures and procedure calls whereas SCALEA can instrument - at source level only - arbitrary code regions including single statements. While TAU and VAMPIR provide begin/end marker routines which allow measuring arbitrary code regions, this feature is instrumented manually by the user. Therefore, the user has to deal with code regions individually, spending a great care in processing multiple-entry multiple-exit code regions. SCALEA also provides more high-level performance metrics (e.g. data movement and control of parallelism overhead) than TAU and VAMPIR do. Moreover, SCALEA differs in many ways from the aforementioned tools by storing experiment-related data into a data repository, by providing multiple instrumentation options (directives, command-line options, and high-level AST instrumentation), and by supporting multi-experiment performance analysis.

Call graph techniques have widely been used in performance analysis. Tools such as VAMPIR, gprof [11], CXperf [17] support a call graph which shows how much time was spent in each code region and its children. In [8] a call graph is used to improve the search strategy for automated performance diagnosis. Our DRG requires space less than the dynamic call tree (each node represents a single activation of a code region, e.g. VAMPIR) and provides information more precisely than the dynamic call graph (each node represents all activations of a code region, e.g. gprof, CXperf). In addition, nodes of the call graph in gprof and CXperf represent function calls. In contrast our DRG defines a node as an arbitrary code region (e.g. function, function call, loop, statement).

In [19], information about each experiment is stored in a Program Event and techniques for comparison between experiments are done automatically. A prototype of Program Event has been implemented in Paradyn. However, the lack of capability to export and share performance data has hindered external tools from using and exploiting data in Program Events.

Prophesy [30] provides a performance data repository. Prophesy uses performance data to perform the automatic generation of performance models. Data measured/analyzed by SCALEA could be used by Prophesy for modeling systems.

USRA Tool family [25] collects and combines information of parallel programs from various sources at level of subroutines and loops. Information is stored in flat files which can further be saved in a format understood by spreadsheet programs. SCALEA's repository provides a better infrastructure for storing, querying and exporting performance data with a relational database system.

Recent work on an OpenMP performance interface [23] is based on directive rewriting which is similar to the SIS instrumentation approach. A general method for instrumenting OpenMP codes is presented. Directives are introduced to mark code regions and to control performance data collection. SCALEA's instrumentation directives are more flexible, as they allow to the user to specify for what code regions which performance metrics should be determined.

9. CONCLUSION AND FUTURE WORK

In this paper, we described SCALEA which is a performance analysis tool for OpenMP/MPI/HPF and mixed parallel programs. We have described the design of SCALEA which includes a new classification of performance overheads for shared and distributed memory parallel programs. The overhead classification provides a detailed view of the sources of performance overheads for parallel programs. A highly flexible instrumentation system can be used to instrument arbitrary code regions and to enable the programmer to request for a large variety of performance metrics ranging from timing information and hardware parameters to performance overheads. SCALEA also provides a high-level instrumentation interface based on an abstract syntax tree and an unparser provided by an external Fortran90 compiler. SCALEA's instrumentation interface can be used by other tools for instrumenting parallel and distributed Fortran programs.

A performance data repository which holds all relevant information about applications, experiments, and performance experiments is introduced. The data repository is capable of exporting experiment data in XML format and has leveraged the collaboration among performance analysis tools and high-level tools by providing a well-defined data schema and interface for accessing information stored in repository. A variety of single- and multi-experiment performance analyses is provided based on the data stored in the data repository and a novel representation for code regions named dynamic code region call graph (DRG). The DRG reflects the dynamic relationship between code regions and its subregions and enables a detailed overhead analysis for every code region. The DRG is not restricted to function calls but covers every code regions ranging from single statements to entire program units.

SCALEA is part of the ASKALON programming environment and tool set for cluster and Grid architectures [3]. SCALEA is used by various other tools in ASKALON to support automatic bottleneck analysis, performance experiment and parameter studies, and performance prediction.

Currently, SCALEA is extended towards a unified system for online monitoring and performance analysis for the Grid. The overhead classification will be extended to cover the Grid fabric, services, middleware and applications. We also work on the dynamic instrumentation and online performance analysis for the Grid. Moreover, we investigate possibilities to extend SCALEA for C/C++, Java programs and other programming paradigms such as component-based model.

REFERENCES

1. G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference*, pages 483–485, 1967.
2. Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 85–96, 1997.
3. ASKALON - A Programming Environment and Tool Set for Cluster and Grid Computing. http://www.par.univie.ac.at/project/askalon. Institute for Software Science, University of Vienna.
4. S. Benkner. VFC: The Vienna Fortran Compiler. *Scientific Programming, IOS Press, The Netherlands*, 7(1):67–81, 1999.
5. 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.
6. 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.
7. J.M. Bull. A Hierarchical classification of Overheads in Parallel Programs. In P. Croll I. Jelly, I. Gorton, editor, *Proceedings of Firs IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 208–219. Chapman Hall, March 1996.
8. Harold W. Cain, Barton P. Miller, and Brian J.N. Wylie. A Callgraph-Based Search Strategy for Automated Performance Diagnosis. In *Euro-Par 2000 Parallel Processing*, pages 108–122, 2000.
9. F. Wolf and B. Mohr. Automatic Performance Analysis of SMP Cluster Applications. Technical Report Tech. Rep. IB 2001-05, Research Center Juelich, 2001.
10. Thomas Fahringer and Clovis Seragiotto. Modeling and Detecting Performance Problems for Distributed and Parallel Programs with JavaPSL. In *Proceeding SC'2001, Denver, USA*, November 2001.

11. Jay Fenlason and Richard Stallman. *GNU gprof.* Free Software Foundation, Inc., September 1997.
12. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *IEEE Computer*, pages 37–46, June 2002.
13. M. Geissler. *Interaction of High Intensity Ultrashort Laser Pulses with Plasmas.* PhD thesis, Vienna University of Technology, 2001.
14. Pallas GmbH. VAMPIR: Visualization and Analysis of MPI Programs. http://www.pallas.com/e/products/vampir/index.htm.
15. William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface.* Scientific and Engineering Computation. MIT Press, Cambridge, MA, USA, November 1999.
16. Robert J. Hall. Call Path Refinement Profiles. *IEEE Transactions on Software Engineering*, 21(6):481–496, June 1995.
17. Hewlett Packard. *CXperf User's Guide*, June 1998.
18. High Performance Fortran Forum. High Performance Fortran Language Specification: Version 2.0. Technical report, Rice University, Houston,TX, January 1997.
19. Karen L. Karavanic and Barton P. Miller. Experiment Management Support for Performance Tuning. In *Proceedings of Supercomputing'97 (CD-ROM)*, San Jose, CA, November 1997. ACM SIGARCH and IEEE.
20. Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing:design and analysis of parallel algorithms.* Benjamin/Cummings, 1994.
21. Allen Malony and Sameer Shende. Performance technology for complex parallel and distributed systems. In *In G. Kotsis and P. Kacsuk (Eds.), Third International Austrian/Hungarian Workshop on Distributed and Parallel Systems (DAPSYS 2000)*, pages 37–46. Kluwer Academic Publishers, Sept. 2000.
22. 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.
23. Bernd Mohr, Allen Malony, Sameer Shende, and Felix Wolf. Towards a performance tool interface for openmp: An approach based on directive rewriting. In *EWOMP'01 Third European Workshop on OpenMPI*, Sept. 2001.
24. W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12(1):69–80, January 1996.
25. Insung Park, Michael Voss, Brian Armstrong, and Rudolf Eigenmann. Parallel programming and performance evaluation with the URSA tool family. *International Journal of Parallel Programming*, 26(5):541–??, ???? 1998.
26. S. Pllana and T. Fahringer. UML Based Modeling of Performance Oriented Parallel and Distributed Applications. In *Proceedings of the 2002 Winter Simulation Conference*, San Diego, California, USA, December 2002. IEEE.
27. PostgreSQL 7.1.2. http://www.postgresql.org/docs/.
28. D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.
29. H.M. Stommel. The western intensification of wind-driven ocean currents. *Transactions American Geophysical Union*, 29:202–206, 1948.
30. V. Taylor, X. Wu, J. Geisler, X. Li, Z. Lan, R. Stevens, M. Hereld, and Ivan R.Judson. Prophesy:An Infrastructure for Analyzing and Modeling the Performance of Parallel and Distributed Applications. In *Proc.of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC's 2000)*, Pittsburgh, August 2000. IEEE Computer Society Press.
31. Paradyn Parallel Performance Tools. http://www.cs.wisc.edu/paradyn/.
32. 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 of the 9th IEEE/ACM High-Performance Networking and Computing Conference (SC'2001)*, Denver, USA, November 2001.
33. T. Cortes V. Pillet, J. Labarta and S. Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. In *WoTUG-18*, pages 17–31, Manchester, April 1995.
34. OpenMP Website. http://www.openmp.org.
35. Alan R. Williamson and Ceri L. Moran. *Java Database Programming: Servlets & JDBC.* Prentice Hall, 1997.