A Holistic Approach towards Automated Performance Analysis and Tuning*

Guogjing Cong, I-Hsin Chung, Huifang Wen, David Klepacki, Hiroki Murata, Yasushi Negishi, and Takao Moriyama

IBM Research

Abstract. High productivity to the end user is critical in harnessing the power of high performance computing systems to solve science and engineering problems. It is a challenge to bridge the gap between the hardware complexity and the software limitations. Despite significant progress in language, compiler, and performance tools, tuning an application remains largely a manual task, and is done mostly by experts. In this paper we propose a holistic approach towards automated performance analysis and tuning that we expect to greatly improve the productivity of performance debugging. Our approach seeks to build a framework that facilitates the combination of expert knowledge, compiler techniques, and performance research for performance diagnosis and solution discovery. With our framework, once a diagnosis and tuning strategy has been developed, it can be stored in an open and extensible database and thus be reused in the future. We demonstrate the effectiveness of our approach through the automated performance analysis and tuning of two scientific applications. We show that the tuning process is highly automated, and the performance improvement is significant.

1 Introduction

Developments in high performance computing (HPC) have primarily been driven so far by bigger numbers of floating-point operations per second (FLOPS). Complex HPC systems pose a major challenge to end users to effectively harness the computing capability. It can take tremendous efforts of a user to develop an application and map it onto the current supercomputers. The US department of advanced research projects agency (DARPA) sponsored the high productivity computing system initiative [7] to develop technologies that help bridge the productivity gap. Improving productivity and maintaining high performance involve multiple areas of computer science research. Despite significant progress, deploying an application to complex architectures for high performance remains a highly manual process, and demands expertise possessed by few engineers.

A typical tuning life cycle is as follows. When the performance of an application is below expectation, a user is faced with the task of observing the behavior,

* This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 33–44, 2009.

formulating hypothesis, and conducting validation tests. First the application is instrumented for performance data collection. Crude hypothesis can be formed based on the data, and observing for minute details (also called tracing) to refine or validate follows. Assuming trace data (oftentimes they are of a huge amount) can be efficiently manipulated, a user has to correlate the runtime behavior with the program characteristics. If a mismatch between the application and the architecture/system is detected, the user usually needs to trace back to the source program to find where the mismatch is introduced. Familiarity with the internals of the compiler is a must since a program goes through transformations by a compiler. The mismatch can be introduced by the compiler (not implementing the best transformations due to its constraints), or by the source program (e.g., implementing a poorly performing algorithm). Once the cause for the performance problem is identified, the user chooses the appropriate ones from her repertoire of optimizations for implementation. In doing so she needs to make sure the transformation does not violate program constraints, and conserve other desired properties such as portability.

Thus performance diagnosis requires in-depth knowledge of algorithm, architecture, compiler, and runtime behavior, and involves collecting, filtering, searching, and interpreting performance data. Tuning further requires coordinating the intricate interactions between multiple components of a complex system. Performance analysis and tuning remain challenging and time consuming even for experienced users. In our study we improve the tuning productivity by providing software services that help the automation of the process. Automated performance optimization has been studied at limited scales (as opposed to whole application or system scale) under different contexts. Here we give brief summary of related work.

An optimizing compiler is one of the most important auto-tuners. Profile-guided compilers (e.g., see [3]) find even further optimization opportunities in a program by utilizing both static and runtime information. Powerful as it is, a compiler does not tune well programs that heavily utilize specialized *primitives* that are largely outside a compiler's control. Some examples of these primitives include MPI communication, I/O operation, and pre-developed library routines. A compiler is usually not able to conduct transformations on the algorithms. It also lacks domain knowledge that is crucial to tunings dependent on the inputs. Even within standard loop transformations, the compile time constraint oftentimes does not allow a compiler to search for the best parameters or combinations.

Auto-tuning libraries (e.g., see [16,14]) are able to tune themselves (e.g., search for appropriate parameter values) for a given architecture configuration. As the tuning is for a predefined set of programs (usually much smaller than a regular application) with relatively limited transformations, accurate modeling and intelligent searching are the main effective techniques. These techniques alone in general will not tune an application.

Performance tools (e.g., see [11,10,12,13]) traditionally facilitate performance data collection and presentation. They aim to provide clues to trained experts

about possible performance problems. They in general do not explicitly point out the problem itself, and require a lot of efforts to digest the performance data.

In this paper we present our approach towards automated performance analysis and tuning. We work with experts to observe the way they tune their applications, and attempt to automate the procedure by mining their knowledge. We developed a framework that actively searches for known performance patterns instead of passively recording all the tracing information. Such a framework is designed to be open and extensible to accommodate new performance patterns. We also provide a mechanism to collect, compare, and correlate performance data from different aspects of the system for performance diagnosis. The framework also attempts to mitigate performance problems by suggesting and implementing solutions. Our approach attempts to unify performance tools, compiler, and expert knowledge for automated performance tuning.

The remainder of this paper is organized as follows: Section 2 describes our strategy for the development of automatic performance optimization tools. Section 3 describes our approach for incorporating multiple performance tools and compiler analysis in performance problem diagnosis. Section 4 presents automated solution proposal. Section 5 presents our case study. We present concluding remarks and future work in Section 6.

2 Performance Optimization Strategy

Working with performance experts, we propose a holistic approach towards performance analysis and tuning. We provide assistance to the user throughout the tuning process, that is, our framework provides services for performance data collection, bottleneck identification, solution discovery and implementation, and iteration of the tuning process. The framework strives to give definite diagnosis to a problem so that the user is no longer left with a sea of perplexing data. Performance tools, compiler, and expert knowledge are key components of our system. Integrating these components improves the tuning efficiency by providing means to compare and correlate performance data from different dimensions, and to propose and implement solutions.

Our work focuses on three levels of support to boost productivity. The first is to provide easy access to a wide array of information from static analysis, runtime behavior, algorithm property, architecture feature, and expert domain knowledge. Based upon such information, a mechanism to compare and correlate performance metrics from different aspects (e.g., computation, memory, communication, I/O) is developed, and helps accurately pinpoint the cause of performance problems. The ultimate goal is to automate tuning as an expert would. It involves proposing solutions and implementing them. Granted that currently (even in the near future) we do not foresee our framework to be able to generate knowledge itself, it can be instructed to follow certain tuning routines of an expert. Mining as much knowledge as possible from the experts liberate them from the repetitive tasks and is expected to greatly improve the productivity of regular users.

Our methodology can be summarized as follows. We collect the cause of performance problems from literature and performance experts, and store them as patterns defined on performance metrics. Our framework inspects and instruments the application, and actively searches for known patterns in the pattern database. Once a pattern is discovered, we claim that the corresponding bottleneck is found, and the framework consults the knowledge database for possible solutions. Solutions are evaluated, and implemented if desired by the user. There are two aspects of our work. One is infrastructure support. That is, we develop the necessary facilities for the automation process. The other is to populate the framework with a rich collection of bottleneck and solution definitions. We realize the sheer number of performance problems can easily overwhelm any study group. We make the framework open and extensible, and distill common utilities that help expert users expand the databases.

3 Bottleneck Discovery

A bottleneck is the part of a system that limits the performance. Achieving maximum performance can probably be formulated as a gigantic combinatorial optimization problem. Yet the sheer complexity determines that oftentimes we have to resort to heuristics for sub-optimal solutions. In practice, bottleneck discovery requires knowledge about application domain, compiler, software system, and architecture, and is traditionally done by a small group of experts. A mechanism to mine the expert knowledge is necessary to automate the tuning process.

The mining act is not trivial as the wisdom is often expressed in fuzzy terms. Formalizing the knowledge takes effort as shown by the following example. MPI derived data types have been proposed to improve the flexibility and performance of transferring non-contiguous data between processors. Not all programmers are fluent with derived data types, and they still compact data explicitly into a contiguous buffer before communication. This practice results in additional copying overhead, and can seriously degrade performance. The tuning guideline from the expert is clear. Simply stated, it asks to get rid of the redundant data packing by using derived data types. Yet it is not straightforward for a machine to detect the packing behavior, especially when the packing is done in a different function than where the communication occurs. Moreover, the packing itself can be complicated, and involves several loop regions. To identify the buffer being sent is simple (trivially through MPI tracing), but confirming the fact that data was being packed is hard. We formalized a scheme that relies on runtime memory access analysis (intercepting loads/stores to the buffer at run time), and flow analysis (through static analysis) to discover the behavior. This example emphasizes the need to integrate tools and compiler for bottleneck detection.

A bottleneck in our framework is a rule (pattern) defined on a set of metrics. The rule is expressed in a mini-language, and most current rules are logic

expressions. The design goal of the language is to be flexible and expressive enough to cover most frequent problem patterns. As more knowledge about bottlenecks is developed, new language features might be added. Rule definitions are acquired from literature and expert knowledge. The rule database is designed to be open for expansion and customization.

In our design a performance metric is any quantifiable aspect about or related to application performance. Examples include the number of pipeline stalls for a given loop, the number of prefetchable streams, the number of packets sent from a certain processor, the size of physical memory, and whether loops have been tiled by the compiler. The bottleneck rule provides a way to compare and correlate metrics from multiple sources and dimensions, and helps the accurate diagnosis of the problem. Having a large collection of metrics helps introducing new rules.

3.1 Metrics from Existing Performance Tools

Each existing performance tool is specialized in collecting certain performance metrics. These metrics can be used in building meaningful bottleneck rules through a mechanism our framework provides for metric import.

In our current implementation, the framework is able to collect many metrics through the IBM high performance computing toolkit (IHPCT) [15]. IHPCT contains a profiling tool [2], a hardware performance monitor (HPM), a simulation guided memory analyzer (SiGMA) [5], an MPI profiling and tracing tool, an OpenMP tracing tool, and a modular I/O tool. Each of these components evaluates and/or measures certain performance aspect of the application, and the wealth of performance data collected serve as metrics for bottleneck definitions. For example, HPM alone collects up to hundreds of metrics about various hardware events.

Metrics from different tools can co-exist, and we also experiment with collecting metrics through TAU [10] and Scalasca [6]. Table 1 shows some sample metrics collected by existing tools.

Combining the analysis of multiple tools can be simply achieved by defining rules that use metrics collected by them. For example, the following rule points to a potential pipeline stalling problem caused by costly divide operations in a loop.

Table 1. Example metrics collected by different performance analysis programs

metric name	description	collected by
PM_INST_CMPL	instruction completed	HPM
L1_miss_rate	L1 miss rate	HPM
Avg_msg_size	average massage size	MPI profiler
Thread_imbalance	thread work load imbalance	Open MP profiler
#prefetches	number of prefetched cache lines	SiGMA
mpi_latesender	Time a receiving process is waiting for a message	Scalasca

$$\#divides > 0 \;\&\&\; \frac{PM\_STALL\_FPU}{PM\_RUN\_CYC} > t \;\&\&\; vectorized = 0$$

Here, *#divides* is the number of divide operations in the loop (collected by some static analysis module or even keywords from the UNIX *grep* utility), while PM_STALL_FPU and PM_RUN_CYC are two metrics collected by HPM that measure the number of cycles spent on stalls due to floating point unit and the total number of cycles, respectively. In this rule t is a constant threshold.

3.2 Metrics from the Compiler

To be able to accurately pinpoint a performance problem, static analysis is oftentimes necessary to understand the structure of the program. It is desirable to bring compiler analysis into the bottleneck discovery. Most compilers are usually not concerned with providing services to external tools, and the complexity of the compiler daunts attempts from outside to utilize its analysis other than compiling a program. Under the DARPA HPCS initiative, currently we are working with the IBM compiler group to expose standard compiler analysis to the tools and users.

Performance metrics provided by the compiler such as estimate of number of prefetchable streams, estimate of pipeline stalls, and number of basic blocks are useful in constructing bottleneck rules. More importantly, since a large class of performance problems are related to optimizations that are not carried out by the compiler (although in theory it is capable of), it is immensely helpful to get a report from the compiler on optimizations performed on the code. Without knowing what optimizations have been applied to the hotspots, it is virtually impossible to provide meaningful diagnostics, especially for problems in the computation domain, as the target binary code becomes a black box to us. Take the *unroll* analysis in our study as an example. Loop *unroll and jam* is a well studied compiler technique. Indeed many compilers claim to have competent unrolling transformations. In practice, however, we found from performance engineers that in many cases even industrial strength compilers do not do the best job in unrolling outer loops. The reasons could be that either only in very high optimization level is outer-loop unrolling triggered (which can be a problem for programs requiring strict semantics), or the compiler decides that other forms of transformation are appropriate which preclude unrolling. As we conduct postmortem analysis of an application and direct tuning efforts to a few code regions, we can afford more detailed analysis for better estimating the costs and benefits of an unrolling vector. Experimental results show that our unroll analysis produces faster code than some industrial-strength compiler [4]. In this case, performance bottleneck is the existence of discrepancy between parameters estimated by our module and those proposed by the compiler. A compiler can also report reasons that a potential performance-boosting optimization is not done. Such information provides insight that help further tune an application.

For bottleneck discovery, we utilize analysis results from the compiler are stored in an XML database. Using this database, metrics and transformations on a certain code region can be retrieved.

4 Solution Composition and Implementation

Our framework attempts to propose measures (which we call solutions) to mitigate the performance bottlenecks. Candidate solutions mined from expert knowledge are stored in the solution database. Solutions are in generic forms, and need to be instantiated. For example, a generic solution for the bottleneck where excessive time is spent on blocking MPI calls is to overlap computation with communication, while whether and how the overlap can be done are application dependent. Instantiating a solution involves the following actions. First legality check is necessary to preserve data dependency. Second, the parameter values are computed. In overlapping communication and computation for MPI programs, non-blocking calls and their locations are the parameters. Next, performance improvement is estimated through modeling or running the application patched with the solution. Lastly, code modifications and environment settings are determined.

Parameter values largely determine the effectiveness of the solutions. One important aspect of solution determination is to find the optimal parameter values. Such analysis for CPU related problems is similar to that done by an optimizing compiler. As time constraint on the tuning process of our framework is in general not as stringent as that on that compiler, oftentimes we can search for the optimal parameter values with the help of a performance estimation engine. For example, for loop unroll as a solution, we generate pseudo instructions for different unroll vectors and schedule the instructions to run on an universal machine model [4]. Performance estimation engine counts the number of cycles spent on the normalized loop body, and the search engine chooses the parameter with the best execution cycles for the optimal solution.

The effectiveness of solution discovery is closely related to the accuracy of bottleneck detection. The more detailed a bottleneck description is, the easier for the system to propose solutions. Consider the following scenario. Suppose the framework detects a bottleneck pattern where excessive pipeline stalls occurred in a code region. It is hard to propose any meaningful solution without further information as possible causes are numerous. If it is also detected that the majority of the stalls are due to data cache misses, the hint is to improve data locality. If static analysis reveals that irregular accesses occurred to specific arrays, solution discovery can focus on those arrays. As our framework is involved with every aspect of the tuning process, the quality of solution can be improved by accurate bottleneck discovery.

We implement three kinds of solutions: standard transformations through compilers, modifications to the source code, and suggestions. Compiler support obviates the need to modify the source code for standard transformations. Our framework focuses on searching for better parameter values, and delegates the actual transformation to the compiler. We work with compiler researchers to develop two different facilities for implementing solutions. One is through compiler directives, and the other is through the polyhedral scripting framework [1].

Directives serve as suggestions to the compiler, and the polyhedral framework provides a flexible interface for the tool to implement its own desired optimization composed from standard transformations. Solution in the form of source code modification is necessary when there is no compiler support for the transformations, for example, optimizations to MPI communications. In section 5 we present examples that employ compiler directives and source code modifications to implement solutions.

5 Case Study

Currently our framework contains many built-in metrics and rules for bottleneck detection. Most notably all metrics collected by hardware event counters are present, together with many metrics collected by static analysis and compiler analysis. On the solution side, the framework is able to automatically tune for several performance problems. Here we give an example of using our framework to analyze and tune an application. Note that expanding the framework to automatically tune the two applications provide utilities that are reusable in future tunings.

The application we consider is Lattice Boltzmann Magneto-Hydrodynamics code (LBMHD) [9]. The Lattice Boltzmann method is a mesoscopic description of the transport properties of physical systems using linearized Boltzmann equations.

Hotspot detection shows that two functions, *stream* and *collision*, take most of the execution time. And it is clear that there are many pipeline stalls, and resources in the system are not fully utilized. Tuning experts point to a performance problem that is due to two different access orderings on the same multi-dimension array in *stream* and *collision*.

Figure 1 shows the loops of interest in subroutine *collision*. For multi-dimensional arrays f, g, feq, and geq, the access order incurred by the j, i, k iteration order does not match with their storage order, and creates massive cache misses (consider the k dimension, for example). There are two ways to match the array access order and the storage order. The first is to change the access order by loop-interchange. In *collision*, however, the loops are not perfectly nested. It is impossible to implement loop interchange without violating the dependency constraints. The remaining option is to change the storage order to match the access order by re-laying out the array. Changing the storage order does not affect the correctness of the program, and is a viable solution. Of course, this optimization affects the performance of all accesses to the affected arrays, a fact that needs consideration for solution implementation.

For arrays f and feq, as the control variables are in the k, i, j order counting from the inner most, the new layout is to store the array such that the k dimension is stored first, followed by the i dimension, then the j dimension. In other words, the new storage order is (3,1,2), the 3^{rd} dimension first, then the 1^{st} dimension, followed by the 2^{nd} dimension. For arrays g and geq, all accesses have the forth subscript as constant. The new storage order should store the 4^{th} dimension first. For the k, i, j control variable access order, the storage order is

```
do j = jsta, jend
   do i = ista, iend
   ...
      do k = 1, 4
         vt1 = vt1 + c(k,1)*f(i,j,k) + c(k+4,1)*f(i,j,k+4)
         vt2 = vt2 + c(k,2)*f(i,j,k) + c(k+4,2)*f(i,j,k+4)
         Bt1 = Bt1 + g(i,j,k,1) + g(i,j,k+4,1)
         Bt2 = Bt2 + g(i,j,k,2) + g(i,j,k+4,2)
      enddo
      ...
      do k = 1, 8
         ...
         feq(i,j,k)=vfac*f(i,j,k)+vtauinv*(temp1+trho*.25*vdotc+ &
             .5*(trho*vdotc**2- Bdotc**2))
         geq(i,j,k,1)= Bfac*g(i,j,k,1)+ Btauinv*.125*(theta*Bt1+ &
             2.0*Bt1*vdotc- 2.0*vt1*Bdotc)
         ...
     enddo
     ...
   enddo
enddo
```

Fig. 1. Code excerpt of collision

similar to that of f, and feq. The final storage order is (4,3,1,2), that is, the 4^{th} dimension first, the $3rd$ dimension second, followed by the 1^{st} and 2^{nd} dimensions. To implement the new storage order, we resort to compiler directives. For example, on IBM platforms, the XL compiler provides the *!IBM SUBSCRIPTORDER* directive that accepts a new storage order. For LBMHD, four arrays have their storage orders changed through the following directive.

$$!IBM\ SUBSCRIPTORDER(f(3,1,2), feq(3,1,2), g(4,3,1,2), geq(4,3,1,2))$$

Changing the storage order of the arrays may create undesirable side effects for other program constructs. In LBMHD, arrays f, feq, g, and geq are shared by several routines. In our earlier analysis, we found proper storage orders for these arrays in *collision*. However, these orders introduce new problems for *stream* (Figure 2).

The access order in *stream* to arrays g and geq matches exactly with the original storage order. Yet to improve the cache performance for *collision*, the storage orders are changed. To remedy, we can try changing the access order of *stream* to match the new storage order. Changing access order is constraint by data dependency. Fortunately for *stream*, loop-interchange can be applied as follows. The whole nested loop is distributed into two perfectly nested loops. Each in turn is interchanged. Then the two perfectly nested loops are fused together. As a result, the outer most loop (do k = 1, 2) is moved to the inner most, the inner most loops (do i = ista, iend) to the middle, and the loop (do j = jsta, jend) is moved to the outer most.

```
do k = 1, 2
   do j = jsta, jend
      do i = ista, iend
         g(i,j,1,k)= geq(i-1,j,1,k)
         ....
       enddo
      enddo
   do j = jsta, jend
      do i = ista, iend
         g(i,j,2,k)= w1*geq(i,j,2,k)+ w2*geq(i-1,j-1,2,k)
                     + w3*geq(i-2,j-2,2,k)
         g(i,j,4,k)= w1*geq(i,j,4,k)+ w2*geq(i+1,j-1,4,k)
                     + w3*geq(i+1,j-2,4,k)
         ...
      enddo
   enddo
 enddo
```

Fig. 2. Code excerpt from *stream*

Using our bottleneck description language, the following rule is inserted to the bottleneck database for detecting the mismatch between iteration order and storage order.

$$STALL\_LSU/PM\_CYC > \alpha \textbf{ and } STRIDE1\_RATE \leq \beta$$
$$\textbf{and } REGULAR\_RATE(n) > STRIDE1\_RATE + \gamma$$

STALL_LSU and PM_CYC are metrics that measure the number of cycles spent on LSU stalls and the total number of cycles, respectively. STRIDE1_RATE estimates the number of memory accesses that are stride-1. REGULAR_RATE estimates the number accesses that have regular stride. What this rule says is that if there is a significant number of cycles spent on LSU unit, and there are more n-stride accesses than stride-1 access, there is potentially a bottleneck that may be removed by array transpose and related optimizations. In the rule, α, β, and γ are constants used as thresholds. Different threshold values signify the different levels of seriousness of the performance problem. The threshold values may also be tied to the performance gain a user expects from removing the bottleneck. Static analysis and runtime hardware events collection modules are responsible for the collection of these metrics.

There is only one solution associated with this bottleneck, as the bottleneck definition narrows the range of possible solutions. Discovering solution parameters and implementing the solution are still manual at this stage. Yet after the development of these modules, utilities such as finding the array access order and the storage order become available to further users, and can simplify their development of similar modules.

Once the bottleneck rule and solution discussed above are introduced, our framework can automate the performance tuning for LBMHD. Similar

performance problems can be detected and mitigated for other applications. Due to limited space, we refer interested readers to IBM *alphaworks* [8] for examples and instructions of using the framework.

In our experiment, the *gtranspose* solution achieved about 20% improvement in execution time with a grid size 2048×2048 and 50 iterations on a P575+ (1.9 GHz Power5+, 16 CPUs. Memory: 64GB, DDR2) on one processor. Note that without loop-interchange for *stream*, the transformation actually degrades the overall performance by over 5% even though the performance of *collision* is improved. Loop-interchange mitigates the adverse impact of the new storage order on *stream*. There is still performance degradation of *stream* after Loop-interchange. The degradation is due to the new memory access pattern. After Loop-interchange, the k-loop becomes the inner-most loop. Although the memory accesses to g and geq are consecutive, the corresponding array dimensions are of a small range (k goes from 1 to 2). And the next dimension introduces strided access. In the original code, both the i-loop and j-loop have sequential accesses to g and geq spanning a region of size $(iend - ista) \times (jend - jsta)$.

6 Conclusion and Future Work

In this paper we presented our study of unifying performance tools, compiler, and expert knowledge for high productivity performance tuning. Our framework facilitates a holistic approach that detects bottlenecks and proposes solutions. Performance data collected by existing performance tools can be used as metrics. The analysis of multiple tools can be correlated and combined through bottleneck rules. Compiler analysis and optimization play a critical role in our framework. We demonstrated the advantages of performance optimization through our framework through tuning two applications. The tuning process is highly automated, and the performance improvement is significant in both cases.

As we observed, the effectiveness of the framework depends on the number and quality of bottleneck rules and solutions in our database. In future work, we plan to populate the database with more rules and solutions. We also expect to improve the services and utilities the framework provides for expansion and customization.

References

1. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: Proc. 13th international conference on parallel architecture and compilation techniques, Antibes Juan-les-Pins, France, September 2004, pp. 7–16 (2004)
2. Bhatele, A., Cong, G.: A selective profiling tool: towards automatic performance tuning. In: Proc. 3rd Workshop on System Management Techniques, Processes and Services (SMTPS 2007), Long beach, California (March 2007)
3. Chen, W., Bringmann, R., Mahlke, S., et al.: Using profile information to assist advanced compiler optimization and scheduling. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1992. LNCS, vol. 757, pp. 31–48. Springer, Heidelberg (1993)

4. Cong, G., Seelam, S., et al.: Towards next-generation performance optimization tools: A case study. In: Proc. 1st Workshop on Tools Infrastructures and Methodologies for the Evaluation of Research Systems, Austin, TX (March 2007)
5. DeRose, L., Ekanadham, K., Hollingsworth, J.K., Sbaraglia, S.: Sigma: a simulator infrastructure to guide memory analysis. In: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, pp. 1–13 (2002)
6. Geimer, M., Wolf, F., Wylie, B.J.N., Abraham, E., Becker, D., Mohr, B.: The SCALASCA performance toolset architecture. In: Proc. Int'l Workshop on Scalable Tools for High-End Computing (STHEC), Kos, Greece (2008)
7. High productivity computer systems (2005), `http://highproductivity.org`
8. High productivity computing systems toolkit. IBM alphaworks, `http://www.alphaworks.ibm.com/tech/hpcst`
9. MacNab, A., Vahala, G., Pavlo, P., Vahala, L., Soe, M.: Lattice Boltzmann Model for Dissipative Incompressible MHD. In: 28th EPS Conference on Contr. Fusion and Plasma Phys., vol. 25A, pp. 853–856 (2001)
10. Malony, A.D., Shende, S., Bell, R., Li, K., Li, L., Trebon, N.: Advances in the tau performance system, pp. 129–144 (2004)
11. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The Paradyn Parallel Performance Measurement Tool. IEEE Computer 28, 37–46 (1995)
12. Mohr, B., Wolf, F.: KOJAK – A tool set for automatic performance analysis of parallel programs. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 1301–1304. Springer, Heidelberg (2003)
13. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: A tool to visualise and analyze parallel code. In: Proc of WoTUG-18: Transputer and occam Developments, vol. 44, pp. 17–31. IOS Press, Amsterdam (1995)
14. Vuduc, R., Demmel, J., Yelick, K.: OSKI: A library of automatically tuned sparse matrix kernels. In: Proceedings of SciDAC 2005, Journal of Physics: Conference Series (2005)
15. Wen, H., Sbaraglia, S., Seelam, S., Chung, I., Cong, G., Klepacki, D.: A productivity centered tools framework for application performance tuning. In: QEST 2007: Proc. of the Fourth International Conference on the Quantitative Evaluation of Systems (QEST 2007), Washington, DC, USA, 2007, pp. 273–274. IEEE Computer Society, Los Alamitos (2007)
16. Whaley, R., Dongarra, J.: Automatically tuned linear algebra software (ATLAS). In: Proc. Supercomputing 1998, Orlando, FL (November 1998), `www.netlib.org/utk/people/JackDongarra/PAPERS/atlas-sc98.ps`