PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications

Martin Burtscher[1], Byoung-Do Kim[2], Jeff Diamond[3], John McCalpin[2], Lars Koesterke[2], and James Browne[3]
[1]Institute for Computational Engineering and Sciences, The University of Texas at Austin
[2]Texas Advanced Computing Center, The University of Texas at Austin
[3]Department of Computer Science, The University of Texas at Austin

***Abstract*—HPC systems are notorious for operating at a small fraction of their peak performance, and the ongoing migration to multi-core and multi-socket compute nodes further complicates performance optimization. The readily available performance evaluation tools require considerable effort to learn and utilize. Hence, most HPC application writers do not use them. As remedy, we have developed PerfExpert, a tool that combines a simple user interface with a sophisticated analysis engine to detect probable core, socket, and node-level performance bottlenecks in each important procedure and loop of an application. For each bottleneck, PerfExpert provides a concise performance assessment and suggests steps that can be taken by the programmer to improve performance. These steps include compiler switches and optimization strategies with code examples. We have applied PerfExpert to several HPC production codes on the Ranger supercomputer. In all cases, it correctly identified the critical code sections and provided accurate assessments of their performance.**

***Index Terms*—Bottleneck diagnosis, HPC systems, multicore performance, performance analysis, performance metric**

I. Introduction

Most HPC applications attain only a small fraction of the potential performance on modern supercomputers. Emerging multi-core and multi-socket cluster nodes greatly increase the already high dimensionality and complexity of performance optimization. Performance optimization requires not only identification of code segments that are bottlenecks but also characterization of the causes of the bottlenecks and determination of code restructurings that will improve performance. While identification of code segments that may be bottlenecks can be accomplished with simple timers, characterization of the cause of the bottleneck requires more sophisticated measurements such as the use of hardware performance counters.

Most modern high-end microprocessors contain multiple performance counters that can each be programmed to count one of hundreds of events [4]. Many of these events have cryptic descriptions that only computer architects understand, making it difficult to determine the right combination of events to track down a performance bottleneck. Moreover, correct interpretation of performance counter results can often only be accomplished with detailed architectural knowledge. For example, on Opteron CPUs, L1 cache miss counts exclude misses to lines that have already been requested but are not yet in the cache, which may make it appear as though there is no problem with memory accesses even when memory accesses are the primary bottleneck. Diagnosing performance problems thus requires in-depth knowledge of the hardware as well as of the compiler and the system software. However, most HPC application writers are domain experts who are not and should not have to be familiar with the intricacies of each system on which they want to run their code.

SC10 November 2010, New Orleans, Louisiana, USA 978-1-4244-7558-2/10/$26.00

There are several widely available performance measurement tools, including HPCToolkit [28], Open|SpeedShop [20], PAPI [21], and Tau [26], that can be used to obtain performance counter measurements. These tools generally provide little guidance for selecting which measurements to make or how to interpret the resulting counter values. Hence, designing and making sense of the measurements requires considerable architectural knowledge, which changes from system to system. None of these tools provide guidance on how to restructure code segments to alleviate bottlenecks once they have been identified. Thus, these tools provide only a part, albeit an essential part, of the solution. As a result, characterizing and minimizing performance bottlenecks on multicore HPC systems with today's performance tools is an effort-intensive and difficult task for most application developers. A 2009 survey of Ranger users showed that fewer than 25% had used any of the several performance tools available on Ranger.

To make performance optimization more accessible to application developers and users, we have designed and implemented PerfExpert, a tool that captures and uses the architectural, system software and compiler knowledge necessary for effective performance bottleneck diagnosis. Hidden from the user, PerfExpert employs the existing measurement tool HPCToolkit [28] to execute a structured sequence of performance counter measurements. Then it analyzes the results of these measurements and computes performance metrics to identify potential bottlenecks at the granularity of six categories. For the identified bottlenecks in each key code section, it recommends a list of possible optimizations, including code examples and compiler switches that are known to be useful for speeding up bottlenecks belonging to the same category. Thus, PerfExpert makes the extensive knowledgebase needed for bottleneck diagnosis available to HPC application writers. In summary, PerfExpert is an expert system for automatically identifying and characterizing intrachip and intranode performance bottlenecks and suggesting solutions to alleviate the bottlenecks, hence the name PerfExpert.

Fig. 1 compares the workflow of a typical code optimization process using performance evaluation tools that only automate the measurement stage with the corresponding workflow using PerfExpert. When optimizing an application with generic profiling tools, users normally follow an iterative process involving multiple stages, and each stage has to be conducted manually. Moreover, the decision making is left to the user and is thus based on his or her (possibly limited) performance evaluation and system knowledge. In contrast, most of this process is automated with PerfExpert. In particular, the measurement and bottleneck determination processes (dotted box) are automatically executed by PerfExpert. Even for the optimization implementation, PerfExpert guides the user by suggesting optimizations from its database. This degree of automation was made possible by confining the domain of analyses to the core, chip and node level.

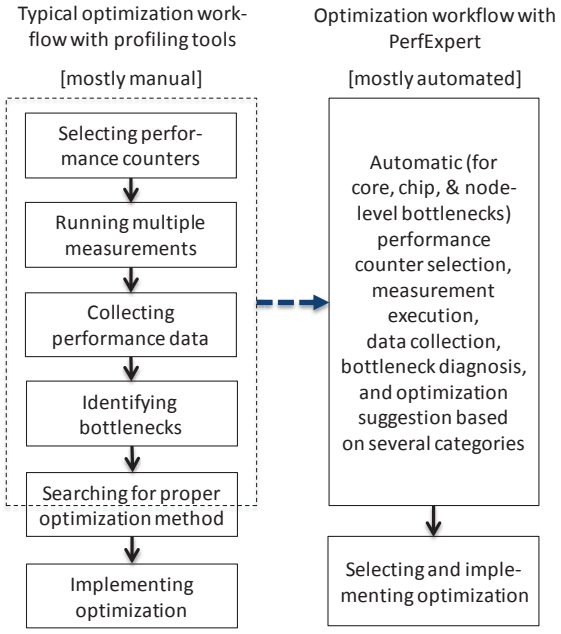

Fig. 1. Profiling and optimization workflow with generic measurement tools (left) and with PerfExpert (right)

The guiding principle behind the design of PerfExpert is to obtain simplicity of use by automating the complex measurement and analysis tasks and embedding expert knowledge about the underlying architecture into the diagnosis of the bottlenecks. PerfExpert is launched through a simple command line (if the job submission system is not used) that only takes two parameters: one parameter controls the amount of output to be generated and the other parameter is the command needed to start the application to be measured. PerfExpert automatically determines which performance experiments to run, and its output is simple and graphical. It is intended to make performance assessment easy, fast, and available to users with little or no performance optimization expertise. Performance experts may also find PerfExpert useful because it automates many otherwise manual steps. However, expert users will probably also want to see the raw performance data.

To successfully accomplish analysis and characterization of performance bottlenecks, we found it necessary to develop a new performance metric. This metric combines performance counter measurements with architectural parameters to compute upper bounds on local cycle-per-instruction (LCPI) contributions of various instruction categories at the granularity of loops and procedures. LCPI is designed to naturally lead to specific bottlenecks, to highlight key performance aspects, and to hide misleading and unimportant details (Sections II & IV).

PerfExpert has been developed for and implemented on the Ranger supercomputer at the Texas Advanced Computing Center. In the current version, the analysis and characterization of the performance of each loop and procedure is based on 15 performance counter measurements and 11 chip- and architecture-specific resource characteristics. These parameters and counter values, which are defined and discussed in Section III, are available or derivable for the standard Intel, AMD, and IBM chips as well as the node structures typical of current supercomputers, allowing PerfExpert to be ported to systems that are based on other chips and architectures.

We have analyzed several HPC production codes on Ranger using PerfExpert. In all instances, PerfExpert correctly determined the important code sections along with their performance bottlenecks. With help from the original application writers, we studied and optimized these code sections to verify that the optimizations suggested by PerfExpert are useful. In this way, PerfExpert has, for example, been helpful in speeding up a global Earth mantle convection simulation running on 32,768 cores by 40%, even though we "only" performed node-level optimizations.

This paper makes the following contributions:

- It introduces a new performance bottleneck diagnosis tool for HPC application writers, called PerfExpert, that is easy to use because it only requires the command line of the application to be measured. It automatically evaluates the core, chip, and node-level performance, including determining which performance counters to use, analyzing the results, determining potential bottlenecks, and outputting only essential information.
- It presents the novel LCPI metric that combines performance counter measurements with architectural parameters to make the measurements comparable, which is crucial for determining the relative severity of potential bottlenecks. LCPI makes it easy to see what aspect of a code section accounts for most of the runtime and therefore represents a key optimization candidate.
- It evaluates PerfExpert and the LCPI metric on actual HPC production codes running on Ranger and reports our experiences and findings with these applications.

The rest of this paper is organized as follows. Section II describes PerfExpert in detail. Section III presents the evaluation methodology. Section IV discusses the results. Section V summarizes related work. Section VI concludes with a summary and future work.

II. Design

This section describes PerfExpert's operation and use, its LCPI performance metric, as well as the user interface.

A. Performance Metric

The primary performance metric used in PerfExpert is *local cycles per instruction* (LCPI). It is local because separate CPI

values are computed for each procedure and loop. PerfExpert computes and reports the total LCPI for each procedure and loop as well as upper bounds for separate contributions to the total LCPI from six operation classes or *categories*: data memory accesses, instruction memory accesses, floating-point operations, branches, data Translation Lookaside Buffer (TLB) accesses, and instruction TLB accesses.

The LCPI is essentially the procedure or loop runtime normalized by the amount of work performed. We found this normalization to be important when combining measurements from multiple runs because some timing dependent nondeterminism is common in parallel programs. For example, it is unlikely that multiple balanced threads will reach a synchronization primitive in the same order every time the program executes. Hence, an application may spend more or fewer cycles in a code section compared to a previous run, but the instruction count is likely to increase or decrease concomitantly. Hence, the (normalized) LCPI metric is more stable between runs than absolute metrics such as cycle or instruction counts.

Currently, PerfExpert measures 15 different event types (Section II.A.1) to compute the overall LCPI and the LCPI contribution of the six categories. Because CPUs only provide a limited number of performance counters, e.g., an Opteron core can count four event types simultaneously, PerfExpert automatically runs the same application multiple times. To be able to check the variability between runs, one counter is always programmed to count cycles. The remaining counters are configured differently in each run to obtain information about data memory accesses, branch instruction behavior, etc. To limit the variability and possible resulting inconsistencies, events whose counts are used together are measured together if possible. For example, PerfExpert performs all floating-point related measurements in the same experiment.

Based on the performance counter measurements, PerfExpert computes an (approximate) *upper bound* of the latency caused by the measured LCPI contribution for the six categories to narrow down the possible causes of bottlenecks for the code sections with a high LCPI. We are interested in computing upper bounds for the latency, i.e., worst case scenarios, because if the estimated maximum latency of a category is sufficiently low, the corresponding category cannot be a significant performance bottleneck. For instance, the branch category's LCPI contribution for a given code section is:

(**BR_INS** * *BR_lat* + **BR_MSP** * *BR_miss_lat*) / **TOT_INS**

Here, bold print denotes performance counter measurements for the code section and italicized print indicates system constants. **BR_INS**, **BR_MSP**, and **TOT_INS** are the measured number of branch instructions, branch mispredictions, and total instructions executed, respectively. *BR_miss_lat* and *BR_lat* are the CPU's branch mispredictions latency and branch latency in cycles. Thus, the above expression in parentheses represents an upper bound of cycles due to branching related activity. It is an upper bound because the latency is typically not fully exposed in a superscalar CPU like the current CPUs from AMD, Intel, IBM, etc., which can execute multiple instructions in parallel and out-of-order, thereby hiding some of this latency. Dividing the computed number of cycles by the measured number of executed instructions yields the LCPI contribution due to branch activity for a given code section. Upper bounds on the LCPI contribution of the other categories are computed similarly. For data memory accesses, PerfExpert uses the following expression:

(**L1_DCA***L1_lat*+**L2_DCA***L2_lat*+**L2_DCM***Mem_lat*)/**TOT_INS**

This is the number of L1 data cache accesses times the L1 data cache hit latency plus the number of data accesses to the L2 cache times the L2 cache hit latency plus the number of data accesses that missed in the L2 cache times the memory access latency divided by the total number of executed instructions. L3 accesses will be discussed shortly. Again, this LCPI contribution represents an upper bound because of CPU and memory parallelism. Note that *Mem_lat* is not a constant because the latency of an individual load can vary greatly depending on the DRAM bank and page rank it accesses and memory traffic generated by other cores on the same chip, to name just a few factors. Fortunately, PerfExpert is dealing, at the very least, with millions of memory accesses, which tend to average out so that a reasonable upper bound for *Mem_lat* can be used. However, this opens up the possibility of underestimating the true memory latency, in which case the LCPI contribution is not an upper bound. Selecting a conservative *Mem_lat* makes this unlikely in practice because experience with multiple codes on a given architecture enables the *Mem_lat* value to be chosen judiciously.

Aside from not being overly susceptible to the inherent nondeterminism of parallel programs, PerfExpert's performance metric has the following benefits and abilities.

1) Highlighting key aspects. For example, a program with a small L1 data cache miss ratio can still be impeded by data accesses. If the program executes mostly dependent load instructions, the Opteron's L1 data cache hit latency of three cycles will limit execution to one instruction per three cycles, which is an order of magnitude below peak performance. The LCPI metric correctly accounts for this possibility.

2) Hiding misleading details. For instance, if a program executes thousands of instructions, two of which are branches and one of them is mispredicted, the branch mispredictions ratio is 50%, which is very bad. However, it does not matter because so few branches are executed. The LCPI contribution metric will not report a branch problem in this case because the total number of cycles due to branching is miniscule.

3) Summarization ability. For example, instead of listing a hit or miss ratio for every cache level, PerfExpert's performance metric can combine this information into a single meaningful metric, i.e., the data access LCPI, to reduce the amount of output without losing important information.

4) Extensibility. If a future or different CPU generation supports a new instruction category (as well as countable events for it), it should be straightforward to define an LCPI computation for the new category and include it in the output.

5) Refinability. If more diagnostically effective performance counter events become available, the existing LCPI calculations can be improved to make the upper bounds more accurate. For example, with hit and miss counts for the shared L3 cache due to individual cores, the above LCPI computation for data accesses can be refined by replacing the term **L2_DCM***Mem_lat* with **L3_DCA***L3_lat*+**L3_DCM***Mem_lat*.

1. Performance counters and system parameters

PerfExpert currently measures the following 15 performance counter events on each core for the executed proce-

dures and loops: total cycles, total instructions, L1 data cache accesses, L1 instruction cache accesses, L2 cache data accesses, L2 cache instruction accesses, L2 cache data misses, L2 cache instruction misses, data TLB misses, instruction TLB misses, branch instructions, branch mispredictions, floating-point instructions, floating-point additions and subtractions, and floating-point multiplications.

The LCPI metric combines these measurements with 11 system parameters to make them comparable, which is crucial for determining the relative severity of potential bottlenecks. CPU cycles have successfully been used in other analyzes as unifying metric to compare results [11]. The eleven system parameters and their values for Ranger are: L1 data cache hit latency (3), L1 instruction cache hit latency (2), L2 cache hit latency (9), floating-point add/sub/mul latency (4), maximum floating-point div/sqrt latency (31), branch latency (2), maximum branch misprediction penalty (10), CPU clock frequency (2,300,000,000), TLB miss latency (50), memory access latency (310). It further uses a "good CPI threshold" (0.5), which is used for scaling the performance bars in the output. The first eight parameters are constant or vary little, so the maximum values are chosen. The TLB miss latency and the memory access latency are highly variable and system dependent. PerfExpert uses conservative values that are not guaranteed but likely to result in an upper bound for most applications. The values are based on the expert opinions of the authors and may well be adjusted as we gain more experience. More information about Ranger is provided in Section III.A.

B. Operation

In part because of the job submission systems common to most supercomputers and in part to make it possible to repeat the analysis with different thresholds (see below), PerfExpert comprises two stages: a measurement stage and a diagnosis stage. The measurements are passed through a single file from the first to the second stage, making it easy to preserve the results. The diagnosis stage supports correlating multiple measurements from the same application. This is useful for detecting bottlenecks in on-chip resources that are shared by multiple cores (Section II.C.2) and for tracking the optimization progress as application code is being improved (Section IV.C).

1. Measurement stage

On Ranger, PerfExpert's measurement stage consists of slightly modifying a provided job submission script, where the user has to specify the project name, the path to the executable, the command line, the number of threads and tasks, and an estimate of the runtime. This information is also required in regular submission scripts. Once the submitted job starts, PerfExpert automatically runs the application several times on top of HPCToolkit to gather the necessary performance counter data. At the end, it stores the measurements in a file.

HPCToolkit [13] uses performance counter sampling to measure program performance at the procedure and loop level and correlates these measurements with the program's source code. It works with unmodified, multilingual, and optimized binaries, incurs low overhead, and scales to large systems.

2. Diagnosis stage

PerfExpert's diagnosis stage requires two or three inputs from the user: 1) a threshold, 2) the path to a measurement file produced by the first stage, and, optionally, 3) the path to a second measurement file for comparison. The diagnosis stage first checks the variability, runtime, and consistency of the data in the measurement file, which typically contains results from multiple cores and multiple HPCToolkit experiments. PerfExpert emits a warning if the runtime is too short to gather reliable results or if the runtime of important procedures or loops varies too much between experiments. Furthermore, PerfExpert checks the consistency of the data to validate the assumed semantic meaning of the performance counters, e.g., the number of floating-point additions must not exceed the number of floating-point operations.

Once the data are deemed reliable, PerfExpert determines the hottest procedures and loops, computes the LCPI performance metrics for them, and outputs the resulting performance assessment. To help the user focus on important code regions, PerfExpert only generates assessments for the top few longest running code sections. The user can control for how many code sections an assessment should be output by changing the threshold. A lower threshold will result in more code sections being assessed, which is useful when multiple important code sections have similar runtimes or when users cannot or do not want to optimize the top few code sections. For example, the *HOMME* benchmark (Section III.B.2) has ten procedures that represent between 5% and 13% of the total runtime, and we found the bottom five of them, which account for 28% of the application's runtime, to be easier to optimize.

C. Output

1. Analyzing a single input

Fig. 2 shows the output generated by PerfExpert for a simple 2000 by 2000 element matrix-matrix multiplication (*MMM*) that uses a bad loop order. The output first lists the name of the measurement file (mmm) and the total runtime. The next two lines specify where the suggested code optimizations, compiler flags, and examples for bottleneck remediation for each category can be found. The suggestions are not directly included so as not to clutter the output. The rest of the output is the performance assessment.

```
total runtime in mmm is 166.00 seconds

Suggestions on how to alleviate performance bottlenecks are available at:
http://www.tacc.utexas.edu/perfexpert/

matrixproduct (99.9% of the total runtime)
-------------------------------------------------------------------------------
performance assessment   great.....good......okay......bad.......problematic
- overall                >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
upper bound by category
- data accesses          >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- instruction accesses   >>>>>>>
- floating-point instr   >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- branch instructions    >>
- data TLB               >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- instruction TLB        >
```

Fig. 2. Output for *MMM*

For each critical procedure and loop (only the top procedure is shown in this example), PerfExpert lists its name (matrixproduct) and the fraction of the total runtime that it represents, followed by the performance assessment below the dashed line. The length of the bar made of ">" symbols specifies how bad the performance is. The overall assessment for *MMM* is "problematic". The remaining assessments list the upper bounds on the LCPI contribution of six categories: data memory accesses, instruction memory accesses, floating-point in-

structions, branch instructions, data TLB accesses, and instruction TLB accesses. In the example, branch instructions as well as instruction memory and TLB accesses are not a problem, as one might expect from the small *MMM* kernel. The kernel executes mostly memory accesses that miss in the cache and TLB as well as dependent (and therefore slow) multi-cycle floating-point instructions. The corresponding categories and the overall performance are correctly assessed as problematic. Note that PerfExpert users do not have to know, e.g., what a TLB is. The category names can simply be used as labels for identifying which suggested optimizations the user may want to consider applying.

2. Correlating two inputs

Fig. 3 shows the output for *DGELASTIC*, a global earthquake simulation code based on the *MANGLL* library (Section III.B.1), with two inputs, one with one thread per chip and one with four threads per chip, to illustrate PerfExpert's correlation ability to detect bottlenecks in shared resources. Comparison between executions has been pioneered by eGprof [25].

```
total runtime in dgelastic_4 is 196.22 seconds
total runtime in dgelastic_16 is 75.70 seconds

Suggestions on how to alleviate performance bottlenecks are available at:
http://www.tacc.utexas.edu/perfexpert/

dgae_RHS (runtimes are 136.93s and 45.27s)
-------------------------------------------------------------------------------
performance assessment   great.....good......okay......bad.......problematic
- overall                >>>>>>>>>>>>>>>>>>>>>>2222222
upper bound by category
- data accesses          >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- instruction accesses   >>>>>>>>>
- floating-point instr   >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>1
- branch instructions    >>
- data TLB               >
- instruction TLB        >
```

Fig. 3. Output for *DGELASTIC* correlating two runs

The format of the correlated output is almost identical to the format with one input except that both database paths and their total runtimes are listed and that absolute runtimes are given for each critical code section. The difference in the metrics between the two inputs is expressed with 1's and 2's at the end of the performance bars. The number of 1's indicates how much worse the first input is than the second input. Similarly, 2's indicate that the second input is worse than the first. In Fig. 3, the upper LCPI bound for floating-point instructions in the dgae_RHS procedure is slightly worse with four threads per node than with 16 threads per node. More importantly, the overall performance is substantially worse with 16 threads per node, which highlights a known problem with many multicore processors, including the quad-core Opteron: they do not provide enough memory bandwidth for all cores when running memory intensive codes. This performance problem is borne out by the row of 2's. The upper bound estimates are basically the same between the two runs, which they should be because upper bounds are independent of processor load.

Given PerfExpert's assessment of *DGELASTIC*, it is easy to see that shared resources (i.e., scaling), data accesses, and floating-point instructions are potential performance bottlenecks in the critical dgae_RHS procedure. Note that this information alone is already valuable. For example, the authors of *DGELASTIC* assumed their code to be compute bound until we performed our analysis. Based on our assessment, they refocused their optimization efforts to target memory accesses, which yielded substantial speedups.

3. Optimization suggestions

PerfExpert goes an important step further by providing an extensive list of possible optimizations to help users remedy the detected bottlenecks. These optimizations are accessible through a web page, which catalogs code transformations and compiler switches for each performance assessment category. A much simplified version of the floating-point instruction category is given in Fig. 4. For each category, there are several subcategories that list multiple suggested remedies. The suggestions include code examples (a through d) or Intel compiler switches (e) to assist the user. For example, an application writer may not remember what the distributivity law is (a), but upon seeing the code example, it should be clear what pattern to look for in the code and how to make it faster.

If floating-point instructions are a problem

Reduce the number of floating-point instructions

a) eliminate floating-point operations through distributivity
*d[i] = a[i] * b[i] + a[i] * c[i]; → d[i] = a[i] * (b[i] + c[i]);*

Avoid divides

b) compute the reciprocal outside of loop and use multiplication inside the loop
*loop i {a[i] = b[i] / c;} → cinv = 1.0 / c; loop i {a[i] = b[i] * cinv;}*

Avoid square roots

c) compare squared values instead of computing the square root
*if (x < sqrt(y)) {} → if ((x < 0.0) || (x*x < y)) {}*

Speed up divide and square-root operations

d) use float instead of double data type if loss of precision is acceptable
double a[n]; → float a[n];

e) allow the compiler to trade off precision for speed
use the "-prec-div", "-prec-sqrt", and "-pc32" compiler flags

Fig. 4. Simplified list of optimizations with examples

We envision the following usage of this information. For example, after running PerfExpert on *DGELASTIC*, the programmer would know that data memory accesses are the problem but may not know how to go about fixing this problem. The web page provides many suggestions for optimizing data memory accesses, thus guiding the programmer and helping him or her get started with the performance optimization. A simplified version of this section of the web page (without code examples for brevity) is provided in Fig. 5. Studying the code of the dgae_RHS procedure will reveal that suggestions (a), (b), and (e) do not apply because the code linearly streams through large amounts of data. Suggestions (g) and (i) also do not apply because the code only uses a few large arrays. We believe eliminating inapplicable suggestions in this way can be done by someone familiar with the code who is not a performance expert.

If data accesses are a problem

Reduce the number of memory accesses

a) copy data into local scalar variables and operate on the local copies

b) recompute values rather than loading them if doable with few operations

c) vectorize the code

Improve the data locality

d) componentize important loops by factoring them into their own procedures

e) employ loop blocking and interchange (change the order of memory accesses)

f) reduce the number of memory areas (e.g., arrays) accessed simultaneously

g) split structs into hot and cold parts and add pointer from hot to cold part

Other

h) use smaller types (e.g., float instead of double or short instead of int)

i) for small elements, allocate an array of elements instead of individual elements

j) align data, especially arrays and structs

k) pad memory areas so that temporal elements do not map to same cache set

Fig. 5. Simplified list of optimizations without examples

The next step is to test the remaining suggestions. We have experimentally verified suggestions (c), (j), and (k) to improve the performance substantially. We were unable to apply suggestion (f) without breaking suggestion (c). However, suggestion (f) aims at reducing cache conflict misses and DRAM bank conflicts, which were already addressed by applying suggestion (k). We have not yet tried suggestions (d) and (h) but believe that they will help speed up the code further. While the user has to try out the suggested optimizations to see which ones apply and work, PerfExpert's suggestions can be invaluable in helping an otherwise lost application developer getting started with the performance tuning.

D. Performance Metric Discussion

PerfExpert explicitly targets intra-node performance to help users with problems related to multi-core and multi-socket issues. Optimizing such problems can have a large performance impact on a parallel application, even when running on many nodes. For example, the intra-node optimizations we applied to *DGADVEC* (Section III.B.1) resulted in a combined speedup of around 40% on a 32,768-core run, which is akin to having over 13,000 additional cores. Note that PerfExpert also assesses the procedures in the communication library if they represent a sufficient fraction of the total runtime.

Like any performance evaluation tool, PerfExpert may produce incorrect assessments. For example, a false positive can be produced for a code section that misses in the L1 data cache a lot but contains enough independent instructions to fully hide the L2 access latency. In this case, PerfExpert may list the code section as having a data access problem, even though optimizing the data accesses will not improve performance. False negatives, i.e., missing actual or potential bottlenecks, are also possible but unlikely because the upper bounds have a tendency to overestimate the severity. Finally, it is possible that an application has a performance bottleneck that is not captured by PerfExpert's categories. The current measurements and analyses target what our experience has taught us is important and what the performance counters can measure. We expect to improve the effectiveness of the assessment as more experience with PerfExpert accumulates.

PerfExpert indicates whether the performance metrics are in the good, bad, etc. range, but deliberately does not output exact values. Rather, it prints bars that allow the user to quickly see which category is the worst so he or she can focus on that. In this sense, the performance assessment is relative instead of absolute, meaning that the value-to-range assignment does not have to be precise. This way, we avoid having to define exactly what constitutes a "good" CPI, which is application dependent, and can instead use a fixed value per system.

In some cases, it may be of interest to subdivide the data access category to separate out the individual cache levels. For example, the array blocking optimization requires a blocking factor that depends on the cache size and is therefore different depending on which cache level represents the main bottleneck. However, most of our recommended optimizations help no matter which level of the memory hierarchy is the problem. For this reason and to keep PerfExpert simple, we currently provide only one data access category. Of course, resolution of data accesses to multiple levels can be readily added if this addition leads to worthwhile improvement in optimizations.

III. Evaluation Methodology

A. System

PerfExpert is installed on the Ranger supercomputer [24], a Sun Constellation Linux cluster at the Texas Advanced Computing Center (TACC). Ranger consists of 3,936 quad-socket, quad-core SMP compute nodes built from 15,744 AMD Opteron processors. In total, the system includes 62,976 compute cores and 123 TB of main memory. Ranger has a theoretical peak performance of 579 TFLOPS. All compute nodes are interconnected using InfiniBand in a seven-stage full-CLOS fat-tree topology providing 1 GB/s point-to-point bandwidth.

The quad-core 64-bit AMD Opteron (Barcelona) processors are clocked at 2.3 GHz. Each core has a theoretical peak performance of 4 FLOPS/cycle, two 128-bit loads/cycle from the L1 cache, and one 128-bit load/cycle from the L2 cache. This amounts to 9.2 GFLOPS per core, 73.6 GB/s L1 cache bandwidth, and 36.8 GB/s L2 cache bandwidth. The cores are equipped with four 48-bit performance counters and a hardware prefetcher that prefetches directly into the L1 data cache. Each core has separate 2-way associative 64 kB L1 instruction and data caches, a unified 8-way associative 512 kB L2 cache, and each processor has one 32-way associative 2 MB L3 cache that is shared among the four cores.

B. Applications

We have tested PerfExpert on the following production codes that represent various application domains and programming languages. They were all compiled with the Intel compiler version 10.1.

1. MANGLL/DGADVEC

MANGLL is a scalable adaptive high-order discretization library. It supports dynamic parallel adaptive mesh refinement and coarsening (AMR), which is essential for the numerical solution of the partial differential equations (PDEs) arising in many multiscale physical problems. *MANGLL* provides nodal finite elements on domains that are covered by a distributed hexahedral adaptive mesh with 2:1 split faces and implements the associated interpolation and parallel communication operations on the discretized fields. The library has been weakly scaled to 32,768 cores on Ranger, delivering a sustained performance of 145 TFLOPS. *DGADVEC* [6] is an application built on *MANGLL* for the numerical solution of the energy equation that is part of the coupled system of PDEs arising in convection simulations, describing the viscous flow and temperature distribution in Earth's mantle. *MANGLL* and *DGADVEC* are written in C.

2. HOMME

HOMME (High Order Method Modeling Environment) is an atmospheric general circulation model (AGCM) consisting of a dynamic core based on the hydrostatic equations, coupled to a sub-grid scale model of physical processes [31]. We use the benchmark version of *HOMME*, which was one of NSF's acceptance benchmark programs for Ranger. It solves a modified form of the hydrostatic primitive equations with analytically specified initial conditions in the form of a baroclinically unstable mid-latitude jet for a period of twelve days, following an initial perturbation [23]. Whereas the general version is designed for using hybrid parallel runs (both MPI and

OpenMP), the benchmark version uses MPI-only parallelism. Although a semi-implicit scheme is used for time integration, the benchmark version is simplified and spends most of its time in explicit finite difference computation on a static regular grid. It is written in Fortran 95.

3. LIBMESH/EX18

The *LIBMESH* library [15] provides a framework for the numerical approximation of partial differential equations using continuous and discontinuous Galerkin methods on unstructured hybrid meshes. It supports parallel adaptive mesh refinement (AMR) computations as well as 1D, 2D, and 3D steady and transient simulations on a variety of popular geometric and finite element types. The library includes interfaces to solvers such as PETSc for the solution of the resulting linear and nonlinear algebraic systems. We use example 18 (*EX18*) of the *LIBMESH* release [17], which solves an unsteady nonlinear system of Navier-Stokes equations for low-speed incompressible fluid flow. *EX18* performs a large amount of linear algebra computations and solves the transient nonlinear problem using the heavily object-oriented FEMSystem class framework. *LIBMESH* and *EX18* are written in C++.

4. ASSET

ASSET (Advanced Spectrum Synthesis 3D Tool) is an astrophysical application that allows computing spectra from 3-dimensional input models as they are provided by hydrodynamical (CFD) simulations of the Sun and other stars. *ASSET* is fully parallelized with OpenMP and MPI. On clusters and on multi-socket workstations, a hybrid setup usually results in the best performance. A single MPI task is started on every socket and OpenMP threads are spawned according to the number of cores per socket. Scaling with OpenMP on quad-core CPUs is good. No domain decomposition is applied for the MPI parallelization, and different MPI tasks handle different and independent frequencies. Information is only communicated at the beginning and at the end of a calculation. Consequently, the MPI scaling is very good and only limited by load balancing because the amount of work per frequency varies slightly. *ASSET* is written in Fortran 90.

IV. RESULTS

We determined the most frequently executed procedures (and loops) of the applications described in the previous section using Tau [30], PAPI [21], HPCToolkit [13], and PIN [22] and assessed the performance of a selection of them to validate PerfExpert. The tools agreed on the top procedures (and loops) and the relative assessment of the performance bottlenecks but differed in some of the details, such as the absolute number of cache misses.

This section presents the results of applying PerfExpert to the four Ranger production codes, which demonstrate the various features of PerfExpert and highlight some interesting performance aspects of the four HPC production codes. To establish the usefulness of PerfExpert's suggestions, we optimized the key code sections of several of these applications and compared what we had to do to improve performance with PerfExpert's recommendations. To keep the output small, we only show the assessment for procedures (no loops) that account for at least 10% of the runtime.

A. DGADVEC

DGADVEC is dominated by two procedures that together account for over half of the total runtime. They contain several important loops that perform a large number of small dense matrix-vector operations. Even though these loops touch hundreds of megabytes of data, they have L1 data-cache miss ratios below 2% in part because of the hardware prefetcher, which is able to prefetch the data directly into the L1 cache. Yet, the loops execute only half an instruction or less per cycle, which is quite low.

Our analysis identified the L1 load-to-use hit latency of three cycles to be the main culprit for this poor performance. This latency cannot be hidden because there are not enough independent instructions available to execute. In other words, *DGADVEC* is memory bound and the primary performance bottleneck is accesses to the L1 data cache. Because of its low L1 miss ratio, this application was previously believed to be compute bound.

Since the L1 load-to-use hit latency is fixed in hardware, we can only reduce the average load-to-use latency by increasing the bandwidth, i.e., reading and writing multiple data items per memory transaction through the use of SSE instructions. Unfortunately, neither the Intel nor the PGI compiler vectorizes the memory accesses in these loops. Hence, we rewrote the loops so that the compiler emits SSE instructions. The full set of code modifications we made is described elsewhere [12].

```
total runtime in dgadvec is 681.74 seconds

Suggestions on how to alleviate performance bottlenecks are available at:
http://www.tacc.utexas.edu/perfexpert/

dgadvec_volume_rhs (29.4% of the total runtime)
-----------------------------------------------------------------------------
performance assessment   great.....good......okay......bad.......problematic
- overall                >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
upper bound by category
- data accesses          >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- instruction accesses   >>>>>>>>
- floating-point instr   >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- branch instructions    >
- data TLB               >
- instruction TLB        >

dgadvecRHS (27.0% of the total runtime)
-----------------------------------------------------------------------------
performance assessment   great.....good......okay......bad.......problematic
- overall                >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
upper bound by category
- data accesses          >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- instruction accesses   >>>>>>>>
- floating-point instr   >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- branch instructions    >>
- data TLB               >
- instruction TLB        >

mangll_tensor_IAIx_apply_elem (14.9% of the total runtime)
-----------------------------------------------------------------------------
performance assessment   great.....good......okay......bad.......problematic
- overall                >>>>>>>>>>>>
upper bound by category
- data accesses          >>>>>>>>>>>>>>>>>
- instruction accesses   >>>>>>>>>>
- floating-point instr   >>>>>>>>>>>>>>>>>
- branch instructions    >>>>>>>
- data TLB               >
- instruction TLB        >
```

Fig. 6. Assessment of *DGADVEC*

Comparing the old and new loop implementations, we found that the number of executed instructions is 44% lower and the number of L1 data-cache accesses is 33% lower due to the vectorization. Note, however, that we did not rewrite *DGADVEC* to fully incorporate our changes because the authors had moved on to write *DGELASTIC*, a new application that simulates the global propagation of earthquake waves. As *DGELASTIC* is also based on the *MANGLL* library, we implemented our changes in the new application. While this

work has not yet been completed, we already see great benefits. For example, the key loop in *DGELASTIC*, which accounts for over 60% of the total execution time (Fig. 3), is vectorized by the compiler and executes 1.4 instructions per cycle, representing a more than two-fold improvement over the *DGADVEC* loop performance. Note, however, that the two *MANGLL*-based applications solve different problems and are therefore not entirely comparable.

Looking at PerfExpert's assessment of *DGADVEC* shown in Fig. 6, we find that it correctly identifies the main procedures. Moreover, it correctly points to a memory access problem in the top two procedures despite their low L1 data-cache miss ratios. These two procedures perform so many memory accesses (almost one out of every two executed instructions accesses memory) that the estimated upper bound on the LCPI contribution is high enough to make memory accesses the most likely bottleneck. PerfExpert's suggested optimizations include vectorization as well as other optimizations that have helped boost the performance (Section II.C.3). In summary, PerfExpert correctly diagnosed the performance bottleneck and suggested several optimizations that have resulted in significant speedup.

B. HOMME

HOMME exhibits near perfect weak scaling. During acceptance testing of Ranger, *HOMME* was run on 16,384 cores with linear speedup. The benchmark version of *HOMME* contains roughly ten procedures that combined represent 90% of the total execution time. PerfExpert correctly identified that about half of these procedures are severely memory bound, with a CPI above four, and illustrated *HOMME*'s poor performance when utilizing more than two cores per chip.

```
total runtime in homme-4x64 is 356.73 seconds
total runtime in homme-16x16 is 555.43 seconds

Suggestions on how to alleviate performance bottlenecks are available at:
http://www.tacc.utexas.edu/perfexpert/

prim_advance_mod_mp_preq_advance_exp_ (runtimes are 86.35s and 159.20s)
-------------------------------------------------------------------------------
performance assessment   great.....good......okay......bad.......problematic
- overall                >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>2222222222222222222
upper bound by category
- data accesses          >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- instruction accesses   >>>>>>>>>>
- floating-point instr   >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>1
- branch instructions    >
- data TLB               >
- instruction TLB        >
```

Fig. 7. Assessment of *HOMME* with 1 and 4 threads/chip

Fig. 7 shows how dramatically performance drops when running the same workload with 16 threads per node instead of just 4 threads per node, and that the single largest problem is data accesses. All the problematic functions have little data reuse but reasonably high cache hit ratios. Because performance drops in these functions when the thread density increases, the primary issue must lie with shared on-chip resources. Because of the small L1 and L2 miss ratios, we assume that the L3 cache capacity and L3 conflict misses are not the issue. The primary issue appears to be DRAM page conflicts. On a Ranger node, only 32 DRAM pages can be open at once, each covering 32 kilobytes of contiguous memory. With 16 threads operating, each thread can access at most two different memory areas simultaneously without severe performance losses. We corrected this problem by applying loop fission so that each loop only processes two arrays. However, because the compiler automatically fused the loops, we had to take the additional step of breaking out each loop into a separate procedure, which results in great speedup despite the call overhead. Applying the loop fission optimization to the preq_robert procedure resulted in a 62% performance increase and much better utilization of four cores. We still have to apply loop fission to other functions in *HOMME*. Note that PerfExpert users do not have to know about DRAM page conflicts. They can just follow PerfExpert's recommendation to fission loops and factor them into separate procedures to improve performance, without necessarily understanding why this optimization helps.

C. LIBMESH

The *EX18* application of *LIBMESH* contains 22 procedures that represent one percent of the total runtime or more but only one procedure that represents over 10% of the runtime. Fig. 8 compares the performance of this procedure before and after we optimized it, thus providing an example of how PerfExpert can be used to track code optimization progress.

```
total runtime in ex18 is 144.78 seconds
total runtime in ex18-cse is 137.91 seconds

Suggestions on how to alleviate performance bottlenecks are available at:
http://www.tacc.utexas.edu/perfexpert/

NavierSystem::element_time_derivative (runtimes are 33.29s and 25.24s)
-------------------------------------------------------------------------------
performance assessment   great.....good......okay......bad.......problematic
- overall                >>>>>>>>>>>>>>>>222
upper bound by category
- data accesses          >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>2
- instruction accesses   >>>>>>>
- floating-point instr   >>>>>>>>>>>>>>>>1111111111
- branch instructions    >
- data TLB               >
- instruction TLB        >
```

Fig. 8. Assessment of *EX18* before and after optimization

The element_time_derivative procedure has somewhat poor floating-point performance and quite poor data access performance. We were able to improve the floating-point performance by factoring out common subexpressions and moving loop invariant code. Based on simple tests, the author of *EX18* assumed that the compiler would do this. However, several of the common subexpressions we found involve C++ templates and most of them involve pointer indirections, which apparently makes the code too complex for the compiler to analyze and perform these optimizations.

The runtimes in Fig. 8 show that these relatively simple optimizations (for a human) made element_time_derivative 32% faster. Because this procedure represents roughly 20% of the total runtime, this node-level code modification yielded an application-wide speedup of 5%. We have not yet tried to perform the other optimizations PerfExpert suggests nor have we attempted to optimize any of the other procedures in *EX18*.

As Fig. 8 highlights, our optimizations substantially reduce the upper LCPI bound of the floating-point instructions (because so many fewer floating-point instructions are executed). However, the overall assessment is worse for the optimized procedure, even though the runtimes clearly show that it executes much faster. The reason is that reducing one bottleneck emphasizes the remaining bottlenecks, in this case the memory accesses. Thus, PerfExpert's assessment correctly reflects that instructions execute more slowly on average in the optimized code (but the optimized code executes a lot fewer instructions, resulting in a speedup).

D. ASSET

Fig. 9 shows the performance assessment of the OpenMP-based *ASSET* application. The top two procedures represent about half of the total runtime. They calculate the flux that is emitted from the volume at a given frequency by integrating intensities along rays pointing inwards starting at the outermost layer of the computational domain. The second procedure, which is called by the first procedure, is a hand-coded exponentiation function that provides a 50% speedup compared to the built-in exp function for a limited argument range. PerfExpert's assessment shows that the second procedure scales perfectly to 16 threads per node and performs well. This part of the calculation is performed in double precision.

The other half of the CPU time is spent in cubic interpolations in 1, 2, or 3 dimensions, which are (mainly) needed to populate a ray with data provided at the grid points of the computational mesh. The interpolation procedure bez3_mono_r4_l2d2_iosg is one of the many single-precision procedures that are hand-tuned for slightly different purposes. It scales poorly because of data accesses that exhaust the processors' memory bandwidth.

```
total runtime in asset_4 is 140.78 seconds
total runtime in asset_16 is 52.25 seconds

Suggestions on how to alleviate performance bottlenecks are available at:
http://www.tacc.utexas.edu/perfexpert/

calc_intens3s_vec_mexp (runtimes are 45.96s and 14.44s)
-------------------------------------------------------------------------------
performance assessment   great.....good......okay......bad.......problematic
- overall               >>>>>>>>>>>>>>2222
upper bound by category
- data accesses         >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- instruction accesses  >>>>>>>>>
- floating-point instr  >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>2
- branch instructions   >
- data TLB              >
- instruction TLB       >

rt_exp_opt5_1024_4 (runtimes are 27.72s and 7.11s)
-------------------------------------------------------------------------------
performance assessment   great.....good......okay......bad.......problematic
- overall               >>>>>>>>>>>>>
upper bound by category
- data accesses         >>>>>>>>>>>>>>>>>>>>>>1
- instruction accesses  >>>>>>>>1
- floating-point instr  >>>>>>>>>>>>1
- branch instructions   >>>>
- data TLB              >
- instruction TLB       >

bez3_mono_r4_l2d2_iosg (runtimes are 21.67s and 9.52s)
-------------------------------------------------------------------------------
performance assessment   great.....good......okay......bad.......problematic
- overall               >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
upper bound by category
- data accesses         >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- instruction accesses  >>>>>>
- floating-point instr  >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- branch instructions   >
- data TLB              >
- instruction TLB       >
```

Fig. 9. Assessment of *ASSET* with 1 and 4 threads/chip

ASSET was developed and heavily optimized by a member of TACC's High-Performance Computing group (Lars Koesterke) before we analyzed it with PerfExpert. For instance, many of the most demanding loops are manually blocked and unrolled. Data are aligned to 128-bit boundaries to enable the use of SSE instructions. Consequently, all performance optimizations that PerfExpert suggests for this code are already included or do not apply.

V. Related Work

Many performance measurement tools exist. We discuss only those that incorporate automated analysis and diagnosis.

The IBM PERCS project is building an automated system targeting the identification and analysis of performance bottlenecks [9] in application codes and providing automated remediation [10] for each bottleneck. The discovery and analysis framework has a control GUI, which allows the user to control the tuning process and presents hotspot and bottleneck information back to the user. The Bottleneck Detection Engine (BDE), which is the core of the framework, utilizes a database of rules to detect bottlenecks in the given application. The BDE compiles, executes and controls modules via a scheduler and feeds the information on bottleneck locations, including metrics associated with the bottlenecks, to the user. It may also suggest how much improvement could be obtained by the optimization of a given bottleneck. Data is collected by both performance estimates derived from static analysis and from execution measurements conducted with the IBM High Performance Computing Toolkit [32]. In addition to suggestions to the user, IBM's tool also supports directly modifying the source code and applying standard transformations through the compiler [3], a feature that we hope to add to PerfExpert in the future. The major differences between our approach and that of the IBM group are the following. 1) We are targeting performance bottlenecks originating in single core, multicore chip, and multi-socket nodes of large-scale clusters, including in communication library code, whereas the IBM project is attempting diagnosis and optimization of both intra-node and inter-node bottlenecks including inter-node communication and load balancing. PerfExpert is focused on making intra-node optimization as automated and simple as possible. We have chosen this narrower target because it enables simpler user interactions and more focused solutions. 2) The user interface of PerfExpert provides a higher degree of automation for bottleneck identification and analysis. 3) The internal use of HPCToolkit allows a wider range of measurement methods spanning sampling, dynamic monitoring, and event tracing. 4) The implementation of PerfExpert is open source and adaptable to composition with a variety of tools.

Acumem AG [1] sells the commercial products ThreadSpotter (multithreaded applications) and SlowSpotter (single-threaded applications), which capture information about data access patterns and offer advice on related losses, specifically latency exposed due to poor locality, competition for bandwidth, and false sharing. SlowSpotter and ThreadSpotter also recommend possible optimizations. While good data access patterns are essential for performance, other things also matter. PerfExpert attempts a comprehensive diagnosis of bottlenecks, targeting not only data locality but also instruction locality, floating-point performance, etc. Acumem's tools do not attempt automated optimizations.

Continuous program optimization (CPO) [7] is another IBM conducted project. CPO provides a unifying framework to support a whole system approach to program optimization that cuts across all layers of the execution stack opening up new optimization opportunities. CPO is a very broad effort combining runtime adaptation through dynamic compilation with diagnosis of hardware/software interactions.

The Performance Engineering Research Institute (PERI) has many performance optimization projects. The project most closely related to PerfExpert is the PERI Autotuning project [1], which combines measurement and search-directed autotuning in a multistep process. It can be viewed as a special

case of an expert system where one flexible solution method is applied to all types of bottlenecks. However, it is unclear whether autotuning by itself can effectively optimize the wide spectrum of bottlenecks that arise when executing complex codes on multi-core chips and multi-socket nodes. Nevertheless, we hope to be able to incorporate methods from this project in a future version of PerfExpert.

The Parallel Performance Wizard [27] has goals similar to PerfExpert. It attempts automatic diagnosis as well as automated optimization. It is based on event trace analysis and requires program instrumentation. Its primary applications have been problems associated with the partitioned global address space (PGAS) programming model, although it applies to other performance bottleneck issues as well.

Paradyn [18], based on Dyninst [5], is a performance measurement tool for parallel and distributed programs. Instrumentation code is inserted into the application and modified at runtime. The instrumentation is controlled by a Performance Consultant module. Its goal is to associate bottlenecks with causes and program parts similar to the diagnostics of our tool.

KOJAK (Kit for Objective Judgment and Knowledge-based Detection of Performance Bottlenecks) [19] is a collaborative research project aiming at the development of a generic automatic performance analysis environment for parallel programs. It includes a set of tools performing program analysis, tracing, and visualization. In terms of analysis, KOJAK provides several options including tree-style hotspot analysis. The user can identify performance bottlenecks by exploring the tree. KOJAK is based on event trace analysis. It requires user interactions in its evaluation process.

Active Harmony [8], [29] is a framework that supports runtime adaptation of algorithms, data distribution, and load balancing. It exports a detailed metric interface to applications, allowing them to access processor, network, and operating system parameters. Applications export tuning options to the system, which can then automatically optimize resource allocation. Measurement and tuning can therefore become first-class objects in the programming model. Programmers can write applications that include ways to adapt computation to observed performance and changing conditions. Active Harmony requires adaptation of the application and is mostly concerned with distributed resource environments.

MAQAO [13] is a performance analysis tool that works at the assembly level. Like PerfExpert, it combines performance counter measurements with static information to generate diagnoses. However, MAQAO derives the static information from the assembly code. It contains a knowledge base of important assembly patterns, which can be associated with hints for possible code optimizations.

One of the earliest tools with similar goals to PerfExpert is Cray's ATExpert [16]. It graphically displays the performance of parallel programs. It also uses an expert system, but it uses it to simulate parallel execution to gain insights into the parallel performance. ATExpert also points the user to specific problem areas in the source code, tries to explain why the problems are occurring, and suggests actions to resolve them.

VI. Conclusions and Future Work

This paper presents and describes PerfExpert, a novel tool that can automatically diagnose core, socket, and node performance bottlenecks in parallel HPC applications at the procedure and loop level. PerfExpert features a simple user interface and a sophisticated new performance analysis metric. We believe simple input and easy-to-understand output are essential for a tool to be useful to the community. PerfExpert's analysis stage combines performance counter measurements with system parameters to compute upper bounds on the LCPI (local CPI) contribution of various instruction categories. The upper bounds instantly eliminate categories that are not performance bottlenecks and can therefore safely be ignored when optimizing the corresponding code section.

For each important procedure and loop, PerfExpert assesses the performance of the supported categories with the LCPI metric and ranks the categories as well as the procedures and loops to help the user focus on the biggest bottlenecks in the most critical code sections. Because HPC application writers are typically domain experts and not performance experts, PerfExpert suggests performance optimizations (with code examples) and compiler switches for each identified bottleneck. We have populated this database of suggestions with code transformation that we have found useful to improve performance during many years of optimizing programs.

We tested PerfExpert on four production codes on the Ranger supercomputer. In all cases, the performance assessment was in agreement with an assessment by performance experts who used other tools. In two cases, PerfExpert's automatic assessment correctly identified a key bottleneck that the application developers were not aware of. We found many of PerfExpert's suggested optimizations to be useful and improve the intra-node as well as the overall performance of HPC applications running on thousands of cores.

In the future, we intend to perform more case studies, especially with applications where the bottleneck is not memory accesses, and to expand the capabilities of PerfExpert by including non-standard performance counters and non-performance-counter-based measurements. We will increase the number of performance categories so that finer-grained optimization recommendations can be made that are more specific and better tailored to each assessed code section. We will continue to grow our optimization and example database and plan to port PerfExpert to other systems. The most challenging goal we have is to extend PerfExpert to automatically implement the suggested solutions for the most common core-, socket-, and node-level performance bottlenecks. In the longer term, we plan to develop separate implementations of PerfExpert for I/O optimization and communication optimization.

Acknowledgment

This project is funded in part by the National Science Foundation under OCI award #0622780. We are grateful to Omar Ghattas, Carsten Burstedde, Georg Stadler, and Lucas Wilcox for providing the *MANGLL* code base and working with us, John Mellor-Crummey, Laksono Adhianto, and Nathan Tallent for their help and support with HPCToolkit, and Chris Simmons and Roy Stogner for providing and helping with *LIBMESH*. The results reported in this paper have been obtained on HPC resources provided by the Texas Advanced Computing Center at the University of Texas at Austin.

RERERENCES

[1] ACUMEM: *http://www.acumem.com/*. April 12, 2010.

[2] D. Bailey, J. Chame, C. Chen, J. Dongarra, M. Hall, J. Hollingsworth, P. Hovland, S. Moore, K. Seymour, J. Shin, A. Tiwari, S. Williams, and H. You. "PERI Auto-Tuning." *Journal of Physics: Conference Series*, 125(1):012089, 2008.

[3] C. Bastoul. "Code generation in the polyhedral model is easier than you think." *Proc. 13th Int. Conference on Parallel Architecture and Compilation Techniques*, pp. 7-16, 2004.

[4] BKDG: http://support.amd.com/us/Processor_TechDocs/ 31116.pdf. Last accesses April 12, 2010.

[5] B. R. Buck and J. K. Hollingsworth. "An API for Runtime Code Patching." *Journal of High Performance Computing Applications*, 14:317-329, 2000.

[6] C. Burstedde, O. Ghattas, M. Gurnis, G. Stadler, E. Tan, T. Tu, L. C. Wilcox, and S. Zhong, "Scalable Adaptive Mantle Convection Simulation on Petascale Supercomputers." *Proc. SC'08*, 2008.

[7] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. "Performance and environment monitoring for continuous program optimization." *IBM J. Res. Dev*., 50(2/3):239–248, 2006.

[8] I.-H. Chung and J. K. Hollingsworth. "Automated Cluster-Based Web Service Performance Tuning." *Proc. 13th IEEE International Symposium on High Performance Distributed Computing*, pp. 36-44. 2004.

[9] I. Chung, G. Cong, D. Klepacki, S. Sbaraglia, S. Seelam, and H-F. Wen. "A Framework for Automated Performance Bottleneck Detection." *13th Int. Workshop on High-Level Parallel Progr. Models and Supportive Environments*. 2008.

[10] G. Cong, I-H. Chung, H. Wen, D. Klepacki, H. Murata, Y. Negishi, and T. Moriyama. "A Holistic Approach towards Automated Performance Analysis and Tuning." *Proc. Euro-Par 2009 (Parallel Processing Lecture Notes in Computer Science 5704, Springer-Verlag)*. 2009.

[11] M. E. Crovella and T. J. LeBlanc. "Parallel Performance Prediction using Lost Cycles Analysis." *Supercomputing Conference*, pp. 600-609. 1994.

[12] J. Diamond, B. D. Kim, M. Burtscher, S. Keckler, and K. Pingali. "Multicore Optimization for Ranger." *2009 TeraGrid Conference*. 2009.

[13] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W. Jalby. "Exploring Application Performance: a New Tool for a Static/Dynamic Approach." *The Sixth Los Alamos Computer Science Institute Symposium*. 2005.

[14] HPCToolkit: *http://www.hpctoolkit.org/*. April 12, 2010.

[15] B. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. "libMesh: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations." *Engineering with Computers*, 22(3/4):237-254. 2006.

[16] J. Kohn and W. Wiliams. "ATExpert." *Journal of Parallel and Distributed Computing*, 18:2, pp. 205-222. 1993.

[17] LIBMESH: http://libmesh.sourceforge.net/. April 12, 2010.

[18] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. "The Paradyn Parallel Performance Measurement Tool." *IEEE Computer*, 28:37-46. 1995.

[19] B. Mohr and F. Wolf. "KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications." *Proc. International Conf. on Parallel and Distributed Computing*. 2003.

[20] Open|SpeedShop: *http://www.openspeedshop.org/wp/*. Last accessed April 12, 2010.

[21] PAPI: http://icl.cs.utk.edu/papi/. April 12, 2010.

[22] PIN: *http://www.pintool.org/*. Last accessed April 12, 2010.

[23] L. M. Polvani, R. K. Scott, and S. J. Thomas, "Numerically Converged Solutions of the Global Primitive Equations for Testing the Dynamical Core of Atmospheric GCMs." *American Meteorological Society*, 132(11):2539-2552. 2004.

[24] Ranger: http://www.tacc.utexas.edu/resources/hpc/# constellation. Last accessed April 12, 2010.

[25] M. Schulz and B. R. de Supinski. "Practical Differential Profiling." *Euro-Par Conference*, pp. 97-106. 2007.

[26] S. Shende and A. Malony. "The Tau Parallel Performance System." *International Journal of High Performance Computing Applications*, 20(2): 287-311.

[27] H-H. Su, M. Billingsley, and A. D. George. "Parallel Performance Wizard: A Performance Analysis Tool for Partitioned Global-Address-Space Programming." *9th IEEE International Workshop on Parallel & Distributed Scientific and Engineering Computing*. 2008.

[28] N. R. Tallent, J. M. Mellor-Crummey, L. Adhianto, M.W. Fagan, and M. Krentel. "HPCToolkit: performance tools for scientific computing." *Journal of Physics: Conference Series*, 125. 2008.

[29] C. Tapus, I.-H. Chung, and J. K. Hollingsworth. "Active harmony: towards automated performance tuning." *Proc. ACM/IEEE Conference on Supercomputing*, pp. 1–11. 2002.

[30] Tau: http://www.cs.uoregon.edu/research/tau/home.php. Last accessed April 12, 2010.

[31] S. J. Thomas and R. D. Loft, "The NCAR Spectral Element Climate Dynamical Core: Semi-Implicit Eulerian Formulation." *Journal of Scientific Computing*, 25(1/2). 2005.

[32] H. Wen, S. Sbaraglia, S. Seelam, I. Chung, G. Cong, and D. Klepacki. "A productivity centered tools framework for application performance tuning." *Proc. Fourth International Conference on the Quantitative Evaluation of Systems*, pp. 273-274. 2007.