Capturing Performance Knowledge for Automated Analysis

Kevin A. Huck*, Oscar Hernandez†, Van Bui†, Sunita Chandrasekaran‡,
Barbara Chapman†, Allen D. Malony*, Lois Curfman McInnes§ and Boyana Norris§
*Computer and Information Science Department, University of Oregon
Eugene, OR 97403-1202 Email: khuck,malony@cs.uoregon.edu
†Department of Computer Science, University of Houston
Houston, TX 77204-3010 Email: oscar, chapman@cs.uh.edu, van.bui@mail.uh.edu
‡Centre for High Performance Embedded Systems, Nanyang Technological University
Singapore, 637553 Email: suni0003@ntu.edu.sg
§Mathematics and Computer Science Division, Argonne National Laboratory
Argonne, IL 60439 Email: curfman,norris@mcs.anl.gov

***Abstract*—Automating the process of parallel performance experimentation, analysis, and problem diagnosis can enhance environments for performance-directed application development, compilation, and execution. This is especially true when parametric studies, modeling, and optimization strategies require large amounts of data to be collected and processed for knowledge synthesis and reuse. This paper describes the integration of the PerfExplorer performance data mining framework with the OpenUH compiler infrastructure. OpenUH provides auto-instrumentation of source code for performance experimentation and PerfExplorer provides automated and reusable analysis of the performance data through a scripting interface. More importantly, PerfExplorer inference rules have been developed to recognize and diagnose performance characteristics important for optimization strategies and modeling. Three case studies are presented which show our success with automation in OpenMP and MPI code tuning, parametric characterization, and power modeling. The paper discusses how the integration supports performance knowledge engineering across applications and feedback-based compiler optimization in general.**

I. Introduction

Accurate parallel performance analysis is a complicated and intimidating task for even an experienced performance analyst. On the one hand, the management of multi-experiment performance data from parametric studies and the application of multi-step processes involving various statistical, data mining, and meta-analysis operations can introduce errors if done manually. On the other, lack of support for analysis automation translates ultimately to the loss of knowledge, earned through experience, about successful performance engineering practices – what analysis methods are useful for what performance problems, how performance models are obtained and validated, and how to interpret performance results relative to opportunities for optimization. Advancement in parallel performance problem solving and its integration in optimization frameworks will depend on creating analysis workflows and capturing expert rules for automated use.

Effective performance analysis automation requires performance tools which support data management, process scripting, and knowledge engineering, as well as their integration with parallel program development environments. While the "heavy lifting" analysis will occur in the performance system, it is the ability to encode general expertise and case-specific methods that allows development tools to direct analysis strategies to problem solving needs. There are two key research challenges. The first is in designing flexible analysis components and usable interfaces for their integration. The second is in engaging the parallel programming and tuning environments to use the knowledge-based analysis automation capabilities.

This paper describes the integration of the PerfExplorer [9] performance data mining system with the OpenUH [13] compiler infrastructure. The union provides OpenUH auto-instrumentation of source code for performance experimentation and automated analysis of the performance results using PerfExplorer scripts. More importantly, PerfExplorer inference rules are developed to recognize and diagnose performance characteristics important for OpenUH modeling and optimization strategies.

To demonstrate the benefits of automating analysis methods, two case studies are presented on OpenMP and MPI code tuning, one targeting load balance problems (a multiple sequence alignment application) and another targeting data locality problems (a fluid dynamics application). In both, the goal is to capture the optimization process and insight gained by the manual tuning of these applications in the form of PerfExplorer analysis scripts and inference rules. The approach is validated by comparing the optimized code to the unoptimized. A third case study is reported for the fluid dynamics application, but in the context of power modeling. Here PerfExplorer scripts and inference rules demonstrate how optimizing various functions affects the power consumption in the hardware.

This paper is organized as follows. Section I gives an introduction to our automation framework. Section II provides a brief introduction to the tools and gives a description on their integration in this project. Section III illustrates the example problems and corresponding graphs and results. Related work

```
# create a rulebase for processing
ruleHarness = RuleHarness.useGlobalRules(
  "openuh/OpenUHRules.drl")
# load a trial
trial = TrialMeanResult(Utilities.getTrial(
  "Fluid Dynamic", "rib 45", "1_8"))
# calculate the derived metric
stalls = "BACK_END_BUBBLE_ALL"
cycles = "CPU_CYCLES"
operator = DeriveMetricOperation(trial, stalls,
  cycles, DeriveMetricOperation.DIVIDE)
derived = operator.processData().get(0)
# compare values to average for application
for event in derived.getEvents():
  MeanEventFact.compareEventToMain(derived,
    mainEvent, derived, event)
# process the rules
ruleHarness.processRules()
```

Fig. 1. Sample Jython script.

```
rule "Stalls per Cycle"
  when f : MeanEventFact (
    m : metric == "(BACK_END_BUBBLE_ALL /
         CPU_CYCLES)",
    h : higherLower == MeanEventFact.HIGHER,
    s : severity > 0.10, e : eventName,
    a : mainValue, v : eventValue,
    factType == "Compared to Main" )
  then
    System.out.println("Event " + e + " has
      a higher than average stall / cycle rate");
    System.out.println("\tAverage stall /
      cycle: " + a);
    System.out.println("\tEvent   stall /
      cycle: " + v);
    System.out.println("\tPercentage of total
      runtime: " + s);
end
```

Fig. 2. Sample JBoss Rules rule.

is discussed in Section IV. The paper concludes with a discussion on how the integration supports performance knowledge engineering across applications and feedback-based compiler optimization in general.

II. Design

In this section, we briefly overview the design of PerfExplorer 2.0 and the OpenUH compiler infrastructure, and describe our approach to their integration.

A. PerfExplorer 2.0

Instrumentation and measurement tools such as TAU [18] can collect very detailed performance data from parallel applications. The potential sizes of datasets and the need to assimilate results from multiple experiments makes it a challenge to both process the information and discover and understand new insights about performance. In order to perform analysis on collections of TAU performance experiment data, we developed PerfExplorer, a framework for parallel performance data mining and knowledge discovery. The framework architecture enables the development and integration of data mining operations that can be applied to parallel performance profiles. PerfExplorer is built on PerfDMF, a data management framework which provides a library to access the parallel profiles and save analysis results in a relational database. PerfDMF includes support for nearly a dozen performance profile formats, including TAU. PerfExplorer is integrated with existing data mining toolkits, and allows for extensions using those toolkits.

In the latest release of PerfExplorer, we have added two new features which will aid in automated analysis and are relevant to this paper. First, we have added a scripting interface for process control. The scripting interface is in Jython, which is a full Python interpreter written in Java. Because PerfExplorer is a Java application, all of the application objects are available to the script interface, but we limit the access to a smaller subset API. With the interface, it is straightforward to derive new metrics, perform analysis, and automate the processing of performance data. An example script is shown in Figure 1. This simple example loads some inference rules, loads a trial from PerfDMF, derives an inefficiency metric, and then compares each event's exclusive value with the inclusive value of main before processing the rules, where an event is defined as any instrumented code region.

The second relevant new feature in PerfExplorer is the integration of the JBoss Rules inference engine for rule processing. The rules which interpret the performance results are easily constructed and modified, and an expert system for explaining parallel performance data can be constructed. PerfDMF and PerfExplorer have been extended for better support of performance context, or metadata, and rules can be constructed which include the metadata to justify conclusions about the performance data. An example rule is shown in Figure 2. This example rule will fire for any and all events which have a higher than average stall per cycle rate, and also account for at least 10% of the total run time.

B. OpenUH Compiler

The OpenUH [13] 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 provides complete support for OpenMP 2.5 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 (WOPT), and code generator (CG). Each of these modules supports frequency-based feedback directed optimizations. OpenUH has five levels of a tree-based intermediate representation (IR) called WHIRL to facilitate the implementation of different analysis and optimization phases. Most compiler optimizations are implemented on a specific level of WHIRL. OpenUH has been enhanced to support the requirements of TAU, Kojak and PerfSuite by supporting an instrumentation API for source code and OpenMP runtime library support.

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 a complete compile-time instrumentation module that works at different compilation phases and covers a variety of program constructs (e.g. procedures, loops, branches, callsites). We have designed a language-independent compiler instrumentation API that can be used to instrument complete applications written in C, C++, Fortran, OpenMP and MPI [8]. MPI operations are instrumented via PMPI rather than by the compiler. OpenMP constructs are handled via runtime library instrumentation, where the fork-join events and implicit and explicit barriers are captured [2]. All these types of instrumentation are related to each other. For example, procedure and control flow instrumentation are essential for relating the MPI and OpenMP-related output to the execution path of the application, or for understanding how constructs behave inside these regions.

The instrumentation module can be controlled via compiler flags, specifying the types of regions we want to instrument. It is invoked at different phases during compilation to provide feedback to IPA, LNO, WOPT, the OpenMP translation, or CG. The compiler instrumentation retains a mapping identifier that can be used to relate performance data back to the intermediate representation at a given optimization phase. The compiler currently supports feedback for branch, loop, and control flow optimizations, and callsite counts to improve inlining. All these optimizations are frequency-based and this work is being done as an initial step towards providing feedback to the internal cost-models of the compiler.

Some compiler optimization modules compute a cost model to guide the optimization strategies. For example, the loopnest optimizer has an explicit processor model, a cache model and a parallel overhead model. OpenUH static cost modeling [29] evaluates different combinations of loop optimizations, using constraints to avoid an exhaustive search. The processor model includes instruction scheduling and register pressure, and is based on the processor's computational resources, latencies and registers. The cache model helps to predict cache misses and the cycles required to start up inner loops. The parallel model was designed to support automatic parallelization by evaluating the cost involved in parallelizing a loop, and to decide which loop level to parallelize. The parallel model accounts for threaded fork-join and reduction overhead.

The cost model can be customized for specific optimization goals. Currently, it can focus on reducing cache misses, register pressure, instruction scheduling, pipeline stalls and parallel overheads.

C. Tool Integration

The ultimate goal with this tool integration is to improve the performance of applications compiled with OpenUH, using feedback suggestions from PerfExplorer to improve cost model optimizations and OpenMP parameterization. Figure 3 shows what this integration would look like in a completed form, and how the tools inter-operate currently. Source code is compiled with OpenUH, which also does instrumentation and code generation. The instrumented application is executed, and TAU profiles are stored in a PerfDMF repository. This data is analyzed with PerfExplorer, and the diagnoses and explanations are passed on to the user as performance suggestions. In the future, we hope to integrate the tools with a feedback optimization loop to improve the compiler cost models, but currently we require manual changes to the source code.

Previously, feedback optimizations have been used to improve runtime behavior for control - improving branches, frequently executed control flow paths, and loop optimizations based on counting the number of times a path or a loop gets executed. By using feedback suggestions from PerfExplorer, we believe we can improve the performance of the application by providing runtime analysis data (with hardware counter information) to the cost model estimation, which is currently constrained to using only static analysis data. By improving the cost models we can guide the compilation process to prefer a transformation that reduces power consumption, or which reduces cache misses, or improves computational density.

III. Results

Our case studies were conducted on the Altix 300 and Altix 3600. We collected the performance characteristics in the Altix 300 and perform production runs in the Altix 3600 with higher number of processors. The Altix 300 is a distributed-shared memory system consisting of 8 nodes with two Itanium 2 processors each. The Altix 3600 consists of 256 nodes, with a total of 512 processors. 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 NUMAlink. 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. 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.

A. Multiple Sequence Alignment

Molecular biologists frequently compute multiple sequence alignment (MSA) to compare protein sequences with unknown functionality to a set of known sequences to detect functional similarities[31]. The steady growth in the size of sequence databases means that the comparisons require increasingly significant scan times. Because the time and space complexities for MSA are in the order of the product of the lengths of the sequences, many heuristic alignment methods have been developed. Among them, progressive alignment is a widely used heuristic. The popular MSA program ClustalW[21] is one such example. It consists of three stages: distance matrix, guided tree, and progressive alignment along the tree. The main purpose of MSA is to infer homology between sequences. Profiling of the ClustalW program on a single processor showed that almost 90% of the time is spent in

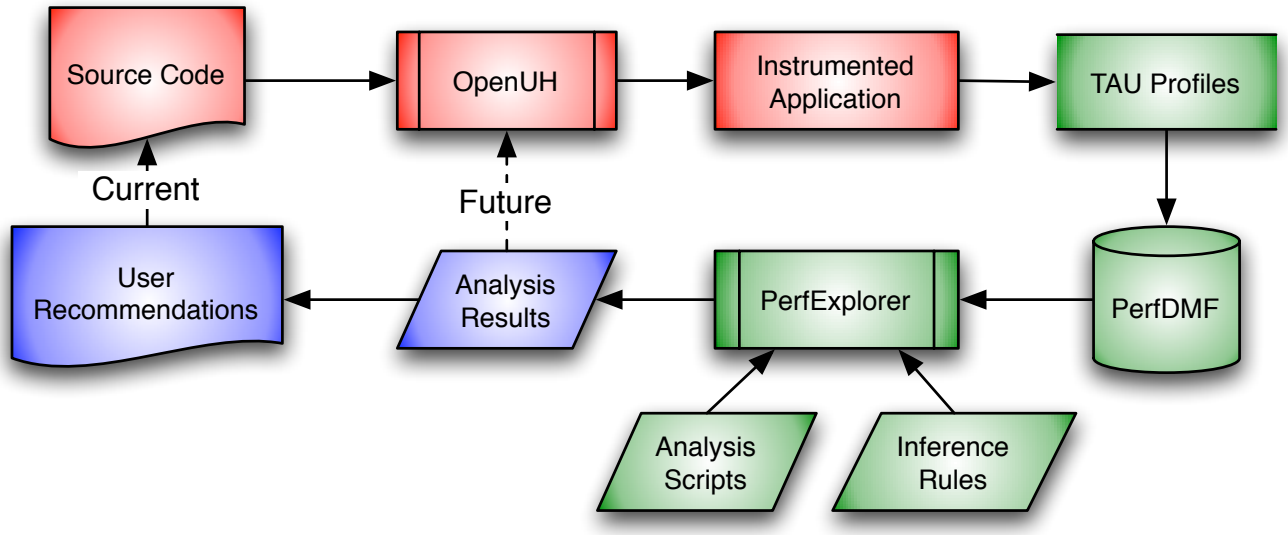


Fig. 3. PerfExplorer integrated with OpenUH. Future capabilities will bypass the need for manual changes to the source code by the user.

the first stage. (i.e. computing the distance matrix of the three stages). This ClustalW first stage is based on the Smith-Waterman algorithm, a dynamic programming approach that computes the optimal local alignment of two sequences. Full details of the MSA stages and the SW algorithm can be obtained from [21] and elsewhere. We parallelized the SW algorithm using OpenMP for the main computational loops but did not get a solution that scaled for large numbers of threads.

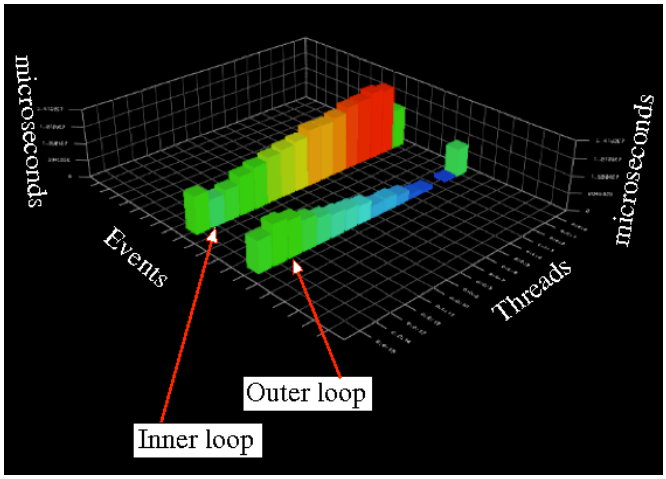
To improve OpenMP performance, we used schedule clauses to specify how the iterations of the main loop should be allocated to the threads. Among the different scheduling mechanisms, we applied static and dynamic scheduling on different protein sequences and varied dynamic chunk sizes to drill down to a suitable scheduling strategy that scaled. In the process, we found that *static even* (the default), and *dynamic even* scheduling experienced load imbalances. Uneven tasks were distributed to the processing units, as shown in Figure 4(a). As for the chunk sizes scheduled for each thread, we found that the imbalance was due to uneven distribution of work. Because the parallel loops in the algorithm were at a very coarse grained level containing four nested loops, small chunk sizes gave the best speedup. Larger chunk sizes tend to change the scheduling behavior to be more like the static even behavior. When applying the right dynamic scheduling, the load imbalances were reduced and scaling efficiency was increased up to 80% with 128 threads on a 1000 sequence set when using a chunk size of one. Figure 4(b) shows the scaling behavior of different schedules on up to 16 threads.

In order to capture this analysis with PerfExplorer, we developed a script which performed a load balancing test of the code. For each instrumented region, or event, we computed the mean and standard deviation of time across all threads, and then computed the ratio of the standard deviation to the mean. Because the outer loop was waiting for the inner loop to complete, we also wanted to detect that for a parent-child relationship in the callgraph, an increase in the time spent in the inner loop meant a shorter time spent in the outer loop. That was done by correlating the times spent in the loops and getting a high negative correlation. We also wanted to compute the amount of useful work done in the outer loop, which would indicate if threads were working or were waiting at the barrier for other threads to finish.

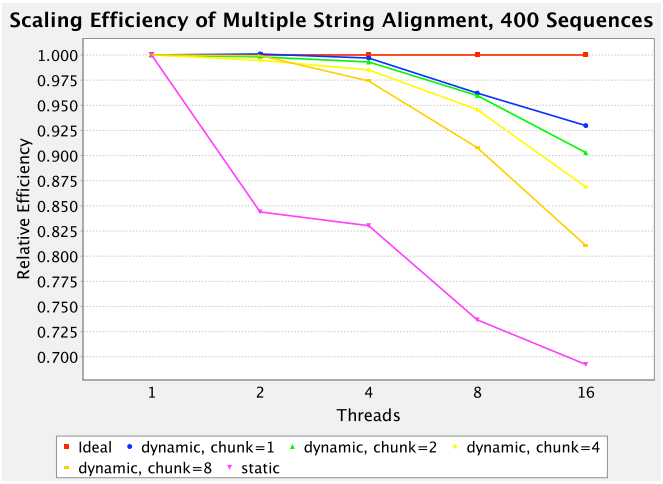
The load imbalance detection rule is activated when the following facts are true. First, two loops have a high standard deviation to mean ratio (> 0.25), which indicates that they are unbalanced across the threads. Second, the loops occupy more than 5% of the total runtime, which indicates the severity that this load imbalance has on the runtime. Third, the events are nested - that is, one of the events calls the other in the call graph. Fourth, on a per-thread basis, the times in the events are highly negatively correlated - that is, a thread that finishes the inner loop early will spend more time in the outer loop waiting at the barrier, whereas a thread which spends more time in the inner loop will spend less time in the barrier. When all these facts are asserted true, the rule will fire and the user will be indicated of the problem, and the suggested scheduling change.

B. GenIDLEST

Generalized Incompressible Direct and Large-Eddy Simulations of Turbulence (GenIDLEST)[20] solves the incompressible Navier-Stokes and energy equations and is a comprehensive and powerful simulation code with two-phase dispersed flow modeling capability, turbulence modeling capabilities, and boundary conditions to make it applicable to a wide range



(a) Load imbalance in inner and outer loops, 16 threads.



(b) Relative Efficiency of MSAP Application. A dynamic schedule with a chunk size of 1 is nearly 93% efficient using 16 processors.

Fig. 4. MSAP scaling behavior for a 400 sequence problem set.

of real world problems. It uses an overlapping multi-block body-fitted structured mesh topology in each block combining it with an unstructured inter-block topology. The multiblock approach provides a basic framework for parallelization, which can be exploited by SPMD parallelism using MPI, OpenMP or a hybrid. In a representative problem with n computational blocks, it can use up to n MPI processors or equivalently n OpenMP threads or various combinations of MPI-OpenMP without loss of generality. Further, within each block, "virtual cache blocks" are used. The "virtual" blocks are not explicitly reflected in the data structure but are used in two-level Additive or Multiplicative Schwarz preconditioners for solving linear systems. In addition to the favorable preconditioning properties, the small "cache" blocks also allow efficient use of cache on hierarchical memory systems in modern chip architectures[27]. The virtual cache blocks also provide an additional level of parallelism.

Two test cases which investigate the internal cooling of turbine blades are presented here: a fully-developed flow in a 45-degree ribbed internal cooling duct using Detached Eddy-Simulations (45rib); and another case with the same geometry but with a 90 degree rib and using the method of Large-Eddy Simulations (90rib). The former has a grid consisting of 128x80x64 decomposed into 8 blocks of 128x80x8 and the latter has a grid of 128x128x128 decomposed into 32 blocks of 128x128x4. The two cases are executed using both MPI and OpenMP on up to 8 and up to 32 processors of the SGI Altix, for the 45 and 90 degree problems, respectively.

In the code framework each computational block has ghost cells at inter-block boundaries and also at periodic boundaries which are used in the flow direction. Ghost cell updates on each processor employ asynchronous MPI communications and involve two additional temporary buffers that enable some overlapping of the `MPI_Isend` and `MPI_Ireceive` operations for greater efficiency.

In the 45rib and 90rib examples executing on 8 and 16 MPI processors respectively, 2 `MPI_Isend` and `MPI_Ireceive` calls are invoked on each processor with 2 on-processor copies, noting that these are done in parallel across MPI processes. However, when using standalone OpenMP (with 8 threads for 45rib and 16 or 32 threads for 90rib), all boundary updates are copies in shared memory initiated by the master thread. Hence there are 30 on-processor copies for 45rib and 126 on-processor copies for 90rib, all initiated by the master thread.

Our methodology is part of an application tuning cycle that consists of iterative application runs that enable scalable instrumentation and feedback optimizations, as follows:

Profiling with Selective Instrumentation: Our selective instrumentation method [7] is designed to create a scoring mechanism for regions of interest based on their importance in the code and call graph. We want to avoid instrumenting regions of code that have small weights (e.g. few basic blocks, statements) and are invoked many times. In this run we focus on procedure level instrumentation. The goal of the initial run is to determine where the processor 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. The formula to calculate the inefficiency for this purpose is:

$$Inefficiency = Floating\_Point\_Operations * (Total\_Stall\_Cycles/Total\_Cycles)$$

This formula is calculated using PerfExplorer for each region being measured. The regions with the highest inefficiency are the regions that the programmer and compiler should focus on optimizing.

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 includes instrumenting loops, branches,

calls, and possibly individual statements. During this run we collect hardware counters to perform the processor bottleneck analysis. The general formula we have adopted for this purpose is the following based on Jarp [10]:

$$Total\_Stall\_Cycles = L1D\_Cache\_Misses + Branch\_Misprediction + Instruction\_Misses + StackEngine\_stalls + Floating\_Point\_Stalls + Pipeline\_Inter\_Register\_Dependencies + Processor\_Frontend\_Flushes$$

We primarily collect performance data for stall cycles from the L1D 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 these two causes, 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 general guideline based on behavior seen in different applications.

Memory Analysis Metrics: In the same way as the second run, we use hardware counters to perform the memory bottleneck analysis based on the following formula:

$$Memory\_Stalls = (L2\_data\_references\_L2\_all - L2\_misses) * L2\_Memory\_Latency + (L2\_misses - L3\_missed) * L3\_Memory\_Latency + (L3\_misses - Number\_of\_remote\_memory\_accesses) * Local\_Memory\_Latency + (Number\_of\_remote\_memory\_accesses) * Remote\_memory\_access\_latency + TLB\_misses * TLB\_miss\_penalty$$

$$Remote\_Memory\_Accesses\_Ratio = Number\_of\_remote\_memory\_accesses / L3\_misses$$

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. The value for remote memory latency accesses is an estimation of the worst-case scenario for a pair of nodes with the maximum number of hops and is system dependent.

In this case study we wanted to understand why the OpenMP implementation of this application does not scale when compared to the MPI implementation in the SGI Altix. The OpenMP version lagged by a factor of 11.16 behind its MPI counterpart for the case of 90rib and 3.48 for the 45rib case. The unoptimized OpenMP version of the application does not scale at all as seen in Figure 5(b).

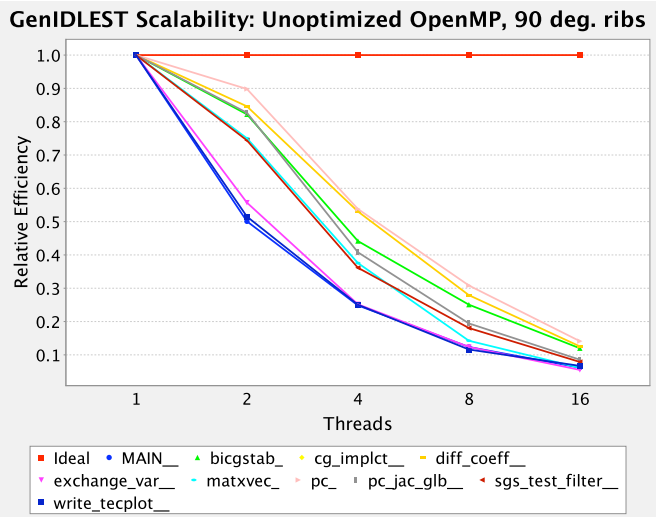
We constructed PerfExplorer scripts to derive the metrics, and created rules to examine the results. For the first metric, we constructed a script which loaded the data, derived the inefficiency metric, and then a rule searched for events with high inefficiency. We used the script and accompanying rules to examine a 16-thread run of the OpenMP implementation on the 90rib problem, six procedures with poor scaling were identified with a higher than average stall-per-cycle rate. We constructed a second script which derived the total stall metric. The rule for the second metric was to look for events which had 90% or more of their stalls caused by floating-point stalls or memory stalls. The same six events, plus two more, were identified as having a high percentage of stalls from those two sources. We constructed a third script to examine the causes of the memory stalls. The script was primarily concerned with the numbers of L3 cache misses and the ratio of local memory references to remote memory references.

The performance slowdown is mostly caused by a data locality difference between the two versions. This was indicated by higher number of L3 cache misses and latencies in the OpenMP version, as opposed to the MPI version. Figure 5(a) shows that the main computation procedures `bicgstab`, `diff_coeff`, `matxvec`, `pc`, `pc_jac_glb` (among others) do not scale. Data locality is important for achieving good performance in the SGI Altix. SGI Altix provides the default *first-touch* policy for placing data, in which a page of memory is allocated/moved to the local memory of the first process to access the page. The use of a default first-touch policy has worked very well on a single threaded or MPI processes code on many NUMA platforms, but may lead to poor performance with OpenMP. In MPI all the memory accesses are to local memory by default. OpenMP has the flexibility to use the first-touch policy to place data in the different nodes since the data are not explicitly mapped to processors as with MPI. In addition, OpenMP has a privatization feature where data can be defined as local to each thread.

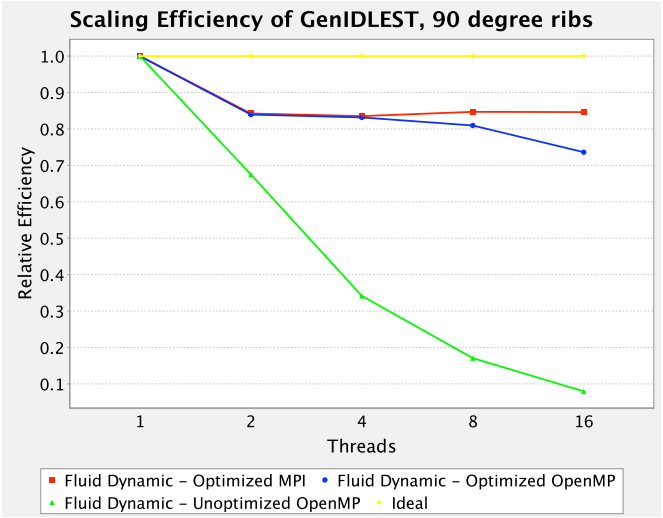
The final major source of performance degradation is caused by the procedure `exchange_var__` as seen in Figure 5(a). This procedure is responsible for driving the exchange of data in the ghost cells. Because one of its subroutines (`mpi_send_recv_ko`) is sequential, it limits the scalability of the application. In the old implementation of the boundary update procedure, which was primarily written for the MPI paradigm, the on-processor copies were done sequentially since most of the work was distributed over MPI processes. However, this became a major bottleneck in the OpenMP paradigm. Four of the events from the previous script were identified as having a lower ratio of local to remote memory references than the application on average. One of these events, `exchange_var__`, represented 31% of the runtime, and was scaling very poorly, which confirms its sequential nature and its local data.

Since PerfExplorer was able to determine that the main problem in the computational procedures were L3 misses and remote memory accesses (when compared to MPI) we discovered that the application was initializing most of its data sequentially, resulting in data being placed on one node. We fixed all the initializations by parallelizing the initialization loops to make sure we place the data correctly across processors.

To remedy the `exchange_var__` problem the on-processor copies were parallelized by eliminating two intermediate steps in the update procedure: that of filling the intermediate send buffer with data to be copied and copying this buffer into an intermediate receive buffer, which are inherently



(a) Speedup per event, unoptimized OpenMP.



(b) Speedup of optimized and unoptimized OpenMP, and optimized MPI.

Fig. 5. GenIDLEST scaling behavior for 90rib problem.

serial operations. In the optimized version, an OpenMP do parallel loop is applied to the blocks residing on the processor (8 for 45rib and 32 for 90rib) and direct copies are initiated from the send buffer to the destination array.

After optimization, both the MPI and OpenMP baseline performance improved, and the OpenMP implementation scaled nearly as well as MPI, as seen in 5(b). The performance difference between the MPI and OpenMP implementations become minimal, in the range of 15% for 90rib and 16.8% for 45rib which is a big improvement from the unoptimized version. The lesson learned here is that we need to provide feedback to the compiler to tell it that it should focus on improving the L3 optimizations by targeting reduction of the cycles predicted in the cache model. We must also feed back information to the inter-procedural array region analyzer to make sure that all the data are initialized and accessed consistently across procedures to improve data locality via the first-touch policy. The feedback presented to the user includes suggestions for the `exchange_var__` procedure.

C. Power Modeling with GenIDLEST

With rising energy costs for managing and running supercomputing systems, there is an increasing need for more integrated tool infrastructures that support both performance *and power* monitoring and analysis capabilities. In addition to the robust, performance-centric, automated analysis capabilities of PerfExplorer shown thus far, we investigate how PerfExplorer may be applied for power analysis.

In our study of modeling processor power consumption and energy efficiency, we use PerfExplorer to compute a power metric based on [23]. The power metric is based on hardware performance counters and on the on-die components of the processor (see Equation 1 and Equation 2). In Eq. 1, power is computed for each component(C_i) of the processor. The maximum power value is the published thermal design power (TDP) for the processor. The power for each component is weighted based on the access rates for each component. Eq. 2 computes the total power consumed by the processor and is based on the sum of the power consumed by *n* components and the idle power. For multiprocessor or multicore systems, the total power across all processing elements can be modeled by summing the total power computed in Eq. 2 for each processor or core.

$$Power(C_i) = AccessRate(C_i) * ArchitecturalScaling(C_i) * MaxPower \quad (1)$$

$$TotalPower = \sum_{i=0}^{n} Power(C_i) + IdlePower \quad (2)$$

We implemented PerfExplorer scripts to obtain power dissipation and energy consumption estimates and analysis. We used GenIDLEST running the 90rib dataset as a power/energy case study. Different levels of standard optimizations for the OpenUH compiler were applied ranging from O0 (all optimizations are disabled) to O3 (applies the most aggressive optimizations including loop nest optimizations). The application was run in parallel with MPI on 16 processors on the Altix 300.

The results from the case study show that power dissipation generally increases with higher optimization levels while energy decreases as more aggressive compiler optimizations are applied (see Table I). These results are consistent with previous studies that examine the effects of compiler optimizations on power and energy efficiency [22], [17]. Also consistent with a previous research study [22], we find that the instruction count is directly proportional to energy consumption and a similar relationship exists between instructions per cycle (IPC) and power dissipation. A higher instruction count translates to more work for the CPU and so energy increases. Optimizations

such as common subexpression elimination and copy propagation that decrease the number of instructions are generally beneficial when compiling for energy efficiency. In the case of compiling for power efficiency, optimizations that increase the overlap in instruction execution while keeping the instruction count fairly constant (and therefore increasing IPC) results in higher power consumption. Examples of optimizations that may increase power dissipation include software pipelining, instruction scheduling, and vectorization.

Compilers apply different sets of standard optimizations at each level. For a given study based on our power model, results will differ dependent on the compiler. The results here are specific for the OpenUH compiler or a compiler that applies similar optimizations at each level. Table I shows that at optimization O1, we get an increase in power, as well as a decrease in energy. At O1, minimal optimizations such as instruction scheduling and peephole optimizations are applied to straight-line code. These optimizations will expectedly have an effect on both power and energy. At O2, the more aggressive optimizations significantly decrease the total instruction count (e.g. dead store elimination and partial redundancy elimination) and so we get a significant decrease in energy consumption and a small drop in power dissipation. At the most aggressive level of optimization (O3), loop nest optimizations (such as vectorization and loop fusion/fission) are enabled leading to increases in instruction execution overlap and therefore increases in power dissipation. Given the results from this case study, PerfExplorer might be able to direct either the compiler or programmer to optimize for low power, low energy, or both using inference rules. The results from Table I suggest that O0 should be enabled for low power, O3 enabled for low energy, and O2 for both power and energy efficiency for the OpenUH compiler. Compiling for low energy can be important for embedded and scientific applications, whereas compiling for low power has more significant long-term effects in terms of system reliability and reduced cooling and operational costs for large-scale servers.

Metric	O0	O1	O2	O3
Time	1.0	0.338	0.071	0.049
Instructions Completed	1.0	0.471	0.059	0.056
Instructions Issued	1.0	0.472	0.063	0.061
Instructions Completed Per Cycle	1.0	1.397	0.857	1.209
Instructions Issued Per Cycle	1.0	1.400	0.909	1.316
Watts	1.0	1.025	1.001	1.029
Joules	1.0	0.346	0.071	0.050
FLOP/Joule	1.0	2.867	13.684	19.305

TABLE I
GENIDLEST RELATIVE DIFFERENCES FOR DIFFERENT OPTIMIZATION SETTINGS, USING 16 MPI PROCESSES ON A 90RIBLET PROBLEM. OPTIMIZATION LEVEL O0 IS THE BASELINE.

IV. RELATED WORK

Feedback optimizations include a variety of techniques that aim to improve the execution behavior of a program based on information on its current or previous runtime behavior. Runtime information, which may be specific to a given instance of a program's execution, helps the compiler direct its efforts to frequently executed regions of code and make better judgments on what set of optimizations can improve the code. There is a large body of work, including our own, that focuses on offline optimizations. Systems such as GEM, IMPACT, SUIF, OpenUH, DCPI, FX!32 Morph, GCC, Alpha Compaq Compilers, SGI compilers, and PROMISE perform high-level and object-level optimizations. Typical optimizations include feedback-directed inlining, partial dead code elimination, instruction scheduling, code reordering and loop optimizations. Several sets of runtime information based on training sets of input data may be used to characterize the typical runtime behavior of the application. Other systems focus on online code re-optimizations via software with the help of hardware. Dynamo, Cursoe, IA32EL, PIN, re-optimize object code, ADAPT [26], Tempo, DyC, and 'C all create specialized versions of the code during runtime. Little work has been devoted to a dynamic compilation system that works for OpenMP. Studies have shown that is feasible to optimize codes with performance information for new multicore architectures. stOMP [3] has been proposed to target OpenMP. stOMP focuses on *value phasing*, optimizing code in a parallel region based on the current values of shared values. Recent work experimented with different OpenMP scheduler configurations at the parallel region and loop level in OpenMP codes [30]. Optimizing at the loop level resulted in better performance, but led to very high runtime overheads mostly from the decision algorithm applied to selecting the OpenMP work scheduling algorithm for a loop. None of these approaches use a combination of performance analysis and modeling to provide feedback to the compiler. Marathe [15] presented a tool for profile-guided automatic page placement for ccNUMAs. The approach is low level, compiler independent but dependent on input data provided to the application.

The use of performance problem solving in automated analysis depends on having rich tools for exploring the relationship between performance and computation behavior. The FINESSE [16] tool demonstrates the use of overhead analysis to explain experimental observations based on models of execution behavior. The benefit was prescriptive in that it provides a basis for successive refinement in program development to better performing solutions. FINESSE targets shared-memory optimization of a molecular dynamics application. KappaPi (Knowledge-based Automatic Parallel Program Analyzer for Performance Improvement) [11] and KappaPi2 are tools for detecting known performance bottlenecks in PVM and MPI applications. The tools determine causes by applying inference rules to the analysis of trace files. The causes are then related back to the source code and include recommendations to the user.

Our work on Poirot [14] considered general support for automating performance diagnosis in parallel tools, and the Hercule[12] tool showed how performance diagnosis can be built on computational model-centric rules for finding symptoms of and explanations for common performance problems

in applications, such as *load imbalance*, *insufficient parallelization*, and *scheduling overhead*. Since performance problems are diagnosed in the context of the application's parallel model, possible solutions for correcting the inefficiencies can be proposed. In contrast to our current work, Hercule analyzes event trace files, not profiles, and lacks support for analysis scripting.

EXPERT[19] is an automatic event-trace analysis tool for MPI and OpenMP applications. It searches the traces for execution patterns indicating low performance and quantifies them according to their severity. The patterns target both problems resulting from inefficient communication and synchronization as well as from low CPU and memory performance. SCALASCA[6] parallelizes the EXPERT trace analysis methods and provides the CUBE[28] graphical viewer for highlighting performance problems in relation to threads of execution and metrics. CUBE implements Performance Algebra, a technique for performing difference, merge and aggregation operations on parallel performance profile data. In contrast to these tools, PerfExplorer provides support for translating performance problems and metrics into performance knowledge and rules, and for integrating these performance analytics capabilities into automated analysis environments.

Performance Assertions[25] have been developed to confirm that the empirical performance data of an application or code region meets or exceeds that of the expected performance. By using the assertions, the programmer can relate expected performance results to variables in the application, the execution configuration (i.e. number of processors), and pre-evaluated variables (i.e. peak FLOPS for this machine). This technique allows users to encode their performance expectations for regions of code, confirm these expectations with empirical data, and even make runtime decisions about component selection based on this data. The use of performance assertions requires extensive annotation of source code, and relies on the application developer's experience and intuition in knowing where to insert the assertions, and what kind of performance result to expect.

Several strategies exist for reducing total power dissipated and energy consumed by a microprocessor. Power and energy saving techniques can be applied at the level of circuits, architectures, system software, and at the application layer[24]. Power analysis and optimizations at the system software and application layers have not been adequately explored, but some progress has been made in recent times. Seng and Tullsen[17] studied the effects of power and energy savings for both standard compiler optimizations and individual optimizations on the Pentium 4. Their experiments suggest that compiling for the best performance is equated with high energy savings. Valluri and John[22] performed a similar but more in-depth study on the Alpha 21264 processor. They found that optimizations that improve performance by reducing the instruction count are optimized for low energy. They also found that optimizations that improve performance by increasing the amount of overlap in execution of instructions increase average power dissipation in the processor. LUNA[5] is a high level power analysis framework for multicore NoC architectures. LUNA has been employed by the compiler to generate power profiles in the network that were used to generate directives which are stored at each router to direct the operation of dynamic-voltage-scalable (DVS) links. The COPPER project[1] applies dynamic compilation strategies for dynamic power management. They introduce techniques for compiler controlled dynamic register file reconfiguration and profile-driven dynamic clock frequency and voltage scaling. At the application layer, PowerPack[4] provides library routines that allow users to embed ACPI calls in applications and reduce the CPU's processing speed via DVS. Very few tools provide an automated framework that would enable the non-expert to successfully apply these optimization techniques to achieve low energy consumption and power dissipation rates in their applications.

V. Conclusion

Automated performance analysis depends both on the processing of multi-experiment performance data and on expert knowledge to direct the processing, interpret the results, and provide decision support. The research reported here represents our first attempt to integrate PerfExplorer's capabilities for capturing and automating performance analysis with tools for performance-directed modeling and optimization. The flexible programmatic support for analysis scripting and rule-based knowledge engineering in PerfExplorer has proven successful in the integration with the OpenUH compiler system and now sets the stage for more sophisticated feedback-directed compiler optimizations.

Our future work will explore several opportunities. The cost model calculation for OpenUH can be modified to integrate feedback from runtime performance to generate more accurate cost models. Different optimization priorities may apply, such as improving caching/memory strategies or utilizing processor functional units more effectively. The parallel model should be improved to feed in information to detect imbalances due to different amounts of work per thread in parallel loops. We also need to feed information with regard to sources of overhead and their causes, such as time spent in atomic operations, locking, and critical sections, and their correlation to the distribution of work in parallel sections and number of threads. In addition, there are strategies for variable privatization and first touch policies to reduce the number of remote memory references that can be better informed by automated performance analysis.

We plan to extend our performance and power inference rules with PerfExplorer and integrate the results with the OpenUH compiler cost model, thus enabling users to target optimizations based on both performance and power models. Furthermore, we will also extend our models to consider the impacts of architecture characteristics and application metadata on compilation strategies for improved power and energy efficiency.

ACKNOWLEDGMENTS

University of Oregon research is sponsored by contracts DE-FG02-07ER25826 and DE-FG02-05ER25680 from the MICS program of the U.S. DOE, Office of Science and NSF grant #CCF0444475. University of Houston research is sponsored by the NSF grants #CCF-0444468 and #CCF-0702775. Research at Argonne National Laboratory is supported through a CISE-BPC supplement under NSF grant #0444345 and DOE contract DE-AC02-06CH11357. We would like to thank Dr. Danesh Tafti from the Mechanical Engineering Department of Virginia Tech for providing us a good description of the GenIDLEST application and useful insights to facilitate the process of optimization.

REFERENCES

[1] A. Azevedo, R. Cornea, I. Issenin, R. Gupta, N. Dutt, A. Nicolau, and A. Veidenbaum, "Architectural and compiler strategies for dynamic power management in the copper project," in *IWIA '01: Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'01)*. Washington, DC, USA: IEEE Computer Society, 2001, p. 25.

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

[3] M. Burcea and M. Voss, "A runtime optimization system for OpenMP," in *WOMPAT*, 2003, pp. 42–53.

[4] K. W. Cameron, R. Ge, and X. Feng, "High-performance, power-aware distributed computing for scientific applications," *Computer*, vol. 38, no. 11, pp. 40–47, 2005.

[5] N. Eisley, V. Soteriou, and L.-S. Peh, "High-level power analysis for multi-core chips," in *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. New York, NY, USA: ACM, 2006, pp. 389–400.

[6] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr, "Scalable parallel trace-based performance analysis," in *Proc. 13th European PVM/MPI Users' Group Meeting*, ser. LNCS, vol. 4192. Bonn, Germany: Springer, September 2006, pp. 303–312.

[7] O. Hernandez, H. Jin, and B. Chapman, "Compiler support for efficient instrumentation," in *ParCo '07: Proceedings of the International Conference ParCo 2007*. Julich, Germany: NIC-Directors, 2007, pp. 661–668.

[8] 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)*, 2006.

[9] K. A. Huck, A. D. Malony, S. Shende, and A. Morris, "Scalable, automated performance analysis with tau and perfexplorer," in *Parallel Computing (ParCo)*, Aachen, Germany, 2007.

[10] S. Jarp, "A methodology for using the itanium-2 performance counters for bottleneck analysis," HP Labs, Tech. Rep., August 2002.

[11] J. Jorba, T. Margalef, and E. Luque, "Performance analysis of parallel applications with kappapi2," *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, vol. 33, pp. 155–162, 2006.

[12] L. Li and A. D. Malony, "Knowledge engineering for automatic parallel performance diagnosis," *Concurrrency and Computation: Practice and Experience*, 2006.

[13] 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*, 2006.

[14] A. Malony and R. Helm, "A Theory and Architecture for Automating Performance Diagnosis," *Future Generation Computer Systems*, vol. 18, no. 1, pp. 189–200, Sep. 2001, (Special issue on Performance Data-mining in Parallel and Distributed Computing).

[15] J. Marathe and F. Mueller, "Hardware profile-guided automatic page placement for ccnuma systems," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2006, pp. 90–99.

[16] G. D. Riley, J. M. Bull, and J. R. Gurd, "Performance improvement through overhead analysis: A case study in molecular dynamics," in *International Conference on Supercomputing*, 1997, pp. 36–43.

[17] J. S. Seng and D. M. Tullsen, "The effect of compiler optimizations on pentium 4 power consumption," in *INTERACT '03: Proceedings of the Seventh Workshop on Interaction between Compilers and Computer Architectures*. Washington, DC, USA: IEEE Computer Society, 2003, p. 51.

[18] S. Shende and A. D. Malony, "The TAU parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–331, Summer 2006. [Online]. Available: http://www.cs.uoregon.edu/research/tau

[19] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore, "An algebra for cross-experiment performance analysis," in *Proceedings of 2004 International Conference on Parallel Processing (ICPP'04)*, Montreal, Quebec, Canada, 2004, pp. 63–72. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ICPP.2004.1327905

[20] D. K. Tafti, "Genidlest - a scalable parallel computational tool for simulating complex turbulent flows," in *Proceedings of the ASME Fluids Engineering Division*, November 2001.

[21] J. Thompson, D. Higgins, and T. Gibson, "CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice," *Nucl. Acids Res.*, vol. 22, pp. 4673–4680, 1994.

[22] M. Valluri and L. John, "Is compiling for performance == compiling for power," 2001. [Online]. Available: citeseer.ist.psu.edu/valluri01is.html

[23] V.Bui, B. Norris, L. McInnes, K. Huck, O. Hernandez, L. Li, and B. Chapman, "A component infrastructure for performance and power modeling of parallel scientific applications," in *Component-Based High Performance Computing*, 2008.

[24] V. Venkatachalam and M. Franz, "Power reduction techniques for microprocessor systems," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 195–237, 2005.

[25] J. S. Vetter and P. H. Worley, "Asserting performance expectations," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–13. [Online]. Available: http://portal.acm.org/ft_gateway.cfm?id=762809&type=pdf&coll=ACM&dl=ACM%&CFID=52735087&CFTOKEN=45432917

[26] M. J. Voss and R. Eigemann, "High-level adaptive program optimization with ADAPT," *ACM SIGPLAN Notices*, vol. 36, no. 7, pp. 93–102, 2001.

[27] G. Wang and D. K. Tafti, "Uniprocessor performance enhancement with additive schwarz preconditioners on origin 2000," *Adv. Eng. Softw.*, vol. 29, no. 3-6, pp. 425–431, 1998.

[28] F. Wolf and B. Mohr, "Automatic performance analysis of SMP cluster applications," Research Centre Julich, Tech. Rep. 05, 2001.

[29] M. E. Wolf, D. E. Maydan, and D.-K. Chen, "Combining loop transformations considering caches and scheduling," in *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 274–286.

[30] Y. Zhang and M. Voss, "Runtime empirical selection of loop schedulers on hyperthreaded smps." in *IPDPS*, 2005.

[31] A. Zomaya, Ed., *Parallel Computing for Bioinformatics and Computational Biology: Models, Enabling Technologies, and Case Studies*, 1st ed., ser. Wiley Series on Parallel and Distributed Computing. Wiley Interscience, 2006.