A Productivity Centered Application Performance Tuning Framework

S. Sbaraglia H. Wen S. Seelam I. Chung G. Cong K. Ekanadham D. Klepacki
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, USA
{*ssbarag, hfwen, sseelam, ihchung, gcong, eknath, klepacki*}*@us.ibm.com*

ABSTRACT

In response to the productivity challenge of the U.S. DARPA HPCS initiative, we have developed a methodology that provides an extremely simple and pain-free interface through which scientists can collect rich performance data from selected parts of an execution, digest the data at a very high level, and plan for improvements. This process can be easily repeated, each time refining the selection of parts of the application and revising the granularity of data collected, until complete insight is gained about bottlenecks. A distinct feature of our approach is that the framework is independent of the features being examined. Recognizing that the features to be examined change with systems/applications and also with depth at which an aspect is being examined, our framework provides an easy interface to continually add new features for examination. Furthermore, many different features can be collected simultaneously and examined in a non-interfering manner. Finally, all this is accomplished without changing the source code in any manner. We believe that this is an ideal platform for building knowledge-based repositories for automatic performance tuning, which is the subject of our future study.

In this paper, we describe our productivity centered framework for application performance tuning. It comprises of three features: an unique source code and binary instrumentation feature, a versatile user-interface that brings all the sophisticated capabilities of the binary instrumentation to the user at a higher level of abstraction, and the functionality to collect different dimensions of performance data. The results of execution are all in terms of source level names and at no point does the scientist needs to worry about low-level details of instrumentation. We believe that it is this ability, of deciphering performance impacts at source level, that leads to high productivity of scientists to understand, direct and tune the behavior of the computing system.

Categories and Subject Descriptors

C.4 [**Computer System Organization**]: Performance of Systems—*Measurement Techniques*

General Terms

Measurement, Performance

Keywords

Performance Tuning, Performance Tool

1. INTRODUCTION

Performance tuning of High Performance Computing (HPC) applications is an iterative task during which the user goes through the following sequence of steps:

1. *selection of performance data to be collected*: In the initial stages of the performance tuning process, users collect performance data at coarser granularity to understand CPU, memory, message-passing communication and I/O characteristics of the application. In the later stages, the granularity and the level of details of the data collected is increased to further examine one or few characteristics of the application that have the most promise in terms of performance improvements. Back of the envelope calculations based on known parameters of the system and the application are often used to evaluate a characteristic in terms of its promise and in detailed examination.

2. *selection of a data collection tool*: Based on the characteristic of promise, the desired granularity, and the required level of detail, users pick an appropriate data collection tool. There exists many tools to pick from but each tool has its own intricate way of instrumenting the application and producing the desired performance data. The challenge here is identifying an appropriate tool, understanding its working behavior, and understanding the necessary modifications to the application so that the tool generates the desired data.

3. *instrumentation of the application*: Each of the tools require some modifications to the application or the manner in which it is being executed, at the least. Some require modifications to the source code, some operate on the binary of the application but must be executed in a particular way, and some require relinking of the application with their libraries. Modifications to the source code often entails a tedious and error-prone process of rewriting parts of the application with calls to the appropriate Application Programming Interfaces (API) of the selected tool, while

ensuring the correctness of the application. In addition such modifications require recompilation of the application. The challenge here is that the effort required, in terms of exploiting the power of the tool, for instrumenting an application varies from tool to tool and also varies depending on the level of details desired.

4. *execution of the instrumented application and generation of performance data*: The instrumented application is executed on the target platform as required by the performance tool. Each performance tool generates performance data in a predefined format that is often specific to the tool. The challenge here is that the data generated by different tools is likely to be in different formats, which makes it difficult to compare performance data across tools in a single unified environment.

5. *visualization and analysis of the collected data*: Each of the existing performance tools offers some kind of visualization tools and presents the collected performance data for further analysis by the user. The two underlying challenges here are not necessarily with any specific tool but they are with the combination of the tools that are often required for tuning. First, when multiple tools are used in the analysis of an application, there exists no support for presenting the data in a centralized manner. Second, the existing tools often render performance data from only one execution of the application, they offer little support for a comparative analysis across multiple executions, which is often performed due to limitations on different performance metrics that can be collected in a single execution.

During the last step, the analysis of the performance data may stimulate the analyst to gather different performance data or to aggregate the collected data differently. The instrumentation is then fine-tuned and all the steps are repeated, but each of these repetitions may potentially require a different tool. This process is continued until a performance bottleneck is successfully isolated, at which point the necessary modifications to the application's source code, runtime environment, or system configuration may be made.

For higher productivity, performance analysts need an efficient and integrated environment which allows them to effectively iterate the performance tuning cycle. Some of the key requirements of such integrated framework are:

1. the environment should free the user from the error-prone process of repeatedly modifying the source code,

2. it should give users the control over the granularity of the instrumentation and the data collection process in order to support profiling of large applications,

3. it should correlate the performance data with the source code, and present the information in an easily understood, interactively-browsable form to speed up the search of performance problems,

4. it should allow easy comparisons of performance data across multiple executions and across multiple granularity.

In response to the productivity challenge of the U.S. DARPA HPCS [1] initiative, we have developed a methodology that addresses the various challenges in the iterative tuning process and goes beyond them to improve user productivity. It simplifies tuning process and it can support automation of the performance tuning cycle. In this paper, we describe our productivity centered platform for application performance tuning. It consists of three core features: (1) a versatile source code, performance metrics, and performance data visualization and analysis GUI, (2) a unique source code and binary instrumentation functionality, and (3) the capabilities to collect various dimensions of performance data (CPU, message passing, threads, memory and I/O) and provide performance data at the granularity of the entire application down to every single instruction executed by the processor and its impact on the state of machine. The combination of these features provides a unified framework for performance analysis that helps speed up and simplify the tuning process.

The remainder of this paper is organized as follows: in Section 2, we describe the architecture of our framework. Section 3 summarizes our binary instrumentation feature. Section 4 describes the GUI, the control center of our infrastructure. Section 5 gives an example to demonstrate how our framework leads to high productivity for scientists to understand and tune their applications. Section 6 discusses the related work. Section 7 presents the concluding remarks.

2. OVERVIEW OF THE PRODUCTIVITY CENTERED FRAMEWORK FOR APPLICATION PERFORMANCE TUNING

Our framework is a versatile and productive environment for performance analysis of sequential and parallel applications. It provides a common platform for IBM's mid-range server offerings, including pSeries, iSeries servers, and Blue Gene systems, on both AIX and Linux. This section presents an overview of our framework and its main characteristics that make it a productivity centered platform for application performance tuning.

The overall structure of our framework is depicted in Figure 1. Our framework provides a versatile source code and performance data visualization and analysis GUI, the biggest rectangle box in the top of the figure. This interface is the control center of the entire infrastructure. It depicts what the user sees and interacts with for instrumentation, execution, visualization, and analysis of the performance data. It displays the source program, a menu for different dimensions of performance data and the visualization for the performance data.

Fundamental to our framework is a powerful binary analysis and instrumentation functionality. Section 3 will describe this in more details.

Another feature of our framework is the capability to collect various dimensions of performance data. From previous studies with a variety of HPC applications [9, 4, 11, 12, 10, 17], we have found that these five dimensions (CPU, memory, message passing, threads and I/O) provide an excellent starting point for a programmer to understand the performance behavior of their applications. The dimensions of performance data provided in our current framework are[1]:

1. CPU (Hardware Counter Data): The hardware coun-

[1]The incorporation of profiling I/O activity is currently in progress.

ters provide comprehensive events that are critical to performance. The events include the usual timing information for each process and thread in a parallel application, as well as CPU/memory information, such as the number of misses at each cache level, the number of floating point instructions executed, the number of load instructions that caused cache or TLB misses etc.

2. Message Passing: The performance information includes the time used by each MPI function call and the size of the messages exchanged. In addition to a “flat” view of the MPI performance, the trace data is collected to show the sequence of communication events on a time-line of the execution.

3. Threads: For parallel applications that use the shared-memory OpenMP programming paradigm, our framework will collect timing of each parallel thread in each parallel region, overhead of the parallel constructs, etc.

4. Memory: The memory performance data helps programmers understand the precise memory references in scientific programs that are causing poor utilization of the memory subsystem. Fine-grained information such as the number of hits and misses in each cache level for each data structure and for each function is useful for tuning loop kernels, understanding the cache behavior of new algorithms, and for investigating how different parts of a program compete for (and interact within) the memory subsystem. The information is presented in a data-centric and control-centric manner which allows the user to identify the data-structures (as specified in the source code) that are responsible for poor memory behavior and further dig down into the functions where these data structures are being accessed.

Most of the performance tools may have their own customized ways of instrumenting the application to produce the desired performance data. The tool may require the modifications of source code in order to monitor the selected regions of user’s applications. In many such cases, users have to relink their applications with the performance tool library. There is no common instrumentation layer among the performance tools. This causes users to spend excess efforts to learn and understand the instrumentation mechanism for each different performance tool. Moreover, each performance tool may generate its own performance data during program execution. Users may need to switch to different visualization tools for rendering various kinds of performance data.

The main contribution of our framework is to provide an environment for application performance tuning. Within the same GUI, users can select the dimensions of performance data, control the granularity of performance data, collect and visualize the performance data. Our framework offers *source code secure*, i.e., it eliminates the user injected bugs and any unintentional changes to source code. In addition, all interactions at the top level by the users are in the language of the source program and all details of binary and instrumented binary, data collected at the binary level are all hidden underneath. Thus, the user can relate the results better with the source program being examined. The results can be saved and a new specification be quickly composed, run and the new results can be compared with the previous results. These are some of the key aspects for productivity.

The main characteristics of our framework are that:

- It offers an integrated environment for simultaneous investigation of a number of aspects of performance (e.g. CPU, memory, threads, message passing).
- It requires no source code modifications. The user interacts with our framework by using symbolic names in the source domain, but all the modifications to the application are transparently performed on the binary. As a result, no recompilation of the application is needed.
- It is able to collect data for CPU, message passing, thread activity and memory, simultaneously in a single run, thereby significantly reducing the time needed to profile the application.
- It presents performance data information in a user-friendly manner that highlights the relation between the performance metrics and the source code statements.
- It supports iterative tuning by allowing the analyst to refine the data collection rules and rerun the experiment results. The results of the new execution can be displayed along with the previous data for direct comparison.
- It allows the user to select the granularity of data collection. For instance, it is possible, to gather information at a function boundary level and later dig down into the details of selected functions.
- Its design allows for easy extension of new dimension of performance data and can therefore be easily expanded to address the needs of different programmers or computing environments.

3. BINARY INSTRUMENTATION IN THE FRAMEWORK

As shown in Figure 1, the binary instrumentation contains two modules, Binary Analysis System (BAS) and Binary Instrumentation System (BIS). BAS provides the program structure of the executable to the visualization system. With the knowledge of the program structure, our framework is able to provide an instructive environment for the instrumentation, control of data collection, and for mapping performance data to the corresponding source code of the application. BIS performs the binary instrumentation by modifying the application executable. Further details of the functionality of BAS and BIS will be described next.

BAS is a module whose task is to analyze an application executable provided in binary form and to provide information about the statements and variables present in the application source code at the time of compilation, the number and type of instructions present in the application and other information which can be extracted from the binary. A BAS is written for a specific machine and for a specific compiler and it interprets the formats of the binary, instructions, addresses and the source program information stashed away in the binary in formats defined by the machine and compiler. Certain general characteristics are assumed for all architectures and compilers as described below and are used in the sequel.

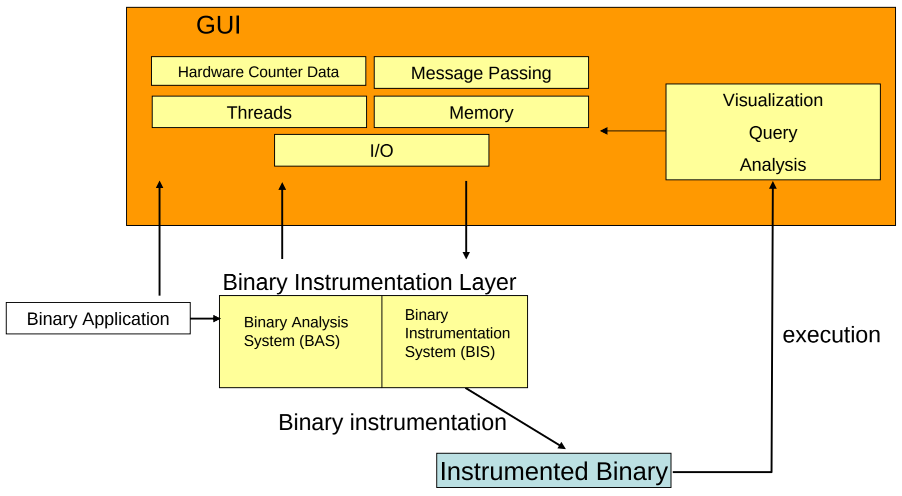


Figure 1: Structure of the Framework

1. The binary contains the code which is a sequence of instructions, each having a distinct instruction address. An instruction contains an opcode which can be represented as an integer and which specifies the operation performed by the instruction.

2. All symbols of variables and functions used in the program, along with their attributes, such as types and addresses when they are statically bound can be deciphered from the binary executable.

The role of the BAS is to generate a list of files:

1. appname.files: contains a list of source file names used in the binary.

2. appname.functions: contains a list of all function names used in the binary and additional information associated with each function.

3. appname.variables: contains a list of all variable names used in the binary and additional information associated with each variable.

4. appname.fcalls: contains information about all function calls and the instruction address of each branch to each function.

5. appname.opcodes: contains information about the type of instructions in the executable.

6. appname.ias: contains a mapping from the instruction address to the source file and source line.

When the *peekperf* GUI opens a binary, it makes a request to the BAS. By processing the BAS files, the *peekperf* GUI is able to provide an instructive interface for scientists to control the instrumentation.

BIS provides a low-level binary modification facility, whose task is to modify the execution of an application in order to execute certain actions specified by the user. BIS is written for a specific machine and for a specific compiler. It takes the following components as the input.

1. Input Application: An application provided in the form of a binary executable.

2. `bisSpec`: A specification of the actions to be executed which consists of a sequence of `action points` and associated `action probes`.

3. Probe Library: A library which defines action probes.

The output of the BIS is the instrumented version of the original application. BIS modifies the execution of the input application according to the rules and actions specified by a `bisSpec` (whose format is described later). A `bisSpec` consists of a sequence of action points and associated action functions. Whenever the execution reaches an action point the corresponding action function is invoked. Each invocation of an action function is implemented as a function call adhering to the conventions of argument passing laid down in the Application Binary Interface (ABI) specified by the compiler. It is further assumed that all the action functions specified in the `bisSpec` are present in the input probe library.

```
bisSpec     ::= actionSpec | actionSpec  bisSpec
actionSpec  ::= actionPoint  actionFunc actionArg
                actionOrder EOL
actionPoint ::= instructionAddress
actionFunc  ::= string
actionOrder ::= before | after | replace
```

Semantics of the `bisSpec`:

1. Each action refers to an instruction (action site) in the binary code. The instruction at which the action is taken is determined by the `actionPoint`, which specifies the instruction address for the instruction.

2. It is possible that an instruction may be associated with multiple actions. All actions with the same `actionOrder` will all be executed in the order in which they are specified except `replace`, there can be only one replace action per instruction.

3. Each action consists of invoking the function identified by the `actionFunc` string. This function must be present in the probe library.

4. The `actionOrder` determines whether the action takes place `before`, `after` or instead of (`replace`) the instruction at the action site. At an action site, first all before actions are executed; then either the instruction at the action site is executed or the sequence of all replace actions are executed; and finally all after actions are executed.

5. An action invokes the function `actionFunc` as a normal function. The first argument passed to the action function is a pointer to a fixed-size buffer. A BIS implementation publishes the size of each field in the buffer. The buffer contains the following fields:

   (a) actionArg

   (b) actionPoint

   (c) binary instruction at the action site

   (d) instruction address target of the branch (only if the instruction at the action site is a branch)

   (e) memory address loaded or stored if the instruction at the action site is a load or store

   (f) data loaded if the instruction at the action site is a load and actionOrder = after. data stored if the instruction at the action site is a store and actionOrder = before, 0 otherwise

   (g) the first argument arg0, if actionOrder = before or replace and the instruction at the action site is a branch instruction. Arg0 is the first argument determined by applying the conventions of the ABI as if the instruction was a call to a function. 0 otherwise.

   (h) the first return value result0, if actionOrder = after and the instruction at the action site is a branch instruction. Result0 is the first return value determined by applying the conventions of the ABI as if the instruction was a call to a function, 0 otherwise.

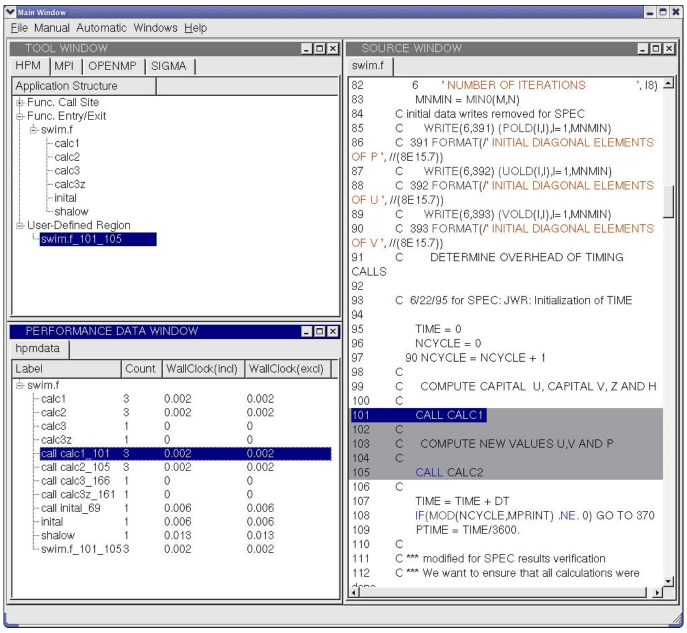
   (i) the value of the stack pointer, if such a notion exists in the architecture, 0 otherwise.

   The remaining arguments to the action function are the same as in the original code (meaningful only if the instruction at the action site is a function call and actionOrder = before or replace).

The *peekperf* GUI makes a request to BIS when the instrumentation points are determined. *peekperf* composes the bisSepc file based on the user's input and pass the bisSpec file along with a list of probe libraries to the BIS. The BIS then creates an instrumented version of the original application.

4. A VERSATILE USER-INTERFACE

The *peekperf* GUI is the control center of our framework. The entire performance tuning process, from instrumentation to execution and analysis of data can be conducted from here. This interface provide three different types of windows (see Figure 2). The source window shows the source code files for the selected application (if such source code files are available). The tool window controls the data collection and instrumentation. It displays the program structure that is specifically designed for different aspects of performance



Application Structure
- Func. Call Site
- Func. Entry/Exit
  - swim.f
    - calc1
    - calc2
    - calc3
    - calc3z
    - inital
    - shalow
- User-Defined Region
  - swim.f_101_105

Label	Count	WallClock(incl)	WallClock(excl)
swim.f			
calc1	3	0.002	0.002
calc2	3	0.002	0.002
calc3	1	0	0
calc3z	1	0	0
call calc1_101	3	0.002	0.002
call calc2_105	3	0.002	0.002
call calc3_166	1	0	0
call calc3z_161	1	0	0
call inital_69	1	0.006	0.006
inital	1	0.006	0.006
shalow	1	0.013	0.013
swim.f_101_105	3	0.002	0.002

```
82        6     ' NUMBER OF ITERATIONS              ', I8)
83          MNMIN = MIN0(M,N)
84      C initial data writes removed for SPEC
85      C     WRITE(6,391) (POLD(I,I),I=1,MNMIN)
86      C  391 FORMAT(/' INITIAL DIAGONAL ELEMENTS
OF P ', //(8E15.7))
87      C     WRITE(6,392) (UOLD(I,I),I=1,MNMIN)
88      C  392 FORMAT(/' INITIAL DIAGONAL ELEMENTS
OF U ', //(8E15.7))
89      C     WRITE(6,393) (VOLD(I,I),I=1,MNMIN)
90      C  393 FORMAT(/' INITIAL DIAGONAL ELEMENTS
OF V ', //(8E15.7))
91      C      DETERMINE OVERHEAD OF TIMING
CALLS
92
93      C  6/22/95 for SPEC: JWR: Initialization of TIME
94
95          TIME = 0
96          NCYCLE = 0
97        90 NCYCLE = NCYCLE + 1
98      C
99      C    COMPUTE CAPITAL  U, CAPITAL V, Z AND H
100     C
101         CALL CALC1
102     C
103     C    COMPUTE NEW VALUES U,V AND P
104     C
105         CALL CALC2
106     C
107         TIME = TIME + DT
108         IF(MOD(NCYCLE,MPRINT) .NE. 0) GO TO 370
109         PTIME = TIME/3600.
110     C
111     C *** modified for SPEC results verification
112     C *** We want to ensure that all calculations were
```

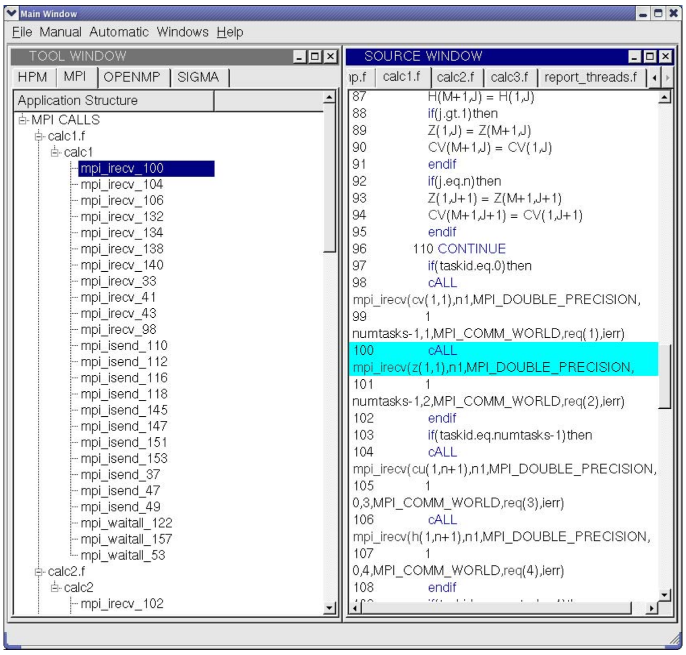
Figure 2: Front-End GUI

data (HPM: hardware counter data, MPI: message passing, OPENMP: threads, SIGMA: memory). The performance data window displays the performance data generated by the instrumented applications. The various options available for instrumentation and performance data visualization are described in the next two subsections.

4.1 Program Structure and Control of Data Collection

By selecting the appropriate tab in the tool window, the program structure for the selected performance dimension is displayed. The program structure is described as a hierarchical tree. For example, the HPM tree represents the function call sites and the function entry/exit points for the entire program. Those are predefined instrumentation points for the collection of hardware counters. Similar tree structure is provided with each of the tools represented by the different tabs. In addition, the user may operate on the source window to define their own instrumentation region. As shown in Figure 2, the swim.f_101_105 label in the HPM tree is selected through the source browser window and its defined region is from line 101 to 105 in the swim.f file. A similar tree view is designed for each dimension of performance data. Figure 3(a) shows this view for MPI tool and Figure 3(b) shows this for OpenMP. The tree view allows the users to easily select the predefined instrumentation regions. User-defined instrumentation region will be reflected in the tree structure after the region is selected through the source window. Users can efficiently control the data collection by navigating the tree structure.

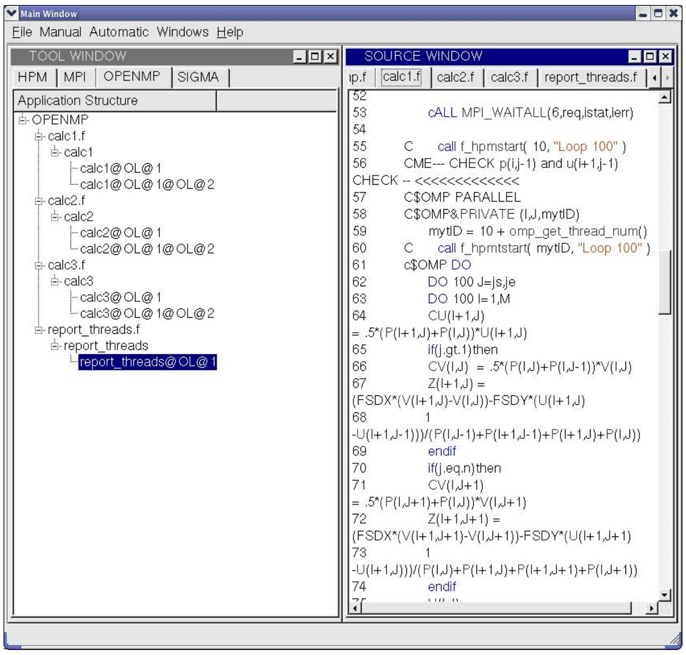
The selected instrumentation points will be highlighted in the tree structure. If the source code file is available, the instrumentation regions will be highlighted in the source window as well. Our infrastructure allows for simultaneous data collection for different dimensions of performance data (for example, hardware counters, message passing, and thread information can all be collected simultaneously in one single


Main Window
File Manual Automatic Windows Help
TOOL WINDOW
HPM MPI OPENMP SIGMA
SOURCE WINDOW
np.f calc1.f calc2.f calc3.f report_threads.f
Application Structure

```
MPI CALLS
  calc1.f
    calc1
      mpi_irecv_100
      mpi_irecv_104
      mpi_irecv_106
      mpi_irecv_132
      mpi_irecv_134
      mpi_irecv_138
      mpi_irecv_140
      mpi_irecv_33
      mpi_irecv_41
      mpi_irecv_43
      mpi_irecv_98
      mpi_isend_110
      mpi_isend_112
      mpi_isend_116
      mpi_isend_118
      mpi_isend_145
      mpi_isend_147
      mpi_isend_151
      mpi_isend_153
      mpi_isend_37
      mpi_isend_47
      mpi_isend_49
      mpi_waitall_122
      mpi_waitall_157
      mpi_waitall_53
  calc2.f
    calc2
      mpi_irecv_102
```

```
           H(M+1,J) = H(1,J)
           if(j.gt.1)then
           Z(1,J) = Z(M+1,J)
           CV(M+1,J) = CV(1,J)
           endif
           if(j.eq.n)then
           Z(1,J+1) = Z(M+1,J+1)
           CV(M+1,J+1) = CV(1,J+1)
           endif
       110 CONTINUE
           if(taskid.eq.0)then
           cALL
mpi_irecv(cv(1,1),n1,MPI_DOUBLE_PRECISION,
           1
numtasks-1,1,MPI_COMM_WORLD,req(1),ierr)
           cALL
mpi_irecv(z(1,1),n1,MPI_DOUBLE_PRECISION,
           1
numtasks-1,2,MPI_COMM_WORLD,req(2),ierr)
           endif
           if(taskid.eq.numtasks-1)then
           cALL
mpi_irecv(cu(1,n+1),n1,MPI_DOUBLE_PRECISION,
           1
0,3,MPI_COMM_WORLD,req(3),ierr)
           cALL
mpi_irecv(h(1,n+1),n1,MPI_DOUBLE_PRECISION,
           1
0,4,MPI_COMM_WORLD,req(4),ierr)
           endif
```

(a)


Main Window
File Manual Automatic Windows Help
TOOL WINDOW
HPM MPI OPENMP SIGMA
SOURCE WINDOW
np.f calc1.f calc2.f calc3.f report_threads.f
Application Structure

```
OPENMP
  calc1.f
    calc1
      calc1@OL@1
      calc1@OL@1@OL@2
  calc2.f
    calc2
      calc2@OL@1
      calc2@OL@1@OL@2
  calc3.f
    calc3
      calc3@OL@1
      calc3@OL@1@OL@2
  report_threads.f
    report_threads
      report_threads@OL@1
```

```
           cALL MPI_WAITALL(6,req,istat,ierr)

      C    call f_hpmstart( 10, "Loop 100" )
      CME--- CHECK p(i,j-1) and u(i+1,j-1)
CHECK -- <<<<<<<<<<<<<<
      C$OMP PARALLEL
      C$OMP&PRIVATE (I,J,mytID)
           mytID = 10 + omp_get_thread_num()
      C    call f_hpmstart( mytID, "Loop 100" )
      c$OMP DO
           DO 100 J=js,je
           DO 100 I=1,M
           CU(I+1,J)
= .5*(P(I+1,J)+P(I,J))*U(I+1,J)
           if(j.gt.1)then
           CV(I,J)  = .5*(P(I,J)+P(I,J-1))*V(I,J)
           Z(I+1,J) =
(FSDX*(V(I+1,J)-V(I,J))-FSDY*(U(I+1,J)
           1
-U(I+1,J-1)))/(P(I,J-1)+P(I+1,J-1)+P(I+1,J)+P(I,J))
           endif
           if(j.eq.n)then
           CV(I,J+1)
= .5*(P(I,J+1)+P(I,J))*V(I,J+1)
           Z(I+1,J+1) =
(FSDX*(V(I+1,J+1)-V(I,J+1))-FSDY*(U(I+1,J+1)
           1
-U(I+1,J)))/(P(I,J)+P(I+1,J)+P(I+1,J+1)+P(I,J+1))
           endif
```

(b)

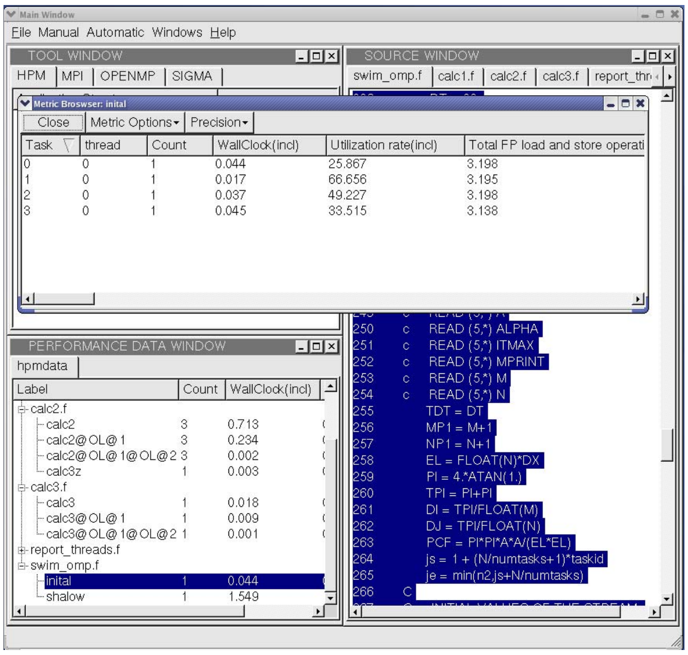
Figure 3: Program Structure for MPI and OpenMP

run). Users can cycle through the instrumentation tabs and properly define all the required performance data. Then the application is instrumented by accessing the menu.

4.2 Performance Data Visualization

When the instrumented applications are executed, resulting performance data is recorded in a set of XML files. Each XML file is associated with one executing process and records one dimension of performance data. The XML file follows a standard specification so that our framework can render the performance data for different dimensions of performance data and can correlate performance metrics with the application source code.

In Figure 4, the performance data window shows the hardware counter metrics. When a performance metric is selected, the corresponding statements are highlighted in the source code window. Therefore, users can immediately relate the collected performance information (e.g., timing, number of cache misses) to the corresponding source code sections (e.g., loops) of the application. This information is essential to isolate the performance bottlenecks and to guide the analyst towards understanding whether the application at hand exhibits inefficiency due to the implementation of the sequential portions of the algorithm, the memory access pattern or the communication aspects. This data is therefore a first step towards isolating possible bottlenecks that can be further investigated with the other dimension of performance data. The interface provides several means of analyzing the collected data. It is possible to sort and filter the data, and access detailed metrics for each instrumented section and for each thread and each process. The metric browser window in Figure 4 gives more detailed hardware counter metrics for each process.



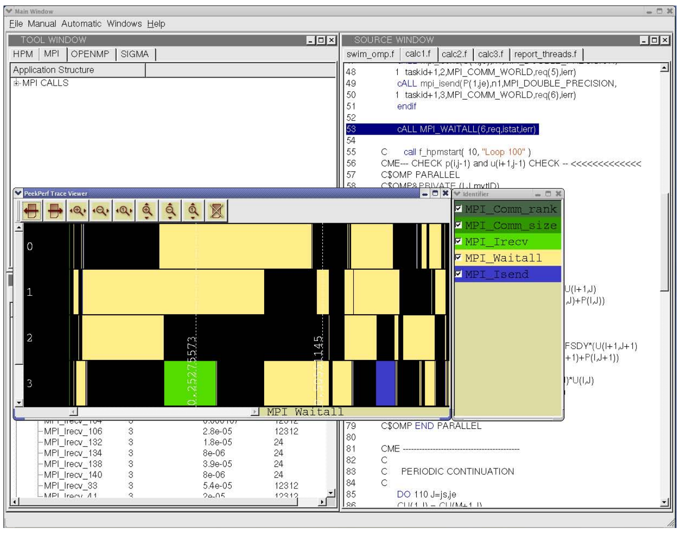
Task	thread	Count	WallClock(incl)	Utilization rate(incl)	Total FP load and store operati
0	0	1	0.044	25.867	3.198
1	0	1	0.017	66.656	3.195
2	0	1	0.037	49.227	3.198
3	0	1	0.045	33.515	3.138

Label	Count	WallClock(incl)
calc2.f		
calc2	3	0.713
calc2@OL@1	3	0.234
calc2@OL@1@OL@2	3	0.002
calc3z	1	0.003
calc3.f		
calc3	1	0.018
calc3@OL@1	1	0.009
calc3@OL@1@OL@2	1	0.001
report_threads.f		
swim_omp.f		
inital	1	0.044
shalow	1	1.549

```
      c    READ (5,*) ALPHA
      c    READ (5,*) ITMAX
      c    READ (5,*) MPRINT
      c    READ (5,*) M
      c    READ (5,*) N
           TDT = DT
           MP1 = M+1
           NP1 = N+1
           EL = FLOAT(N)*DX
           PI = 4.*ATAN(1.)
           TPI = PI+PI
           DI = TPI/FLOAT(M)
           DJ = TPI/FLOAT(N)
           PCF = PI*PI*A*A/(EL*EL)
           js = 1 + (N/numtasks+1)*taskid
           je = min(n2,js+N/numtasks)
      C
```

Figure 4: Hardware Counter Metrics

In addition to the "flat" view of the performance data, our performance data visualization also offers the tracer view for the MPI trace. Our tracer display shows the sequence of communication events on a time-line of the execution. It is then possible to graphically identify the communication pattern and spot possible inefficiencies. As with all the performance data generated in our framework, the trace events are mapped back to the source code symbols and functions. For example, when selecting an individual MPI function in the tracer display, the corresponding MPI function call is automatically highlighted in the source window (see Figure 5).



```
           1  taskid+1,2,MPI_COMM_WORLD,req(5),ierr)
           cALL mpi_isend(P(1,je),n1,MPI_DOUBLE_PRECISION,
           1  taskid+1,3,MPI_COMM_WORLD,req(6),ierr)
           endif

           cALL MPI_WAITALL(6,req,istat,ierr)

      C     call f_hpmstart( 10, "Loop 100" )
      CME--- CHECK p(i,j-1) and u(i+1,j-1) CHECK -- <<<<<<<<<<<<<
      C$OMP PARALLEL
      ...
      C$OMP END PARALLEL

      CME ---------------------------------------
      C
      C   PERIODIC CONTINUATION
      C
           DO 110 J=js,je
```

MPI_Irecv_106	3	2.8e-05	12312
MPI_Irecv_132	3	1.8e-05	24
MPI_Irecv_134	3	8e-06	24
MPI_Irecv_138	3	3.9e-05	24
MPI_Irecv_140	3	8e-06	24
MPI_Irecv_33	3	5.4e-05	12312

Figure 5: MPI Performance Data

A typical performance tuning session would start with the user collecting the hardware counter metrics and the message passing data. This would allow the user to identify the most computationally expensive functions and detect possible communication bottlenecks. The analysis would then be directed towards selected functions, possibly employing the memory analysis capabilities and the thread profiling ability. Our framework makes it possible to cycle through different sets of performance data seamlessly by selecting different tabs in the tool window. It is possible to incrementally vary the instrumentation to generate more data that can be compared with previously generated data sets. By operating in this manner the performance analyst can effectively dig down into the relevant segments of the application and narrow down any performance inefficiency.

5. USAGE EXAMPLE

In this section, we briefly illustrate how our framework is used by walking through a simple analysis of ks_imp_dyn2, a dynamic QCD with improved staggered quarks of 2 masses from the MIMD Lattice Computation (MILC) [5][2].

The hardware platform used for this application's performance analysis is a POWER5 [19] processor chip containing two microprocessor cores. Each core contains a 64MB level1 (L1) instruction cache, a 32KB L1 data cache, two fixed-point execution units, two floating-point execution units, two load/store execution units, one branch execution unit, and one execution unit to perform logic operations on the condition.

The MILC application is executed serially on a single processor and we collect only the hardware counter metrics for this example. To minimize the overhead of instrumentation and the volume of collected performance data, we instrument limited number of code segments in each iteration. In the first iteration, we instrumented only the *main* and the function calls inside the main function. The code tree structure displayed in the tool window provides users an efficient way for this selective instrumentation (Figure 6). After the instrumentation and the execution, the performance data is loaded to *peekperf* as shown in Figure 7. The data displayed in the performance data window (bottom left corner, under WallClock(excl) time) shows that most of the execution time is spent in the *update* function (over 90 seconds out of the 101 seconds of total execution time). Therefore, in the second iteration, we instrumented only this function and all its callees. We changed the monitored event set group to one of the few stall events, the LSU Stalls. These are the events of interest as they show lost cycles due to stalling of the processor pipeline, one of the main sources of performance degradation of HPC applications. Changing the event set group is easily done through the context menu of tool window. Figure 8 shows the result of the second iteration. The time spent in the *update_h* function call is about 2/3 of total execution time for the *update* function. With the right click of the mouse button on the performance data item, one can see more details of the performance metrics associated with this performance data item. By observing the large number of stalls caused by the LSU unit (PM_CMPLU_STALL_LSU), the scientist may speculate that there is some potential performance problem in their application due to load/store stalling.

Based on the observation from the second iteration, we repeat the process and instrument *img_gauge_force*, *eo_fermion_force_3f* and the call sites inside the functions. With further analysis, we identify that *img_gauge_force*, *eo_fermion_force_3f*, and *su3* matrix-matrix multiplication functions experience the most stalls. We do not intend to address the performance tuning techniques in this section. The walk-through example is to demonstrate how our framework can help scientists understand, direct and tune their application.

Following the approach described above, scientists can continue the performance analysis from the GUI and efficiently perform the five steps of the iterative performance tuning cycle outlined in Section 1.

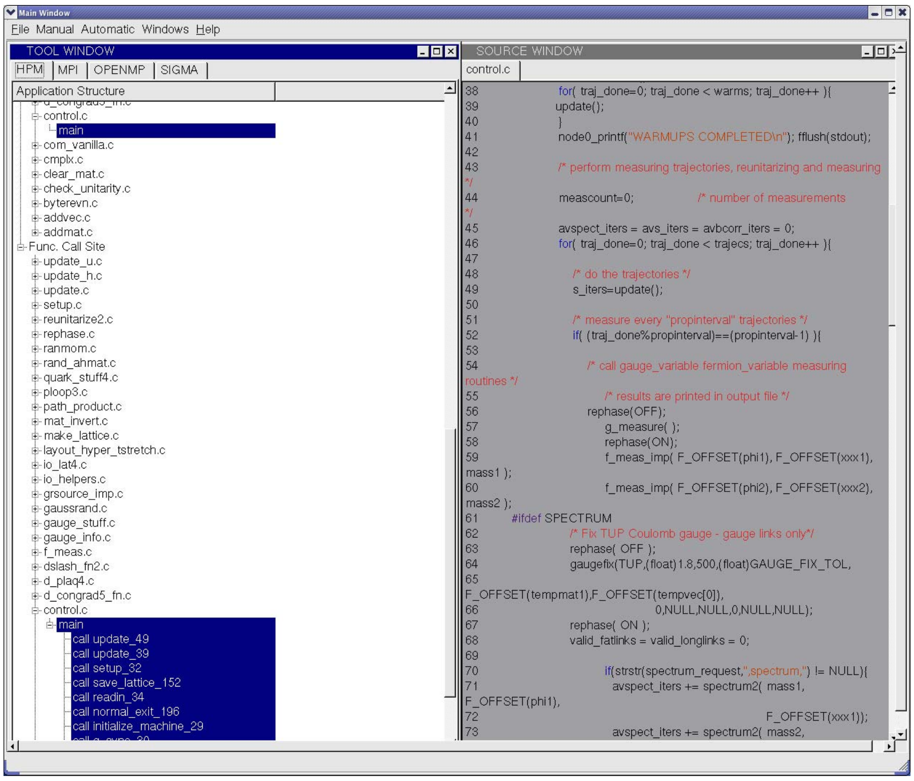
6. RELATED WORK

In this section, we describe two categories of related work. First, we describe the related core instrumentation technologies including binary patching and runtime patching. Then we describe some performance tools built on top of those core instrumentation technologies.

6.1 Instrumentation

Similar to our binary instrumentation functionality, performance tools like ATOM [20], and PMaCInst [13] also provide binary instrumentation functionality. In other words, those tools instrument the application by rewriting the binary executable. ATOM was one of popular binary patching tools. However, it only works only on the Alpha platform

[2]This work was in part based on the MILC collaboration's public lattice gauge theory code. See http://physics.utah.edu/ detar/milc.html



```
Application Structure
    control.c
      main
    com_vanilla.c
    cmplx.c
    clear_mat.c
    check_unitarity.c
    byterevn.c
    addvec.c
    addmat.c
  Func. Call Site
    update_u.c
    update_h.c
    update.c
    setup.c
    reunitarize2.c
    rephase.c
    ranmom.c
    rand_ahmat.c
    quark_stuff4.c
    ploop3.c
    path_product.c
    mat_invert.c
    make_lattice.c
    layout_hyper_tstretch.c
    io_lat4.c
    io_helpers.c
    grsource_imp.c
    gaussrand.c
    gauge_stuff.c
    gauge_info.c
    f_meas.c
    dslash_fn2.c
    d_plaq4.c
    d_congrad5_fn.c
    control.c
      main
        call update_49
        call update_39
        call setup_32
        call save_lattice_152
        call readin_34
        call normal_exit_196
        call initialize_machine_29
```

```
        for( traj_done=0; traj_done < warms; traj_done++ ){
        update();
        }
        node0_printf("WARMUPS COMPLETED\n"); fflush(stdout);

        /* perform measuring trajectories, reunitarizing and measuring
*/
        meascount=0;            /* number of measurements
*/
        avspect_iters = avs_iters = avbcorr_iters = 0;
        for( traj_done=0; traj_done < trajecs; traj_done++ ){

            /* do the trajectories */
            s_iters=update();

            /* measure every "propinterval" trajectories */
            if( (traj_done%propinterval)==(propinterval-1) ){

                /* call gauge_variable fermion_variable measuring
routines */
                    /* results are printed in output file */
                rephase(OFF);
                    g_measure( );
                    rephase(ON);
                    f_meas_imp( F_OFFSET(phi1), F_OFFSET(xxx1),
mass1 );
                    f_meas_imp( F_OFFSET(phi2), F_OFFSET(xxx2),
mass2 );
    #ifdef SPECTRUM
            /* Fix TUP Coulomb gauge - gauge links only*/
            rephase( OFF );
            gaugefix(TUP,(float)1.8,500,(float)GAUGE_FIX_TOL,

F_OFFSET(tempmat1),F_OFFSET(tempvec[0]),
                            0,NULL,NULL,0,NULL,NULL);
            rephase( ON );
            valid_fatlinks = valid_longlinks = 0;

                    if(strstr(spectrum_request,",spectrum,") != NULL){
                      avspect_iters += spectrum2( mass1,
F_OFFSET(phi1),
                                                F_OFFSET(xxx1));
                      avspect_iters += spectrum2( mass2,
```

Figure 6: Instrumentation

which is no longer in production. PMaCInst is close to our binary instrumentation and tries to provide binary instrumentation on PowerPC/AIX platform. It provides API to allow users specify the locations for instrumentation. Our binary instrumentation uses anchor points to allow instrumentation at any binary address.

In addition to binary patching/instrumentation, performance tools like Pin [8], Dyninst [7], DPCL [2] provide runtime instrumentation. During the runtime, the application image in the memory is modified to accommodate the inserted code for instrumentations. Pin provides instrumentation by using a just-in-time (JIT) compiler. The application is running on top of Pin, and Pin intercepts the application with an interruption to perform instrumentation. Dyninst replaces an instruction from the application with a jump instruction to the function call stub and instrumentation code. DPCL was IBM's version of Dyninst and now is released to the public domain. However, these kinds of runtime instrumentation usually require system daemons running in the background. This may be acceptable in most cases, but does not optimize productivity needs. For DPCL, it cannot perform instruction-level instrumentation which may meet most of HPC needs, but not those of HPCS. Our binary instrumentation does not require any system daemons running and is able to perform instruction-level instrumentation.

6.2 Integrated Performance Tool

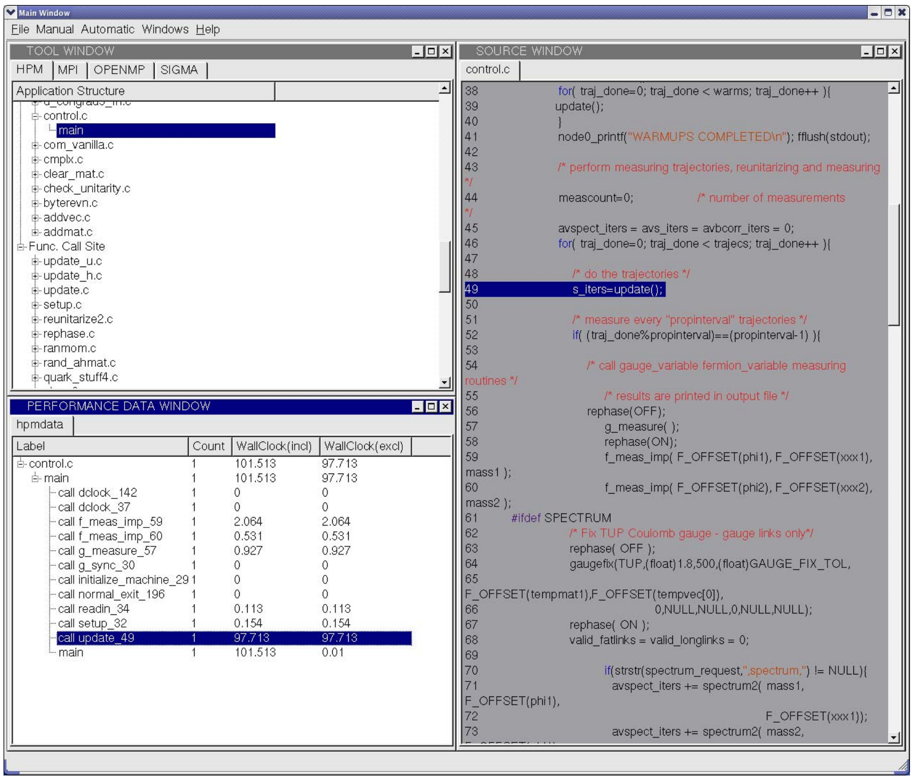
HPCView [15] presents the performance data collected and correlate it to program structure information to produce a performance database. Similar to our visualization interfaces, it uses hierarchical display for top-down analysis. In terms of data collection, it uses scripts to automate the performance data collection process. However, we believe controlling the data collection through navigating the program structure provides a more interactive and user-friendly process. Their collected performance data is also converted into XML-based format files which is similar to our framework.

TAU [18] is probably one of the full featured HPC analysis tools. It provides both manual and automatic source code instrumentation. It also allows dynamic instrumentation by using Dyninst. Unlike our framework targeting at interactive performance debugging, TAU's visualization, ParaProf, focuses on the presentation and can present performance data from different data collectors.

SpeedShop [6] is a performance analysis tool for SGI systems. Its dynamic instrumentation is also based on Dyninst. It can run performance experiments on an application and present the results to help users remove performance obstacles. The new design enables users to extend functionality with their own experiments. The design strategies are similar to our framework except it uses the runtime dynamic instrumentation.

DynTG [14], which uses DPCL for dynamic instrumentation. The functionality of interactive performance tuning design is similar to our framework except instrumentation mechanism. It uses performance modules to reconfigure the data acquisition and provides a source browser that allows users to insert probes dynamically into the target application. Due to the limitation of DPCL, the location that can be instrumented is limited.

Paradyn [16], which is rewritten on top of the Dyninst, is able to collect performance data dynamically during the run-



```
Application Structure
    control.c
      main
    com_vanilla.c
    cmplx.c
    clear_mat.c
    check_unitarity.c
    byterevn.c
    addvec.c
    addmat.c
  Func. Call Site
    update_u.c
    update_h.c
    update.c
    setup.c
    reunitarize2.c
    rephase.c
    ranmom.c
    rand_ahmat.c
    quark_stuff4.c
```

Label	Count	WallClock(incl)	WallClock(excl)
control.c	1	101.513	97.713
main	1	101.513	97.713
call dclock_142	1	0	0
call dclock_37	1	0	0
call f_meas_imp_59	1	2.064	2.064
call f_meas_imp_60	1	0.531	0.531
call g_measure_57	1	0.927	0.927
call g_sync_30	1	0	0
call initialize_machine_29	1	0	0
call normal_exit_196	1	0	0
call readin_34	1	0.113	0.113
call setup_32	1	0.154	0.154
call update_49	1	97.713	97.713
main	1	101.513	0.01

```
38          for( traj_done=0; traj_done < warms; traj_done++ ){
39          update();
40          }
41          node0_printf("WARMUPS COMPLETED\n"); fflush(stdout);
42
43          /* perform measuring trajectories, reunitarizing and measuring
*/
44          meascount=0;            /* number of measurements
*/
45          avspect_iters = avs_iters = avbcorr_iters = 0;
46          for( traj_done=0; traj_done < trajecs; traj_done++ ){
47
48              /* do the trajectories */
49              s_iters=update();
50
51              /* measure every "propinterval" trajectories */
52              if( (traj_done%propinterval)==(propinterval-1) ){
53
54                  /* call gauge_variable fermion_variable measuring
routines */
55                      /* results are printed in output file */
56                  rephase(OFF);
57                      g_measure( );
58                      rephase(ON);
59                      f_meas_imp( F_OFFSET(phi1), F_OFFSET(xxx1),
mass1 );
60                      f_meas_imp( F_OFFSET(phi2), F_OFFSET(xxx2),
mass2 );
61      #ifdef SPECTRUM
62              /* Fix TUP Coulomb gauge - gauge links only*/
63              rephase( OFF );
64              gaugefix(TUP,(float)1.8,500,(float)GAUGE_FIX_TOL,
65
F_OFFSET(tempmat1),F_OFFSET(tempvec[0]),
66                              0,NULL,NULL,0,NULL,NULL);
67              rephase( ON );
68              valid_fatlinks = valid_longlinks = 0;
69
70                      if(strstr(spectrum_request,",spectrum,") != NULL){
71                        avspect_iters += spectrum2( mass1,
F_OFFSET(phi1),
72                                                      F_OFFSET(xxx1));
73                        avspect_iters += spectrum2( mass2,
```

Figure 7: Performance Data Visualization

time. Paradyn provides automatic search for performance bottlenecks. Its Performance Consultant determines 3W (where, when, why) of performance problems. In the future, we plan to improve Our framework for the interactive tuning process to eliminate usersŠ guesswork of manual problem determination.

Many performance tools like described above are built on top of the dynamic instrumentation technologies. Dynamic instrumentation provides flexibility that allows "probes" modified during runtime (e.g., on or off). However, the instrumentation needs to be done every time the executable is loaded into memory. Our binary instrumentation provides an alternative so the instrumentation is done only once and the instrumented binary can be used again. This is especially useful when the binary executable is large and the instrumentation process can be time consuming.

7. CONCLUDING REMARKS

In this paper, we describe a new framework for application performance tuning that emphasizes productivity. The GUI framework encompasses the entire performance tuning methodology. This GUI is the single interface for both instrumentation and analysis, and is the means by which the programmer selectively chooses the degree of probing. Furthermore, the GUI is the centralized component that allows simultaneous analysis of all dimensions of application performance analysis (CPU, memory, message passing, threads, and I/O). The GUI is assisted by a sophisticated binary instrumentation facility that takes specifications at the source-level and renders results in terms of source-level names, without resorting to any changes to the source programs. By abstracting the details of binary instrumentation away from the analyst and providing an automated and quickly repeatable probe-result cycle at the source level, our framework contributes to increased productivity of the end user. It takes away the burden of low-level details of instrumentation and secures the source code from unintentional changes.

From our past experience developing performance tools and dealing with performance tuning experts, we recognize that a typical analyst, looks at certain patterns in the result data and initiates further actions. If we can capture the final patterns used to diagnose a problem and associate them with the suggested improvements, we can create an "intelligent" tool that can predict remedies from symptoms. We plan to extend our system to create knowledge-based repositories for automatic performance tuning, further increasing the productivity of the user.

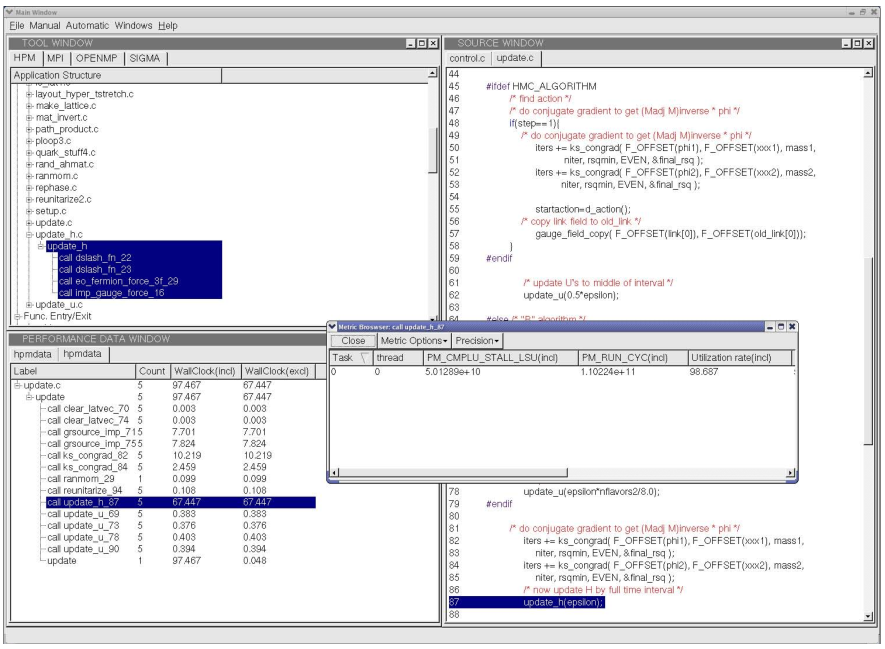
Our current and future productivity technologies will be integrated into our existing framework [3]. Our framework is currently available on a number of platforms and provides much of the basic functionality. However, the newer productivity technologies HPCS program will be introduced as this initiative moves forward.

8. REFERENCES

[1] Darpa high productivity computing systems (hpcs). *http://www.darpa.mil/IPTO/programs/hpcs/index.htm*.

[2] Dynamic Probe Class Library. *http://sourceforge.net/projects/dpcl/*.

[3] IBM Advanced Computing Technology Center. *http://www.research.ibm.com/*.


Main Window
File Manual Automatic Windows Help
TOOL WINDOW
HPM | MPI | OPENMP | SIGMA
SOURCE WINDOW
control.c | update.c
PERFORMANCE DATA WINDOW
hpmdata | hpmdata
Metric Broswser: call update_h_87
Close | Metric Options | Precision

```
Application Structure
  layout_hyper_tstretch.c
  make_lattice.c
  mat_invert.c
  path_product.c
  ploop3.c
  quark_stuff4.c
  rand_ahmat.c
  ranmom.c
  rephase.c
  reunitarize2.c
  setup.c
  update.c
  update_h.c
    update_h
      call dslash_fn_22
      call dslash_fn_23
      call eo_fermion_force_3f_29
      call imp_gauge_force_16
  update_u.c
Func. Entry/Exit
```

Label	Count	WallClock(incl)	WallClock(excl)
update.c	5	97.467	67.447
update	5	97.467	67.447
call clear_latvec_70	5	0.003	0.003
call clear_latvec_74	5	0.003	0.003
call grsource_imp_71	5	7.701	7.701
call grsource_imp_75	5	7.824	7.824
call ks_congrad_82	5	10.219	10.219
call ks_congrad_84	5	2.459	2.459
call ranmom_29	1	0.099	0.099
call reunitarize_94	5	0.108	0.108
call update_h_87	5	67.447	67.447
call update_u_69	5	0.383	0.383
call update_u_73	5	0.376	0.376
call update_u_78	5	0.403	0.403
call update_u_90	5	0.394	0.394
update	1	97.467	0.048

```
44
45      #ifdef HMC_ALGORITHM
46              /* find action */
47              /* do conjugate gradient to get (Madj M)inverse * phi */
48              if(step==1){
49                  /* do conjugate gradient to get (Madj M)inverse * phi */
50                      iters += ks_congrad( F_OFFSET(phi1), F_OFFSET(xxx1), mass1,
51                              niter, rsqmin, EVEN, &final_rsq );
52                      iters += ks_congrad( F_OFFSET(phi2), F_OFFSET(xxx2), mass2,
53                              niter, rsqmin, EVEN, &final_rsq );
54
55                      startaction=d_action();
56                  /* copy link field to old_link */
57                      gauge_field_copy( F_OFFSET(link[0]), F_OFFSET(old_link[0]));
58              }
59      #endif
60
61                  /* update U's to middle of interval */
62                  update_u(0.5*epsilon);
63
64      #else /* "B" algorithm */
...
78                  update_u(epsilon*nflavors2/8.0);
79      #endif
80
81              /* do conjugate gradient to get (Madj M)inverse * phi */
82                  iters += ks_congrad( F_OFFSET(phi1), F_OFFSET(xxx1), mass1,
83                      niter, rsqmin, EVEN, &final_rsq );
84                  iters += ks_congrad( F_OFFSET(phi2), F_OFFSET(xxx2), mass2,
85                      niter, rsqmin, EVEN, &final_rsq );
86                  /* now update H by full time interval */
87                  update_h(epsilon);
88
```

Task	thread	PM_CMPLU_STALL_LSU(incl)	PM_RUN_CYC(incl)	Utilization rate(incl)
0	0	5.01289e+10	1.10224e+11	98.687

Figure 8: Performance Data Visualization - High Stalls Caused by LSU

[4] IBM Advanced Computing Technology Center MPI Tracer/Profiler. *http://www.research.ibm.com/actc/projects/mpitracer.shtml*.

[5] Mimd lattice computation (milc) collaboration. *http://physics.utah.edu/ detar/milc.html*.

[6] Open speedshop. *http://oss.sgi.com/projects/openspeedshop/*.

[7] B. Buck and J. Hollingsworth. An api for runtime code patching. *Journal of High Performance Computing Applications*, 14(4), 2000.

[8] R. M. C. K. Luk, R. Cohn et al. Pin: building customized program analysis tools with dynamic instrumentation. *PLDI*, pages 190–200, 2005.

[9] L. DeRose. The hardware performance monitor toolkit. *Proceedings of Euro-Par*, pages 122–131, August 2001.

[10] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia. Sigma: A simulator infrastructure to guide memory analysis. *Proceedings of Supercomputing*, November 2002.

[11] L. DeRose, B. Mohr, and S. Seelam. An implementation of the pomp performance monitoring interface for openmp based on dynamic probes. *Proceedings of the fifth European Workshop on OpenMP - EWOMP'03*, September 2003.

[12] L. DeRose, B. Mohr, and S. Seelam. Profiling and tracing openmp applications with pomp based monitoring libraries. *Proceedings of the International Conference on Promotion and Advancement of Parallel Computing - Euro-Par 2004*, 31st August - 3rd September 2004.

[13] L. C. M. Laurenzano, M. Tikir and A. Snavely. The pmac binary instrumentation library for power pc. *Workshop on Instrumentation and Applications, held in conjunction with ASPLOS XII*, October 2006.

[14] J. C. G. Martin Schulz, John May. Dyntg: A tool for interactive, dynamic instrumentation. *International Conference on Computational Science*, 2005.

[15] J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent. Hpcview: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–101, April 2002.

[16] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 28:37–46, November 1995.

[17] S. Sbaraglia, K. Ekanadham, S. Crea, and S. Seelam. psigma: An infrastructure for parallel application performance analysis using symbolic specifications. *Proceeding of the European Workshop on OpenMP, Stockholm.*, October, 18-22 2004.

[18] S. Shende and A. D. Malony. Tau: The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.

[19] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, July 2005.

[20] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. *PLDI*, pages 196–205, 1994.