Scientific Programming 16 (2008) 135–153
DOI 10.3233/SPR-2008-0253
IOS Press

A performance tuning methodology with compiler support

Oscar Hernandez [a], Barbara Chapman [a] and Haoqiang Jin [b]
[a] *Computer Science Department, University of Houston, Houston, TX, USA*
E-mails: {oscar, chapman}@cs.uh.edu
[b] *NASA Advanced Supercomputing Division, NASA Ames Research Center, Moffet Field, CA, USA*
E-mail: hjin@nas.nasa.gov

Abstract. We have developed an environment, based upon robust, existing, open source software, for tuning applications written using MPI, OpenMP or both. The goal of this effort, which integrates the OpenUH compiler and several popular performance tools, is to increase user productivity by providing an automated, scalable performance measurement and optimization system. In this paper we describe our environment, show how these complementary tools can work together, and illustrate the synergies possible by exploiting their individual strengths and combined interactions. We also present a methodology for performance tuning that is enabled by this environment. One of the benefits of using compiler technology in this context is that it can direct the performance measurements to capture events at different levels of granularity and help assess their importance, which we have shown to significantly reduce the measurement overheads. The compiler can also help when attempting to understand the performance results: it can supply information on how a code was translated and whether optimizations were applied. Our methodology combines two performance views of the application to find bottlenecks. The first is a high level view that focuses on OpenMP/MPI performance problems such as synchronization cost and load imbalances; the second is a low level view that focuses on hardware counter analysis with derived metrics that assess the efficiency of the code. Our experiments have shown that our approach can significantly reduce overheads for both profiling and tracing to acceptable levels and limit the number of times the application needs to be run with selected hardware counters. In this paper, we demonstrate the workings of this methodology by illustrating its use with selected NAS Parallel Benchmarks and a cloud resolving code.

Keywords: Compiler optimizations, performance tuning methodology, feedback directed optimizations, performance tools

1. Introduction

The difficulty of developing high performance applications has increased greatly with the growth in size and architectural complexity of each new generation of supercomputers. Current systems rely on massive parallelism. Their relatively high memory access costs require the application developer to pay careful attention to memory accesses, both within and across the nodes of a platform, in addition to expressing the parallelism in the application program. The nodes increasingly configure processors with chip multithreading (CMT) and possibly also simultaneous multithreading (SMT) capabilities. When multiple instruction streams run on such processors, they share some resources, leading to additional possible sources of contention for memory or functional units. Hardware accelerator technology is also starting to be used in coprocessors. If configured, the bus bandwidth with the processor becomes a new possible source of bottlenecks.

Unfortunately tools have not kept pace with these complex architectures. Moreover, tool use has remained fragmented, with little interoperability among the different software systems. As a result, the productivity of developers is hampered as they have to worry about understanding the implications of details of the underlying hardware, and learning how to use a number of tools. Studies have confirmed that a lack of appropriate tools can negatively impact the productivity of HPC researchers [25].

Performance analysis starts once an application is free from errors and stable enough to collect performance information. A performance analysis methodology consists of exploring the reasons for performance degradation, and quantifying how this affects the overall program. The information gained in this process gives hints to the user on what changes are needed to tune the application. A performance tuning methodology should identify, localize, repair, verify and vali-

date the performance problems. The performance tuning process forms a cycle of analysis and modifications that frequently has to be repeated many times until the performance reaches a satisfactory level. In practice, the cycle ends when the application developer either runs out of resources (time, or budget) or ideas for improvement.

Several different approaches to performance collection play a significant role in supporting attempts to understand the behavior of an application. For example, *profiling* aggregates the results of event tracking for phases in the code or for the entire execution of the program, while *tracing* can keep track of every single instance of an event at run time. Both approaches can be used to summarize and measure a variety of different kinds of events, but the higher accuracy means higher overheads and perturbations. One way to implement profiling is via program instrumentation, which inserts calls to the program to measure different events. The advantage is that it is a precise method to directly measure events, but if it is not done carefully it might incur code bloating, interference with compiler optimizations (if done at the source code level) and over-instrumentation. If instrumentation is not performed carefully, this can lead to very large trace files and may degrade overall system performance, also negatively affecting other applications sharing the same resources. Studies have shown that the overheads of tracing [17] are particularly severe in systems consisting of thousands of processors with a shared file system.

Sampling is a way to minimize overheads by using statistical sampling of aspects of the application's execution behavior. Data can be sampled at fixed intervals in time or may be based on hardware or software event thresholds. If there is no hardware support for sampling, it requires the interruption of the application processes to poll for events, resulting in high overheads for context switches. When sampling is used, it is challenging to map low level performance information back to different language constructs in the source code (such as loops, or parallel constructs). In addition, sampling entire applications can yield large amounts of low level information that can be overwhelming for the user. Some processor architectures, such as the Itanium 2 processors [15] and the PowerPC [23], support sampling by providing specialized hardware such as performance monitoring units.

When applications are deployed on systems with thousand of processors, it is critical to provide a scalable strategy for generating performance data. A suitable approach must minimize perturbations and reduce the amount of data gathered, while at the same time providing a significant coverage of the important code regions. A solution might consist of mixing sampling, profiling and tracing to exploit their strengths for solving performance problems. A combination of sampling when there is hardware support and selectively instrumenting important regions of code at the right granularity would seem to be essential.

Compilers can help the user to focus on areas of interest and minimize the instrumentation overhead by providing a selective instrumentation capability. Compilers can also be useful in the interpretation and relation of low level performance information back to the source code since they are in control of the translation process from source to object code. However, research compilers have not been used in this way because they have not been sufficiently robust, have not been kept up to date with advances in architectures and operating systems, or possess limited instrumentation capabilities. On the other hand, compilers in general lack portable instrumentation APIs that support interoperability of performance tools. Very few standard APIs exist in this area.

PDT [12] is a toolkit that was designed in an attempt to overcome the lack of a portable compiler instrumentation API with support for C, C++, Fortran and OpenMP. It gathers static program information via a parser and represents it in a portable format suitable for use in source code instrumentation. But this approach does not permit full exploitation of compiler technology; in particular, it does not provide any insight into the compiler translation, or enable the evaluation of any assumptions the compiler makes when deciding to apply a given transformation (which includes metrics used to evaluate static cost models). A compiler that exposes such information may help to facilitate the interpretation of performance data.

Compile-time instrumentation has several advantages over both source-level and object-level instrumentation. Compiler analysis can be used to implement selective instrumentation derived from static analysis (as is shown in this paper), thereby reducing the perturbation of the instrumented application. It can also maintain a mapping that relates performance data back to the source code. Moreover, the instrumentation can be performed during different compilation phases, allowing some optimizations to take place before the instrumentation. These capabilities play a significant role in the reduction of instrumentation points, in reducing the instrumentation overhead and the size of performance trace files, and in improving a user's ability to determine the impact of program optimizations.

1.1. Contents of the paper

In this paper, we introduce Copper (Compilation and Optimization with Performance Feedback), a performance tuning environment that we have built in collaboration with the providers of several popular performance tools. We also introduce a methodology that uses these tools together for bottleneck analysis that is enabled by Copper and describe its use. The paper is organized as follows. We first discuss related work in Section 2. Section 3 next introduces our integrated performance environment. Then, Section 4 presents our tuning methodology. Section 5 describes some case studies and Section 6 summarizes our conclusions and future work.

2. Related work

Malony [14] emphasizes the importance of the scientific method of systematic testing of performance hypotheses by measuring the phenomena, performing an analysis of collected data, and modeling the empirical results, to the process of performance analysis. He delineates an idealized model of a parallel performance evaluation environment which highlights the process of successive refinement of a hypothesis about performance behavior based on observation and accumulation of knowledge. Pancake [18] and Wolf and Mohr [27] present a conceptual framework that describes this cycle in more detail.

Different tools have tried to address different aspects of the tuning cycle, and for a variety of programming languages and architectures. Early tools such as the Fortran D editor [5] combined with SvPablo [22] explored supporting performance tuning for distributed memory machines where high level languages such as HPF were deployed. SUIF Explorer [11], an interactive and interprocedural parallelizer for C, provides profiling infrastructure to assist in the parallelization process with metrics such as parallel coverage and granularity. Ursa Minor [19] and Interpol provide a system and a methodology for dealing with OpenMP performance problems in an environment [7] that supports no more than 32 processors in a shared memory system. OpenSpeedShop [24] provides an environment for performance tuning, that supports timing metrics for MPI. Since it relies on object instrumentation it does not have static analysis capability and does not address scalability issues in their instrumentation. The PTP project [26], is built on top of Eclipse, and is an environment that includes a parallel debugger, static analysis tools and interfaces to performance tools but most of these tools are in the early stages of integration and do not yet address scalability issues in their instrumentation.

Programs may be instrumented at different levels of a program's representation: at the source code level, at multiple stages during the compiler translation, at object code level, and embedded within the run time libraries. All of these techniques have inherent advantages and disadvantages with respect to their ability to enable the instrumentation to be performed efficiently and automatically, the type of regions that can be instrumented, the mapping of information to the source code, and the support for selective instrumentation. Among the many performance tools with instrumentation capabilities are KOJAK [16] and TAU [14], which rely on PDT [12] and OPARI [16] to perform source code instrumentation, and Dyninst [2] which performs object code instrumentation. Strategies for static scalable instrumentation have been explored with some of these tools.

OpenSpeedShop [24] and Paradyn support Dyninst and DPCL [21] but instrumenting at the object code level implies a loss of the semantics of the program (e.g. loop level information). The EP-Cache project [13] uses the NAG compiler to support instrumentation at procedure and loop levels, but it is a closed source system and the developers have not explored ways to improve instrumentation via compiler analysis. Intel's Thread Checker [20] performs instrumentation to detect semantic problems in an OpenMP application but it lacks scalability, as it heavily instruments memory references and synchronization points. Other tools such as PerfSuite [9], Sun Analyzer [29] and Vtune [28] rely on sampling to provide profile data. Although sampling is a low overhead approach, it requires extra system resources, in some cases needing extra threads/processors to support processor monitoring units, and it focuses on low level data gathering.

3. Copper, an integrated tuning environment

We have created a tuning environment called *Copper* that consists of the OpenUH compiler and the performance tools TAU, KOJAK and PerfSuite. Our environment enables the user to edit source code, invoke the compiler and tools, and visualize the application structure annotated with high-level performance information. The environment supports tech-

niques for selective, automatic instrumentation, minimizing perturbation and the amount of data generated while maximizing diagnostic value. At runtime, PerfSuite guides the initial instrumentation by providing low-overhead application profiles that already narrow down the most important bottlenecks. This information combined with static analysis is used by the compiler to guide TAU and KOJAK profile and trace specific regions of the application, collect relevant performance data, and automatically identify performance problems by applying a pattern analysis to the stream of performance-relevant events. The benefits of this environment includes an automated solution for scalable instrumentation and a framework together with a methodology to understand and solve performance problems. To do so, we have developed and implemented the APIs needed to enable them to interact. Copper can optimize applications written in C, C++, Fortran 95, OpenMP, MPI and Hybrid (OpenMP + MPI) codes. Copper gives its users the ability to tune their applications following a general methodology, instead of requiring the separate and laborious collection of information from a variety of independent tools. Each tool in Copper has strengths that can be exploited to improve the overall tuning process, and defines a tuning methodology. In the following subsections we describe the tools separately.

3.1. The OpenUH compiler and Dragon analysis tool

The OpenUH [10] compiler is a branch of the open source Open64 compiler suite for C, C++, and Fortran 95, supporting the IA-64, IA-32e, and Opteron Linux ABI and standards. OpenUH supports OpenMP 2.5 and provides complete support for OpenMP compilation and its runtime library. The major functional parts of the compiler are the front ends, the inter-language interprocedural analyzer (IPA) and the middle-end/back end, which is further subdivided into the loop nest optimizer (LNO), auto-parallelizer (with an OpenMP optimization module), global optimizer, and code generator. OpenUH has five levels of a tree-based intermediate representation (IR) called WHIRL to facilitate the implementation of different analysis and optimization phases. They are classified as being Very High, High, Mid, Low and Very Low levels, respectively. Most compiler optimizations are implemented on a specific level of WHIRL. For example, interprocedural array region and dependence analysis is implemented in the high level WHIRL. OpenUH has been enhanced in many ways to support the requirements of the integrated environment by supporting the different tools APIs.

3.2. Tools analysis and utilities (TAU)

TAU [14] is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, Java or Python, with MPI, OpenMP and Pthreads.

The results are displayed in aggregate and single node/context/thread forms, thus identifying the performance bottleneck. A number of profiling modes exists including: statistical sampling of the program counter or call stack, hardware counter sampling, instruction counting and timer based instrumentation. The time and hardware counter data are analyzed by ParaProf and PerfExplore, both parts of the TAU toolkit. In our environment we use TAU to visualize performance data, derive metrics, and manage the performance data that supports our tuning methodology.

3.3. KOJAK

The KOJAK trace-analysis environment [16] is an automatic performance evaluation system for MPI, OpenMP, and hybrid applications. KOJAK generates event traces from running applications and automatically searches them offline for execution patterns indicating inefficient performance behavior. It can compare the behavior of multiple versions. Its main feature is its ability to automatically diagnose the reasons for inefficient runtime behavior on a high level of abstraction. Most of the functionality of KOJAK is in the process of being upgraded to the SCALASCA system [1] (SCalable performance Analysis of LArge SCale Applications), where the tracing analysis is done in parallel.

3.4. PerfSuite

PerfSuite [8,9] is a collection of tools, utilities, and libraries that together provide the user with several options for performance monitoring including support for hardware performance counting and profiling of application programs. PerfSuite consists of a set of command line utilities that includes support for statistical profiling, hardware performance counting, deriving high-level performance metrics, access to machine level characteristics, and customizing performance experiments.

3.5. Tools interactions

The OpenUH compiler infrastructure has been extended to perform selective instrumentation: it deter-

mines the regions of interest while at the same time, reducing the instrumentation overheads by inserting appropriate instrumentation calls for these regions only. We created a set of libraries implemented via TAU and KOJAK, to link the compiler instrumentation with the performance tools. The library was designed with an API that support the main requirements of different performance tools with context and mapping information to the source code and the intermediate representation of the compiler.

In our environment, PerfSuite has the role of providing a first assessment of the behavior of an application with very low overhead. PerfSuite is moreover able to return information on regions of code that we cannot instrument with TAU or KOJAK, due to the significant overheads the latter tools would incur. The compiler automatically instruments the application on selected regions and use TAU to generate profile regions of interest in the application and detect the location of bottlenecks, using the callstack to recreate the program's dynamic callgraph. TAU data management, visualization and derived metrics capabilities help us to visualize the profile data and perform the analysis. We use KOJAK for tracing and to analyze large trace files to find performance problem patterns. KOJAK is able to detect *when* the performance problem occurs in time. We use the expert component of KOJAK to search for known patterns that indicate a certain type of performance problem. Then, the OpenUH compiler can be used to analyze the performance data and compare it to different levels of translation/optimization of the source code to help explain *why* we have a given bottleneck. Figure 1 shows our integrated tuning environment.

3.6. Compile time instrumentation

One of the keys to the integration of these components is the ability of the compiler to instrument source code. The revised version of OpenUH provides

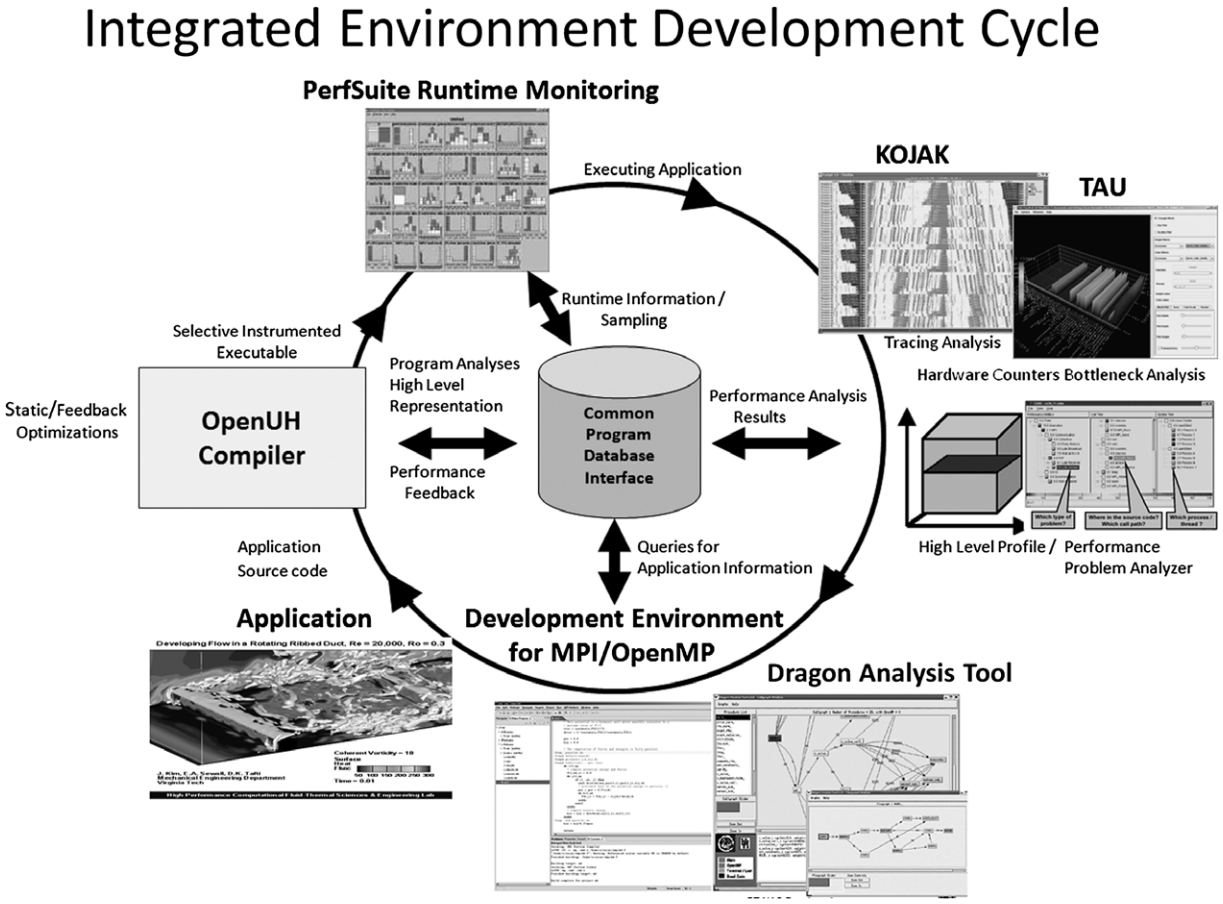

Fig. 1. Integrated performance tuning environment.

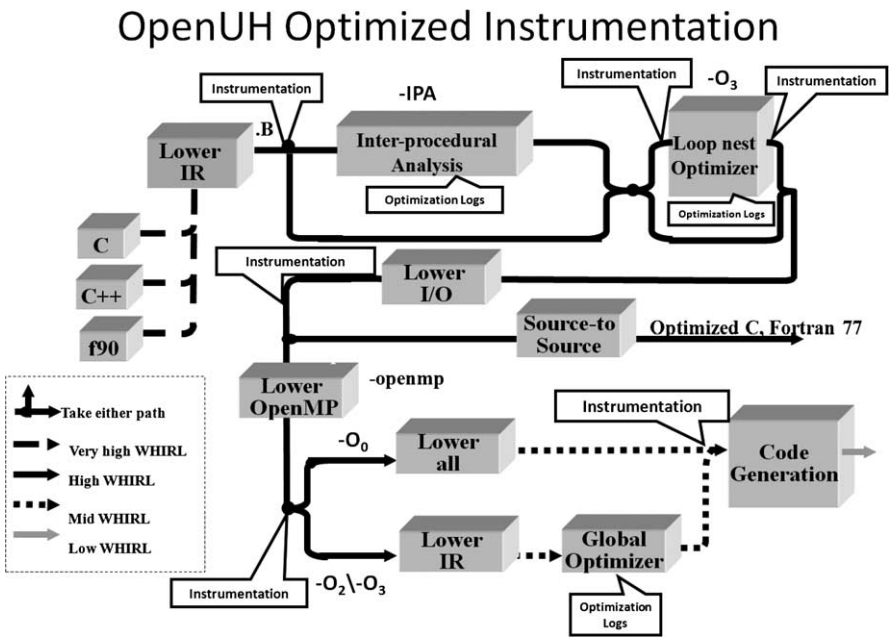

Fig. 2. Multiple stages of compiler instrumentation.

a complete compile-time instrumentation module covering different compilation phases and different program scopes. We have designed a compiler instrumentation API that can be used to instrument a program. It is language independent to enable it to interact with performance tools such as TAU and KOJAK. The instrumentation module in OpenUH can be invoked at six different phases during compilation, which come before and after three major stages in the translation: interprocedural analysis, loop nest optimizations, and SSA/DataFlow optimizations. Figure 2 shows the different places in the compilation process where instrumentation can be performed, along with the corresponding intermediate representation level.

For each phase, the following kinds of user code region[1] can be instrumented: functions, conditional branches, switch statements, loops, callsites, and individual statements. Each type of code region is further divided into subcategories when possible. For instance, a loop may be of type *do loop* or *while loop*. Conditional branches may be of type *if then*, *if then else*, *true branch*, *false branch* or *select*. We also have *switch* statement regions, *callsites*, and statement level instrumentation (that can instrument floating point operations). MPI operations are instrumented via PMPI so that the compiler does not instrument these callsites. OpenMP constructs are handled via runtime library instrumentation, where the fork and joint events, implicit and explicit barriers [3] are captured. All these types of instrumentation are related to each other. For example, procedure and control flow instrumentation is essential to relate the MPI and OpenMP-related output to the execution path of the application, or to understand how constructs behave inside these regions.

The compiler instrumentation is performed by first traversing the intermediate representation of an input program to locate different program constructs. The compiler inserts instrumentation calls at the start and exit points of constructs such as procedures, branches and loops. If a code region has multiple exit points (for example, goto, stop or return statements may provide alternate exit points), they will all be instrumented. A brief description of the API can be found in [4].

4. Tuning methodology and selective instrumentation

We have defined an iterative approach to application tuning based on the COPPER environment that includes bottleneck analysis; it leverages the different sources of information available. Our methodology attempts to narrow down the location of bottlenecks, then to determine when they occur and what is causing them. We use a top down approach where we first per-

[1] Here, we consider a code region to be a segment of the source code with a single entry and with one or multiple exits.

form a performance analysis of the entire application and then identify and focus on specific regions of code that appear to warrant closer investigation. We use the compiler's static analysis (as described in Section 4.2) to reduce program perturbation due to instrumentation.

We use information from the compiler to understand what optimizations or transformations are being applied, together with static analysis, source-to-source transformations, and optimization logs which are useful to understand and relate low level performance results to the source code. Our methodology accounts for every cycle that we measure to determine whether or not it is a hot spot. It also helps to find higher level parallel inefficiencies such as load imbalances, synchronization problems, and communication overhead. Our approach focuses on finding coarse level inefficiencies at the procedure level and then performs finer grain measurements to understand the problems in more detail. After we have an understanding of a problem and possible solutions, we also validate our modifications using a performance algebra, which is a performance comparison between two different tuning experiments that help the application developer to understand the regions of code affected by the changes and to deduce why a given optimization worked or did not work.

4.1. Description of the methodology

Our methodology is part of an application tuning cycle that consists of five application runs that enable scalable instrumentation and feedback optimizations. Our methodology consists of a run for monitoring the application, three runs for profiling important regions of the code and one run for tracing MPI and OpenMP behavior. All runs differ from each other on the information they collect and their method of collection (sampling, instrumentation). From run to run we narrow down the bottlenecks to specific regions of interest. After these five runs, we have two extra steps in the methodology to help the user interpret the performance results and search for a solution to the performance problem.

1. *First run. Performance monitoring.* In the first run of the application we use PerfSuite to monitor the application and provide results in profiling mode. This serves the purpose of getting an overall assessment of the application via flat profiling. This also servers the purpose of identifying regions of code that should not be instrumented due to high overheads. These are, in particular, small regions of code that are executed many times.
2. *Second run. Profiling with selective instrumentation.* Our selective instrumentation method is designed to create a scoring mechanism for regions of interest based on their importance in the code and callgraph. We want to avoid instrumenting regions of code that have small weights (small in size) and are invoked many times. In this run we focus on procedure level instrumentation. The approach is based on the interprocedural analysis phase in the compiler, which statically calculates a cost for making inlining decisions. The goal of this run is to determine where the bottlenecks are located. Depending on whether the application is integer or floating point based, we select: Wall Clock Time, Total Cycles (equivalent), Total Stall Cycles and either number of floating point or integer instructions. Using this approach we attempt to locate procedures that are running inefficiently. The formula to calculate the inefficiency for this purpose is:

$$\begin{aligned} &\mathit{Inefficiency} \\ &\quad = \mathit{Floating\_Point\_Operations} \\ &\qquad * (\mathit{Total\_Stall\_Cycles}/\mathit{Total\_Cycles}). \end{aligned}$$

   This formula is calculated for each region being measured. The region with the highest inefficiency is the region that the programmer and compiler should focus on optimizing.
3. *Tracing.* While using the same selective instrumentation, we use the tracing capability of KOJAK to locate bottleneck patterns in OpenMP and MPI. In this case, we locate regions that have synchronization problems, load imbalances, a implicit barriers. The 2nd run focuses on obtaining inefficiencies at the processor/hardware counters level. The 3rd run focuses on parallel performance problems.
4. *Fourth run. Collection of in-depth performance information for the inefficient regions in profiling mode.* In this run we do not turn on all the instrumentation, but only instrument specific code regions or procedures of interest, collecting more fine grain information. This will include instrumenting loops, branches, calls, and possibly individual statements. On systems with few hardware counters we might need multiple runs or use multiplexing to perform the processor bottleneck analysis.
   The general formula we have adopted for this purpose is the following:

$$
\begin{aligned}
&\textit{Total Stall Cycles}\\
&\quad = \textit{Level 1 Data Cache Misses}\\
&\qquad + \textit{Branch Misprediction}\\
&\qquad + \textit{Instruction Misses}\\
&\qquad + \textit{StackEngine stalls}\\
&\qquad + \textit{Floating Point Stalls}\\
&\qquad + \textit{Pipeling Inter}\\
&\qquad - \textit{Register Dependencies}\\
&\qquad + \textit{ProcessorFrontend Flushes}.
\end{aligned}
$$

We primarily collect performance data for stall cycles from the Level 1 data Cache Misses and Floating Point Stalls (on the Itanium, the floating-point registers are fed directly from level 2 cache). If 90% of the stalls are due to them (we determine this by a comparison with the total stall cycles calculated in the second run), we ignore other sources of stalls in the formula. If that is not the case, we will have to perform additional runs to calculate the other components of the formula. The 90% is a rule of thumb based on behavior seen in different applications.

5. *Fifth run.* In the same way as the fourth run, we use hardware counters to perform the memory bottleneck analysis based on the following formula:

$$
\begin{aligned}
&\textit{MemoryStalls}\\
&\quad = (\textit{L2\_data\_references\_L2\_all}\\
&\qquad - \textit{L2\_misses})\\
&\qquad * \textit{L2\_Memory\_Latency}\\
&\qquad + (\textit{L2\_misses} - \textit{L3\_missed}) * 10\\
&\qquad + (1 - \textit{Number\_of\_remote\_memory\_}\\
&\qquad\quad \textit{accesses}/\textit{L3\_misses})\\
&\qquad * \textit{Local\_Memory\_Latency}\\
&\qquad + (\textit{Number\_of\_remote\_memory\_}\\
&\qquad\quad \textit{accesses})\\
&\qquad * \textit{Remote\_memory\_access\_latency}\\
&\qquad + \textit{TLB\_misses} * \textit{TLB\_miss\_penalty}.
\end{aligned}
$$

This formula is specific to Itanium 2 processors and needs to be modified for other architectures. The coefficients in this formula are the different latencies (in cycles) for the different levels of memory for the Itanium 2 processor (Madison), and the interconnection latencies of the SGI NumaLINK 4 for local and remote memory accesses (see Section 5). To calculate this formula in one run we will need to enable multiplexing of hardware counters. If we prefer to disable multiplexing because it will produce less accurate results as a result of sharing hardware counters to collect the data then we will require two runs. The value for remote memory latency accesses is an estimation of the worse case scenario for a pair of nodes with the maximum number of hops and is system dependent.

The additional steps to interpret the data gathered in the previous runs and to find a solution to the performance problem are as follows:

1. *Interpreting performance results.* After performing the performance analysis, we relate the results with the compiler optimization log, which records the optimizations applied along with the and source-to-source translations. This will help the user understand what happens between the translation of the source code and the object code. This is useful to interpret low level performance data and see if the performance bottlenecks are due because of a given compiler optimization or lack of optimizations.
2. *Search for a solution to the performance problem.* We can perform experiments in order to overcome the identified inefficiency and evaluate the outcome by using performance algebras. A performance algebra as provided by KOJAK allows us to compare two different version of a code, and see the differences in their performance in detail. This is important if we want to understand and validate why a given optimization works (or otherwise).

4.2. Selective instrumentation analysis

We take advantage of the interprocedural analysis within the compiler to reduce the number of instrumentation points. Here the compiler performs inlining analysis, which attempts to determine where the program will benefit from replacing a procedure call with the actual code of the called procedure's body. As part of this, the compiler must determine if a procedure is invoked frequently and whether the caller and

callee meet certain size restrictions in order to avoid code bloat. We adapted this methodology to enable selective instrumentation, for which we have defined a cost model in the form of scores to evaluate the above conditions. We avoid instrumenting any procedure that meets the criteria for inlining. We do instrument procedures that are significant and are infrequently called and have large bodies. We call a procedure significant if it contains many callsites in the form of out edges in the callgraph.

Our cost model consists of three metrics in the form of instrumentation scores. The first metric computes the *weight* of the procedure using the compiler's control flowgraph, which is defined as $PU_{weight} = (5 * \textit{total basic blocks}) + \textit{total statements} + \textit{total callsites}$. As can be seen, this metric puts emphasis on procedures with multiple basic blocks. If runtime information is available, the PU_{weight} formula will use the number of times or *effective* number of basic blocks, statements and callsites invoked at runtime. The other metric we use is the frequency with which a procedure is invoked, taking their position within loop nests into account. The formula used is: $PU_{loop\text{-}score} = (100 - \textit{loopnest level}) * 2048$. This formula gives higher scores to procedures invoked with fewer nesting levels. The third metric is a score that quantifies how many calls exist within a procedure: $PU_{callsite\text{-}score} = (\textit{callsites in callee}) * 2048^2$. This formula gives a small score to procedures invoked as leaf nodes in the callgraph or that have few calling edges. The constants of the formulas were determined empirically based on the inlining algorithm of the compiler which was tuned to avoid under or over-inlining. Our assumption here is that important procedures are connected with others, and thus are associated with several edges in the callgraph. The overall score used to decide whether we will instrument a procedure is as follows:

$$\textit{Instrumentation Score} = PU_{weight} + PU_{loop\text{-}score} + PU_{callsite\text{-}score}.$$

Our strategy for computing this score means that we will favor procedures with large weights, those invoked few times and with multiple edges connecting them to other procedures in the callgraph. As noted above, this helps us to avoid instrumenting small procedures invoked at high loopnest levels and procedures that are leaf nodes in the callgraph.

We may define a threshold value so that procedures whose scores are below it will not be instrumented. The threshold can be changed depending on the size of the application, in order to avoid over or under-instrumentation.

$$\textit{Instrument Procedure} < \textit{Threshold} < \textit{Do not Instrument}.$$

Table 1
Instrumentation scores for the BT OpenMP benchmark

Procedure	Weight	Loop score (loop level)	Callsite score (callsites)	Instrumentation score
matvecsub	23	198,656 (3)	0 (0)	198,679
binvchrs	240	198,656 (3)	0 (0)	198,896
binvrhs	115	198,656 (3)	0 (0)	198,771
matmul_sub	27	198,656 (3)	0 (0)	198,683
lhsinit	57	198,656 (3)	0 (0)	198,713
exact_solution	23	196,608 (4)	0 (0)	196,631
adi	45	202,752 (1)	20,971,520 (5)	21,174,317
x_solve	278	204,800 (0)	41,943,040 (10)	42,148,118
y_solve	278	204,800 (0)	41,943,040 (10)	42,148,118
z_solve	278	204,800 (0)	41,943,040 (10)	42,148,118
main	459	204,800 (0)	58,720,256 (14)	58,720,256

Table 1 contains the instrumentation scores for some of the procedures in the BT OpenMP benchmark from the NAS parallel benchmarks. It shows that procedures corresponding to leaf nodes in the callgraph have a low instrumentation score. Heavily connected nodes in the callgraph (those that have several callsites) are among the ones with the highest score, as is to be expected. If we set the threshold to be 204,800 (the score of an empty procedure with no callsites being invoked outside a loop), then we will not instrument the following procedures: *matvecsub, binvchrs, matmul_sub, lhsinit, exact_ solution*. The following ones will be instrumented: *adi, x_ solve, y_solve, z_solve, main*. Note that the weight recorded is a very small number when compared to the loop and callsite score. This is the case for static analysis, but runtime information can be used to improve this aspect of the model by substituting the actual number of times a given basic block, statement or callsite was executed.

5. Case studies

Our case studies were conducted on an SGI Altix 300, which consists of Itanium 2-based nodes connected via the NumaLINK 4 high speed memory interconnect. The Itanium processor is a 64-bit register-rich explicitly-parallel architecture. It supports predi-

cation, speculation, and branch prediction. The architecture implements 128 integer registers, 128 floating point registers, 64 one-bit predicates, and eight branch registers. The floating point registers are 82 bits long in order to preserve precision for intermediate results. The Itanium processor supports long word instructions. Each 128-bit instruction word contains three instructions, and the fetch mechanism can read up to two instruction per clock cycle. The Itanium 2 (Madison) processor has 16 kB of Level 1 instruction cache and 16 kB of Level 1 data cache. The L2 cache is unified (both instruction and data) and is 256 kB. The Level 3 cache is also unified. The different characteristics of the main components of the Itanium 2 processor can be measured via the hardware counters.

The Altix 300 used here is a distributed-shared memory system consisting of 8 nodes with two Itanium 2 processors each. A single address space is seen by all the processors/nodes and its global memory is based on a cache-coherent Non-Uniform Memory Access (ccNUMA) system implemented via the NUMAlink4. Each node has a local memory; two nodes are connected via a memory hub to form a computational brick (C-brick). The C-bricks are connected via memory routers in a hierarchical topology.

5.1. Application description

To evaluate our selective instrumentation strategy, we used the NAS parallel benchmarks which are a small set of programs derived from computational fluid dynamics (CFD) applications. For the evaluation of our performance tuning methodology we used the production version of a cloud resolving model which implements the three dimensional version of the Goddard Cumulus Ensemble (GCE) model. The application domain can be partitioned in a one or two dimensional decomposition with halos that partially overlap with the domains. The halos are updated in a step after the computation stage. The code is implemented in MPI. We tested it with different combinations of 1D and 2D grid layouts with 16 processors (e.g. $1 \times 1, 1 \times 2, 2 \times 1, 2 \times 2, 2 \times 4, 4 \times 2, 8 \times 2, 2 \times 8$).

5.2. Evaluating selective instrumentation

We applied the selective instrumentation algorithm to six benchmarks from the NAS parallel benchmarks in both MPI and OpenMP implementations. Our experiments used the class A problem size and were conducted on an SGI Altix 300 with 16 Itanium 2, 1.6 GHz processors, and the NUMAlink interconnect. Figures 3 and 4 show the overheads incurred when performing full procedure instrumentation versus performing selective instrumentation with OpenUH. The TAU profiling libraries were used, and TAU's THROTTLE switch was turned off for the MPI benchmarks. The purpose of THROTTLE is to disable the instrumentation library at runtime when a procedure reaches a given threshold. We disabled TAU THROTTLE to get an accurate assessment on the effectiveness of our selective instrumentation approach. When THROTTLE disables a in-

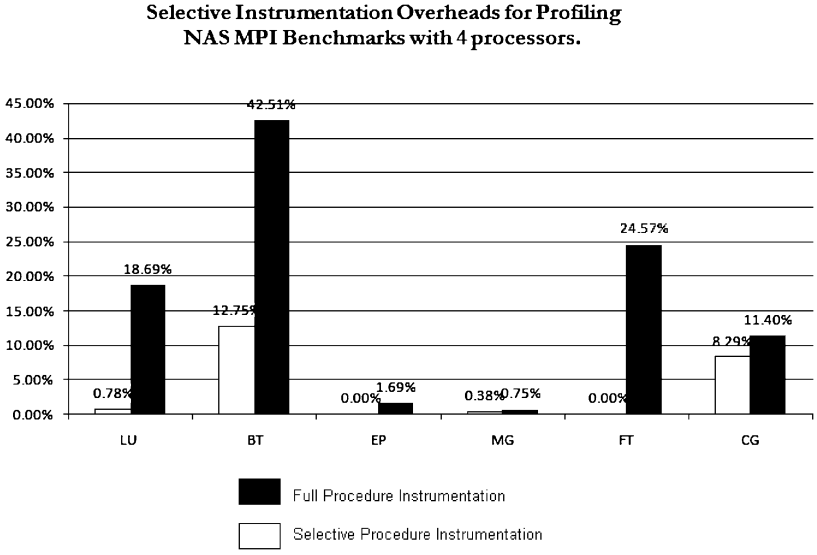

Fig. 3. Selective instrumentation in the NAS MPI benchmarks.

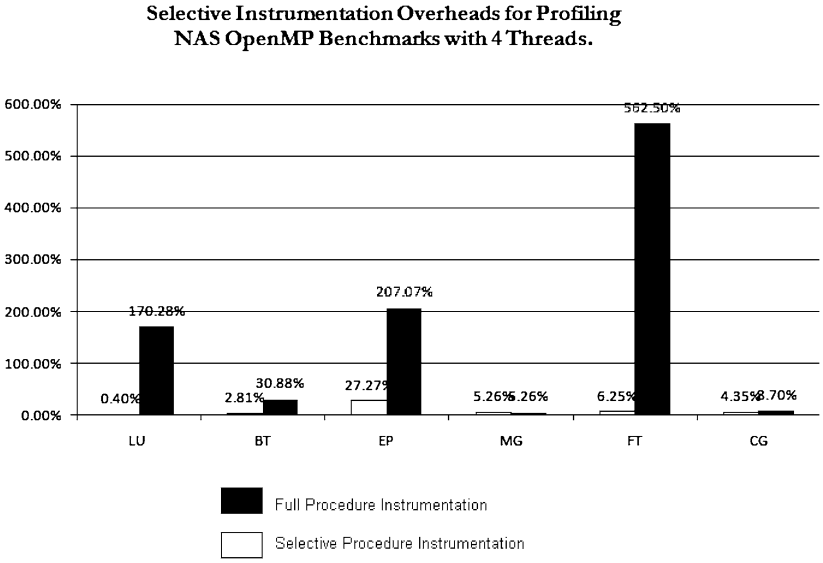

Fig. 4. Selective instrumentation in the NAS OpenMP benchmarks.

strumentation call, its results get aggregated to the parent procedure, making it hard to differentiate to which region belongs (parent or children) belongs the performance information.

Because of the massive overheads of full instrumentation in the case of OpenMP, we turned on the TAU THROTTLE for NUMCALLS = 300 and PERCALL = 300,000 environment variables. Selective instrumentation reduces the overheads by an average of 90 times in the OpenMP version, even when TAU's THROTTLE is enabled in full instrumentation.

As the figures show, the FT-OMP benchmark has particularly high overheads for full instrumentation compared to the rest of the benchmarks. This is because it invokes the procedures *fttz2* and *cfftz* a significant number of times. We note that *cfftz* calls *fttz2*. Our selective instrumentation score for *cfftz* is 202,817 and *fftz2* is 198,772, which is below our instrumentation threshold. As a result, *fftz2* is not instrumented; nor is *cfftz*, since it only has one callsite that is not being instrumented. In the LU OpenMP benchmark we do not instrument the procedures *buts*, *jacu*, *blts*, *jacld* and *exact*: these are leaf nodes in the callgraph with no callsites. However these procedures play a significant role in the computation. In order to ensure that these procedures are instrumented we will need to adapt the procedure weight to account for feedback information collected during runtime (we are working on giving the user an easy way to do this via compiler options). The higher overheads in OpenMP instrumentation versus MPI are due to memory contentions and locks on memory locations where the performance data is stored and modified. On cc-NUMA architectures, this contention becomes a problem due to cache line invalidations and remote memory accesses.

We also applied our algorithm to an MPI implementation of a cloud-resolving model code [6] using a grid size of $104 \times 104 \times 42$ and 4 MPI processes for our experiment. Our selective algorithm determined that we should not instrument three leaf procedures in the callgraph. We were able to reduce the profiling overhead from 51% to 3% based on this alone, making out performance measurements more accurate and with less perturbations to the application.

5.3. Performance analysis for the BT MPI benchmark

We now apply our performance tuning methodology to conduct performance analysis on the BT MPI benchmark. We use the profiling capabilities of TAU and PAPI to gather application information. Figure 5 shows the measured wall clock time, in seconds, of different procedures in the application. We observe that the three solvers *x_solve_cell*, *y_solve_cell* and *z_solve_ cell* are the time-consuming procedures in the application. We also observe that the solvers have good load balance because there is a small variance among them in wall clock time, as shown in the 3D graph on the left of Fig. 5.

Figure 6 shows that the three solvers are executing similar amounts of floating point operations (the difference is within 1.77%) but that *z_solve_cell*

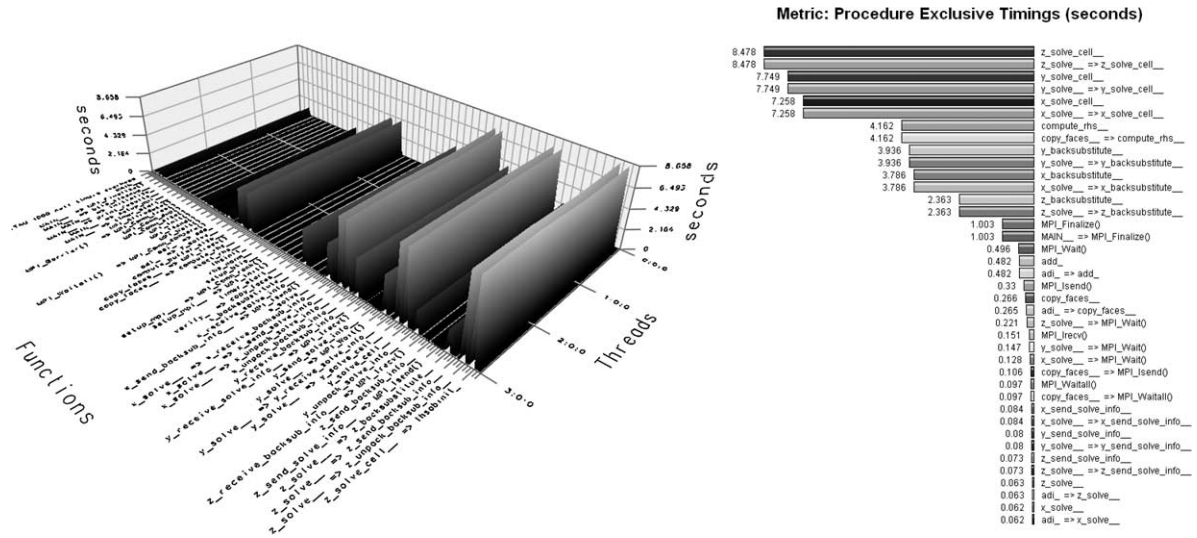

Fig. 5. Wall clock time for the BT MPI benchmark.

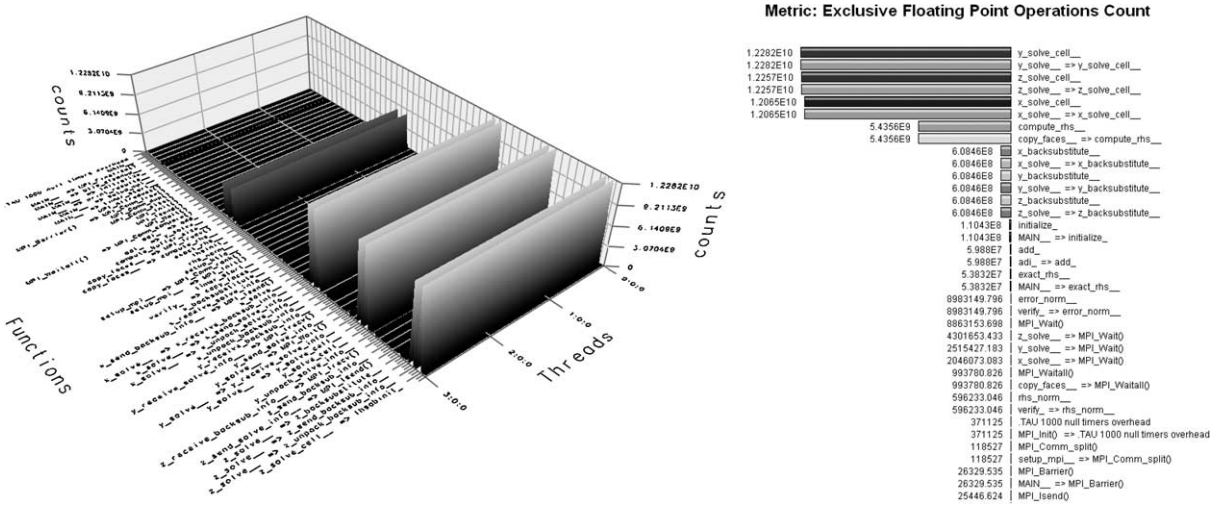

Fig. 6. Floating point operations for the BT MPI benchmark.

and *y_solve_cell* take 14% and 6% longer than *x_solve_cell*, respectively. In order to understand this difference we measured stall cycles using hardware counters to derive the inefficiency metric. We find that this metric can filter procedures with high stalling ratios and with a large number of floating point computations. Figure 7 shows the inefficiency metric for the BT MPI benchmark when using 16 processors. We can see a correlation between the wall clock time and the inefficiency metric. Here the *z_solve_cell* and *y_solve_ cell* runs 13% and 6% more inefficiently than *x_solve_cell*. We conclude that *z_solve_cell* and *y_solve_cell* are taking longer because they are stalling more frequently, and to an extent that is proportional to their wall clock time. The next step is to investigate why this is the case.

Figure 8 shows that in the case of the *z_solve_ cell*, performance degradation is due to L1 data cache stalls. When we look at its performance data in more detail, we see that the TLB misses and the L1 data misses are related to each other, since most of the address calculations needed for array accesses are carried out in the L1 cache. In other words, *z_solve_cell* takes longer because it is not accessing data efficiently. On the other

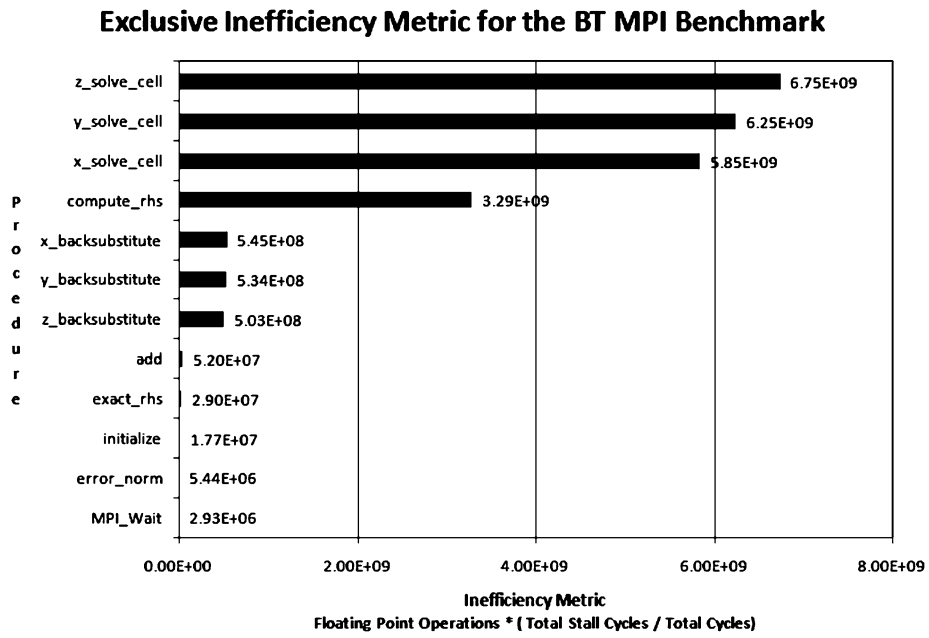

Fig. 7. Inefficiency metric for the BT MPI benchmark.

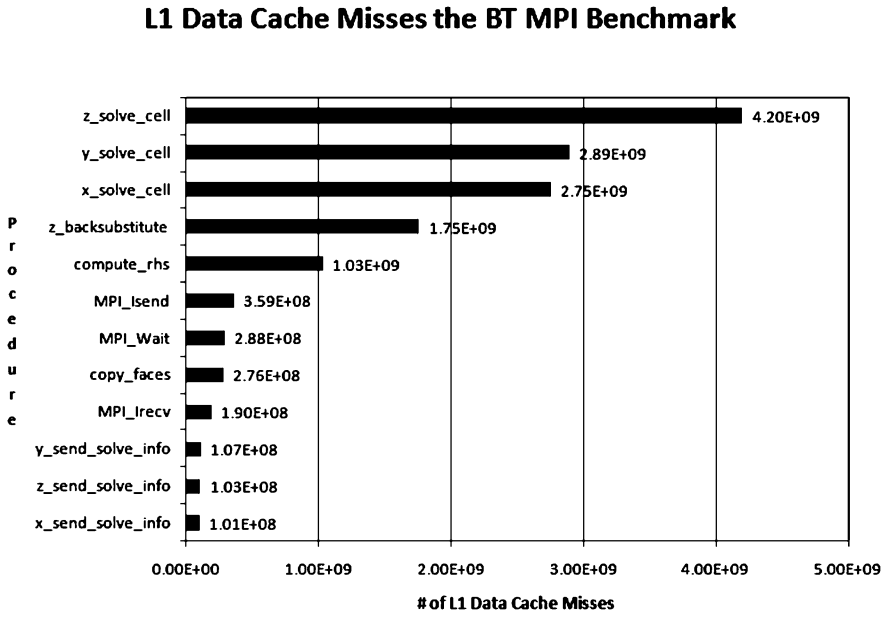

Fig. 8. Data cache 1 and TLB misses for the BT MPI benchmark.

hand, Fig. 9 shows that the stalls for *y_solve_cell* are primarily due to L2 cache stalls. We see also that the floating point stalls are also a problem with other routines.

We can safely conclude that the reason why portions of the BT benchmark take longer than other procedures is related to the way they are accessing its data. With a little background insight, the information gathered on stalls clearly identifies the source of the problem.

5.4. Performance analysis of the cloud code

Our next case study is the cloud resolving model code implemented using MPI and provided by NASA.

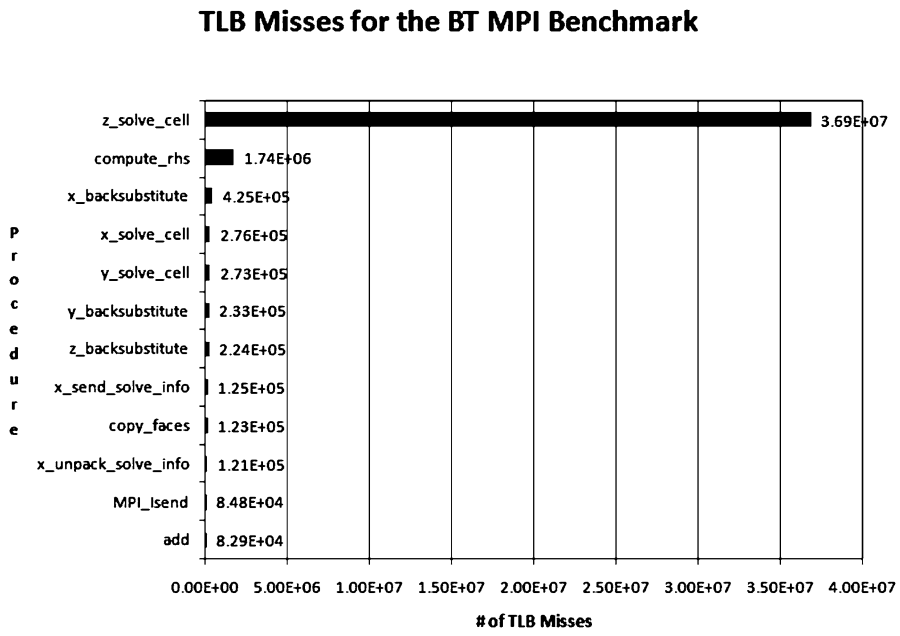

Fig. 9. Inefficiency metric for the BT MPI benchmark.

We applied the methodology described in the previous section. The experiments were conducted on an SGI Altix 300, with 16 Itanium 1.6 GHz processors. Data was collected using the TAU performance library based upon the OpenUH instrumentation infrastructure. After applying the compiler's algorithm for selective instrumentation, we were able to reduce the instrumentation overheads from 51% to a mere 3%, without using TAU THROTTLE. Our selective instrumentation analysis detected three procedures that have small weights (small size), were leaf nodes in the callgraph (no callees) and were invoked many times (their callsites were inside deeply nested loops). By carefully eliminating these procedures during the instrumentation process, we were able to minimize the perturbation of instrumentation and get very accurate results.

Figure 10 shows the time-based profiling results of the cloud resolving model code. The *irrad* and *solir* routines belong to the radiation module of the simulation. These routines are called every fourth time step in the simulation due to their high computational cost. The 3-dimensional graph in Fig. 10 shows the exclusive time taken by these routines, comparing it to the remainder of the code. The 3-dimensional figure also shows that the work is balance across the processors.

In the second run we calculate the inefficiency metric: the results are shown in Fig. 11. We used this data to calculate the inefficiency metric. Figure 11 shows that procedure *solir* runs less efficiently than *irrad* due to its higher number of stall cycles. Both procedures are responsible for 65% of the total inefficiencies of the program; we therefore focus on investigating the reasons for this in more detail. In this application, we focus on weighting stall cycles with floating point operations, to show the impact of inefficiency in this respect on the entire application. (For other kinds of applications, an integer-based metric might be more appropriate.) We multiply the difference by the total number of floating point operations in order to filter out the regions with few operations. One thing to note is that inefficiencies (in stall cycles) from MPI calls do not appear in this graph. We proceed to investigate the MPI behavior by looking at the analyses offered by KOJAK. Its MPI results indicate that less than 3% of the application time is spent in MPI communication. This validates the results from Fig. 10, where no MPI routines appeared in the list of inefficient procedures.

Using the information derived from the 2nd and 3rd steps, we see that the inefficiencies are mostly related to the stalls in the floating point unit for both *irrad* and *solir*. *Solir* has more stalls due to L1D cache misses. We need to understand whether the stalls in the floating point unit were related to accesses to L2 data or dependences in the pipeline. In the Itanium processor, the floating point data is fetched directly from L2 cache.

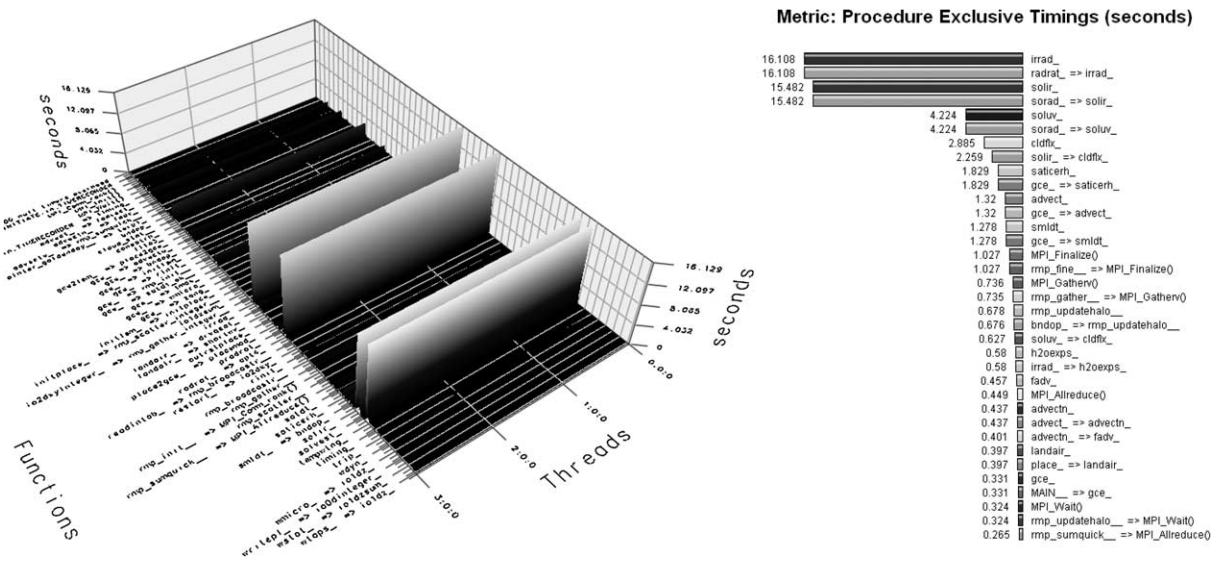

Fig. 10. Profiling and time spend in the cloud resolving application.

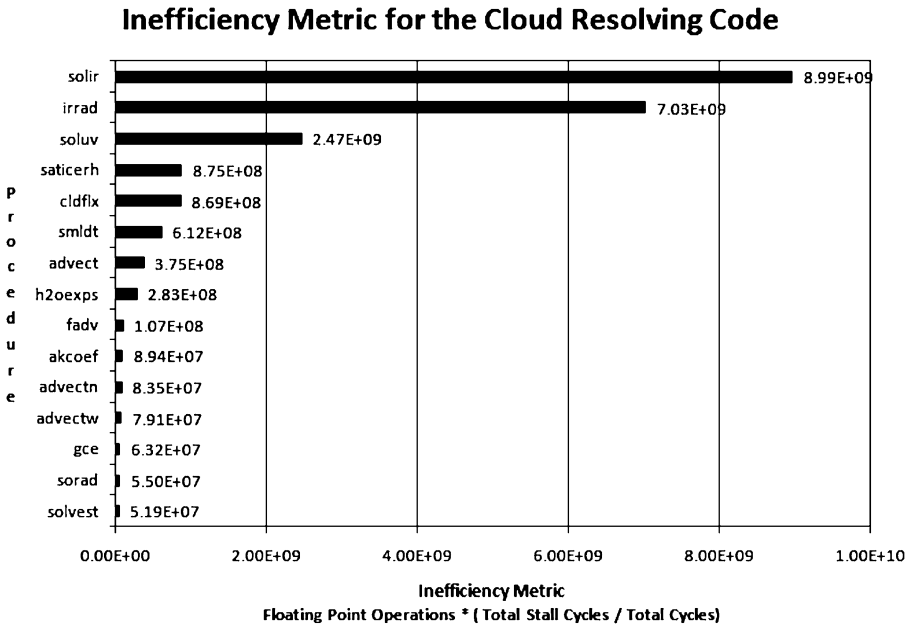

Fig. 11. Inefficiency metric for the cloud resolving application.

Figure 12 gives details on the stalls. It is clear from this that the inefficiencies in *irrad* are mainly due to floating point unit stalls. The inefficiencies in *solir* are due to both floating point unit stalls and L1 data cache stalls. Memory analysis results shown in Fig. 13 indicate that both routines have good memory access patterns, in particular that there is very little access to local and remote memories. We also notice that the L2 cache is heavily used: its bandwidth to the processor is possibly being saturated. In the case of solir 10% of its memory accesses are cycles spent in L3 cache. Overall, the application significantly stresses the L2 data cache. The possible stress in the L2 cache is due in part to L1 misses; it may also arise from the floating point unit that obtains data directly from L2 and the prefetcher that loads data to the L2 cache.

The compiler optimization logs does not report that the loop has been optimized: this has been prevented

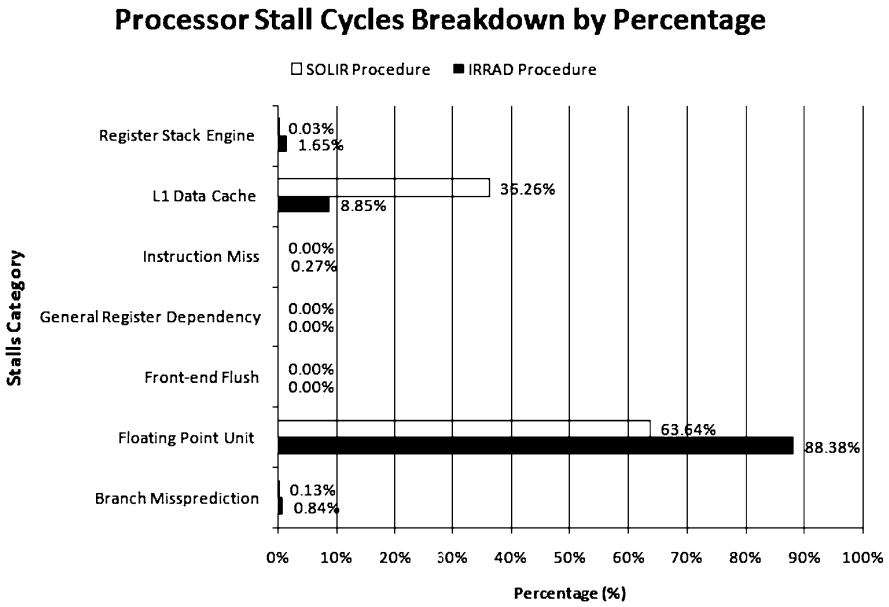

Fig. 12. Processor stalls in the cloud resolving code.

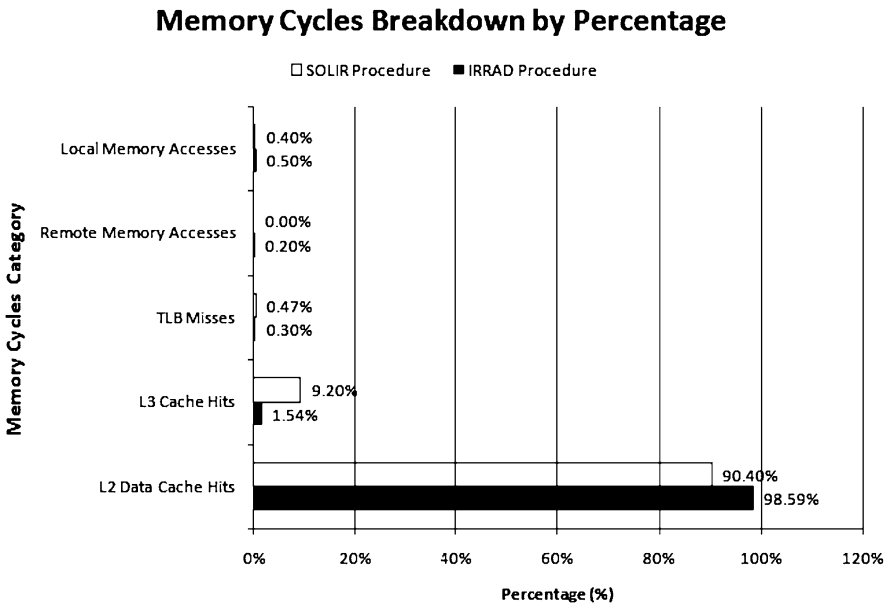

Fig. 13. Memory cycles spent in the cloud resolving code.

by the branch condition inside the loop nest. The frequency of the conditionals indicates that the branches are evenly executed. However, we note there is an imbalance in the amount of work performed in the body of the conditionals; the *then* branch executes many more statements than the *else* branch. We also note that the compiler attempts to prefetch the data for the loops based on branch prediction. The data prefetching not only brings unnecessary data into cache, it also leads to the eviction of data that is in use and saturates the bandwidth. (We confirmed that this was the case in a subsequent step of our methodology.) To avoid this, we try to precompute the values of the conditional in order to enable a better prefetching approach. Also, by removing the conditional from the computational loop, the compiler can better optimize the inner loops.

Figure 14 shows pseudo code for these loops before and after the manual change to ensure the code precomputes the value of the conditional. After this minor change to the loops involved, we observe up to 26% improvement in the overall performance of the entire application. Figure 15 shows timings for the cloud formation code when run with varying numbers of processors. We see that the optimization scales well.

Understanding how the optimization worked is an important task. We use the compiler translation and the performance algebra to determine this. We collected performance information in the optimized version and converted all the TAU profiles to CUBE profiles. When comparing the original code with the optimized version we see that the main reason for the improvements is because it eliminated the problems with the cache bottlenecks. When comparing the original loops with the optimized version using CUBE's performance algebra we see that the optimization eliminated 51% of the cache stalls due to the cache being full. It reduced 12% of the instruction prefeteches, 13% reduction on data prefeteches and 8% of reduction in the cache stalls. These improvements were reflected in reductions to the values in the hardware counters pertaining to each region. What this infrastructure allowed us to do was to explain to us how effective the optimization to address the performance problem and its side effects.

```
DO I=0 N
  X1 = ....
  WE = ...+ (X1)
  IF (WE.gt.W1-2)
   THEN
     ... (several statements)
     TRANS(I) =
   ELSE
     TRANS(I) = 0.9999
ENDDO
```

Original

```
DO I=0 N
  X1(I) = ..
  WE(I)=..
  RES(I) = 0
 IF (WE.gt.W1-1)
  RES(I) = 1
ENDDO

DO I=0 N
   .... (several statements)
  TRANS(I) = RES(I)*TRANS(I)
           + (1-RES(I))*0.99999
ENDDO
```

Optimized

Fig. 14. Code transformation/optimizations.

6. Conclusions and future work

In this thesis we have presented a methodology for solving performance problems that exploits the capabilities of an integrated tuning environment created in a collaboration between open source compiler developers and performance tools providers. We have moreover described a selective instrumentation algorithm that can be implemented in a compiler by adapting a typical strategy for performing inlining analysis. In

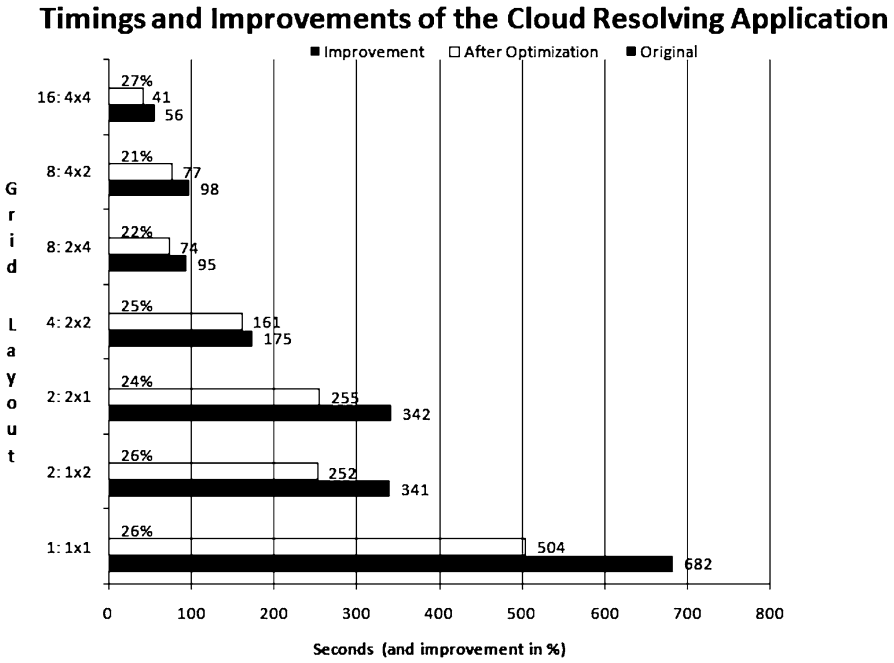

Fig. 15. Timings in seconds for the cloud resolving application for different MPI processor layouts before and after the optimizations. The improvements are shown in percentage over the original time.

the examples presented here, selective instrumentation based upon this algorithm was able to reduce profiling overheads by 90 times on average in the NAS OpenMP parallel benchmarks and by 17 times for the cloud formation code.

We also have also introduced in some detail a methodology for using this environment to carry out performance bottleneck analysis. Our methodology enabled us to detect and deal with the performance problems of the cloud resolving. Our methodology combined monitoring, profiling and tracing to detect the application bottlenecks. We also used a set of derived metrics to perform processor and memory bottleneck analysis with TAU and used KOJAK tracing patterns to detect MPI performance problems.

Our future work will combine feedback-directed optimizations to improve selective instrumentation and a set of optimizations that the compiler can use to help resolve the bottlenecks. We will also continue to explore opportunities to enhance the working of performance tools via direct compiler support.

Acknowledgments

This work was supported by the National Science Foundation, under contracts CCF-0444468 and CCF-0702775. We would like to thank our partners in the Copper Project: Felix Wolf, Shirley Moore, Rick Kufrin and Danesh Tafti for providing valuable feedback on how to improve and design the environment. We would also like to thank Bob Hood and Davin Chang from CSC at Nasa Ames for providing us access to their Altix systems to perform the experiments reported on here.

References

[1] D. Becker, W. Frings and F. Wolf, Performance evaluation and optimization of parallel grid computing applications, in: *PDP'08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, Washington, DC, USA, IEEE Computer Society, 2008, pp. 193–199.

[2] B. Buck and J.K. Hollingsworth, An API for runtime code patching, *The International Journal of High Performance Computing Applications* **14**(4) (2000), 317–329.

[3] V. Bui, O. Hernandez, B. Chapman, R. Kufrin, D. Tafti and P. Gopalkrishnan, Towards an implementation of the openmp collector api, in: *PARCO*, Jülich, Germany, 2007.

[4] O. Hernandez, F. Song, B. Chapman, J. Dongarra, B. Mohr, S. Moore and F. Wolf, Instrumentation and compiler optimizations for mpi/openmp applications, in: *International Workshop on OpenMP (IWOMP 2006)*, Reims, France, 2006.

[5] S. Hiranandani, K. Kennedy and C.-W. Tseng, Compiler support for machine-independent parallel programming in Fortran D, Technical Report CRPC-TR91132, Rice University Houston, TX, USA, 1991.

[6] H.-M.H. Juang, W.-K. Tao, X. Zeng, C.-L. Shie, S. Lang and J. Simpson, Parallelization of the NASA Goddard Cumulus Ensemble model for massively parallel computing, *Journal of Terrestrial Atmospheric and Oceanic Sciences* **18** (2007), 593–622.

[7] S. Kortmann, I. Park, M. Voss and R. Eigenmann, Interactive and modular optimization with interpol, in: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, USA, 2000.

[8] R. Kufrin, Measuring and improving application performance with perfsuite, *Linux J. 2005* **135** (2005), 4.

[9] R. Kufrin, PerfSuite: An accessible, open source, performance analysis environment for Linux, in: *6th International Conference on Linux Clusters (LCI-2005)*, Chapel Hill, NC, USA, April 2005.

[10] C. Liao, O. Hernandez, B. Chapman, W. Chen and W. Zheng, OpenUH: An optimizing, portable OpenMP compiler, in: *12th Workshop on Compilers for Parallel Computers*, Coruña, Spain, 2006.

[11] S.-W. Liao, A. Diwan Jr., R.P.B., A.M. Ghuloum and M.S. Lam, SUIF explorer: An interactive and interprocedural parallelizer, in: *Principles Practice of Parallel Programming*, Atlanta, GA, USA, 1999, pp. 37–48.

[12] K.A. Lindlan, J.E. Cuny, A.D. Malony, S. Shende, B. Mohr, R. Rivenburgh and C. Rasmussen, A tool framework for static and dynamic analysis of object-oriented software with templates, in: *Supercomputing*, Dallas, TX, USA, 2000.

[13] M. Gerndt and E. Kereku, Selective instrumentation and monitoring, in: *International Workshop on Compilers for Parallel Computers (CPC 04)*, Chiemsee, Germany, 2004.

[14] A.D. Malony, S. Shende, R. Bell, K. Li, L. Li and N. Trebon, Advances in the tau performance system, *Performance Analysis and Grid Computing* (2004), 129–144.

[15] C. McNairy and R. Bhatia, Montecito: A dual-core, dual-thread itanium processor, *IEEE Micro* **25**(2) (2005), 10–20.

[16] B. Mohr and F. Wolf, KOJAK – a tool set for automatic performance analysis of parallel applications, in: *Proc. of the European Conference on Parallel Computing (EuroPar)*, Klagenfurt, Austria, 2003, pp. 1301–1304.

[17] K. Mohror and K.L. Karavanic, A study of tracing overhead on a high-performance linux cluster, in: *PPoPP'07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, ACM Press, 2007, pp. 158–159.

[18] C.M. Pancake, Applying human factors to the design of performance tools, in: *Euro-Par'99: Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, London, UK, Springer-Verlag, 1999, pp. 44–60.

[19] I. Park, M. Voss, B. Armstrong and R. Eigenmann, Parallel programming and performance evaluation with the URSA tool family, *International Journal of Parallel Programming* **26**(5) (1998), 541–561.

[20] P. Petersen and S. Shah, Openmp support in the intel thread checker, in: *WOMPAT*, Toronto, ON, Canada, 2003, pp. 1–12.

[21] C.A.F.D. Rose and H.-U. Heiss, Dynamic processor allocation in large mesh-connected multicomputers, *Lecture Notes in Computer Science* **2150** (2001), 783.

[22] L.D. Rose, Y. Zhang and D.A. Reed, SvPablo: A multi-language performance analysis system, *Lecture Notes in Computer Science* **1469** (1998), 352–355.

[23] C. Roth, F. Levine and E. Welbon, Performance monitoring on the powerpc 604 microprocessor, in: *ICCD'95*, Washington, DC, USA, 1995, p. 212.

[24] M. Schulz, J. Galarowicz and W. Hachfeld, Open – speedshop: open source performance analysis for linux clusters, in: *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, ACM Press, 2006, p. 14.

[25] M.L.V.D. Vanter, D.E. Post and M.E. Zosel, Hpc needs a tool strategy, in: *SE-HPCS'05: Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, New York, NY, USA, ACM, 2005, pp. 55–59.

[26] G.R. Watson and N.A. DeBardeleben, Developing scientific applications using eclipse, *Computing in Science and Eng.* **8**(4) (2006), 50–61.

[27] F. Wolf and B. Mohr, Automatic performance analysis of hybrid mpi/openmp applications, *J. Syst. Archit.* **49** (10/11) (2003), 421–439.

[28] J.H. Wolf, Programming methods for the Pentium III processor's streaming SIMD extensions using the VTune performance enhancement environment, *Intel Technology Journal* **3**(Q2) (1999), 11.

[29] P.-T. Wu and P. Narayan, Multithreaded performance analysis with sun workshop thread event analyzer, in: *SPDT'98: Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, New York, NY, USA, ACM Press, 1998, p. 161.