Generating Performance Bounds from Source Code

Sri Hari Krishna Narayanan, Boyana Norris, and Paul D. Hovland
Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, IL 60439
[snarayan,norris,hovland]@mcs.anl.gov

Abstract

Understanding and tuning the performance of complex applications on modern hardware are challenging tasks, requiring understanding of the algorithms, implementation, compiler optimizations, and underlying architecture. Many tools exist for measuring and analyzing the runtime performance of applications. Obtaining sufficiently detailed performance data and comparing it with the peak performance of an architecture are one path to understanding the behavior of a particular algorithm implementation. A complementary approach relies on the analysis of the source code itself, coupling it with a simplified architecture description to arrive at performance estimates that can provide a more meaningful upper bound than the peak hardware performance.

We present a tool for estimating upper performance bounds of C/C++ applications through static compiler analysis. It generates parameterized expressions for different types of memory accesses and integer and floating-point computations. We then incorporate architectural parameters to estimate upper bounds on the performance of an application on a particular system. We present validation results for several codes on two architectures.

1 Introduction

Developing high-performance applications and optimizing their performance require a thorough understanding of the algorithms, their implementation, compilers and external libraries, and the underlying hardware. Often, the performance achieved is a small fraction of peak, and substantial time and effort are invested in trying to improve that fraction. As others have noted over the years (e.g., Gropp et al. [4]), theoretical peak performance is not a good upper bound estimate for the majority of applications. Furthermore, different computations fail to achieve good performance for different reasons: some are memory intensive (e.g., sparse linear algebra), while others are operation intensive (such as dense linear algebra). Determining whether a code fragment is memory- or CPU-bound by manual inspection of the code is a nontrivial task and is only practical for relatively small and self-contained kernels. Thus, such determination is usually done by using postmortem analysis of detailed performance data that includes counts of memory accesses, cache misses, and floating-point operations. A number of performance tools enable the collection of this data, but the process is still involved and must be customized for each platform. As an alternative to hardware

counter-based analysis, we have developed a tool for computing more realistic (than theoretical peak) upper performance bounds based on source analysis and transformation. We refer to this tool as PBound in the remainder of this paper. It has been built and is available at : `http://trac.mcs.anl.gov/projects/performance/wiki/Pbound` for use.

Theoretical peak performance is a frequently used method for comparing architectures and evaluating the performance of applications. The most popular peak performance values are based purely on the clock rate and the number of floating-point units available in the system. Less frequently, the latency and bandwidth of data transfers between different levels of memory and the processor are also taken into consideration. Peak performance is not realistically achievable even by heavily optimized synthetic benchmarks, and is thus not a good upper bound for guiding labor-intensive performance tuning of scientific applications.

The usual approach to generating more accurate estimates the efficiency of a given code is to execute it on a given architecture (or a simulator) and collect performance information ranging from wall-clock time to low-level hardware performance counters. The performance counter data can then be used to build a profile of the execution which can be used to pinpoint different bottlenecks, such as memory- or computationally-intensive regions. The disadvantage of this approach is that performing these studies is a nontrivial task and requires familiarity with and availability of performance tools on the architectures of interest. Furthermore, depending on the tool, multiple runs for the same inputs may be necessary, some with substantial profiling overheads resulting in heavy, non-production, resource use for the performance data gathering process.

PBound generates upper performance bounds for C/C++ applications through static analysis of the source code. It generates parameterized expressions for different types of memory accesses and computations. Combined with architectural information, the upper bounds on the performance of an application on a particular platform can be estimated. Application designers can test observed performance against these bounds to calculate the efficiency of their implementation where efficiency is defined to be the ratio of achieved performance to the performance bound. Different aspects of performance can be considered, including memory bandwidth, floating-point operations per second (FLOP/s) or wall-clock time. A more realistic estimate of ideal efficiency can save a lot of wasted optimization effort. For example, if a code is achieving 7% of the theoretical peak FLOP/s rate, and the performance bound is equivalent to 10% of the peak FLOP/s rate (e.g., if the implementation is memory-intensive), it would not be possible to optimize the existing algorithm to achieve anything more than 10% and substantially different algorithms with lower memory bandwidth requirements should be considered if possible.

PBound can be used to study the maximum achievable performance of an application on a given architecture without having to either run the application on that architecture or a simulator. Therefore, it can be used for rapid exploration of the application performance space for different architectures or architectural configurations. Static analysis, however, can at best conservatively estimate the dynamic behavior of some codes. Recursion, runtime parameters and dynamic memory allocation can complicate the source-based counts of memory accesses and arithmetic operations. Furthermore, competition for resources between applications cannot be modeled. However, we believe that the bounds estimates can be invaluable in critical kernels organized as a series of nested loops, which are at the heart of many scientific applications.

(a) Original code:

```
void axpy4(int n, double *y, double a1, double *x1,
           double a2, double *x2, double a3, double *x3,
           double a4, double *x4)
{
    register int i;
    for (i=0; i<=n-1; i++)
        y[i]=y[i]+a1*x1[i]+a2*x2[i]+a3*x3[i]+a4*x4[i];
}
```

(b) Generated code:

```
#include "pbound_list.h"
void axpy4(int n, double *y, double a1, double *x1, double a2,
                    double *x2, double a3, double *x3,
                    double a4, double *x4)
{
#ifdef pbound_log
  pboundLogInsert("axpy.c@6@5",8,0,40 * ((n - 1) + 1) + 32,
     8 * ((n - 1) + 1),3 * ((n - 1) + 1) + 1,4 * ((n - 1) + 1));
#endif
}
```

Figure 1: Example code (a) and generated performance bounds (b).

1.1 Example

We illustrate the use of PBound with a simple example code implementing a generalized *axpy* computation of the form $y = y + a_1x_1 + \cdots + a_nx_n$, where $a_1, \ldots, a_n$ are scalars and $y, x_1, \ldots, x_n$ are one-dimensional arrays. Figure 1(a) shows the source code for $n = 4$. The results are generated by first giving the code as input to the tool, which generates : (1) a log file containing memory access and arithmetic operation counts for each loop and function and (2) a new version of the original code (shown in Figure 1(b)). The generated code is a *slice* of the original computation and contains only the statements that are necessary for computing the bounds. The parameterized counts in either the log file or the transformed code can be evaluated either manually (by substituting appropriate values for all variables) or by simply compiling the generated source code and executing it in the same manner as the original application. The generated code contains a call to the `pboundLogInsert` function with parametrized expressions for the number of loads, stores, and arithmetic operations, with separate expressions for integer and floating-point values. A simple runtime library implements `pboundLogInsert` to keep track and output the loads, stores, and arithmetic operations. Since the generated code contains only a few simple expressions, it normally takes much less time than the original application. Thus, in many cases one can compute the performance bounds for a large parallel application on much smaller resources such as a laptop.

Table 1 shows the different expressions that are generated. The expressions contain the parameter n, which is the upper bound of the loop in the example code. After the code is executed, the arguments of the function call are computed by using the runtime parameter n. For example, if $n = 2,000,000$, then there are 8,000,000 floating-point operations performed, requiring 10,000,000 loads, indicating that the application is memory bound on most modern architectures.

Table 1: Performance metric expressions for example code in Fig. 1.

Metric	Expression
Integer loads	2
Integer stores	0
Floating-point loads	5 * ((n - 1) + 1) + 1 + 4
Floating-point stores	1 * ((n - 1) + 1)
Integer operations	3 * ((n - 1) + 1) + 1
Floating-point operations	8 * ((n - 1) + 1)

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents our implementation approach. Section 4 contains experimental validation results. Section 5 concludes with a discussion of future work.

2 Related Work

The paper by Gropp et al. [4] provided the motivation for developing a tool for computing performance bounds automatically. The manual process described in that paper relies on the definition of a "general" form of the computation, which is then analyzed statement by statement to count the number of loads, stores, and arithmetic operations for integer and floating-point values. PBound does not require the creation of a separate high-level representation; instead it analyzes the source code directly, as described in Section 3.

Vuduc et al. [17] manually derive upper bounds on the performance of a sparse A^TAx kernel. Sparse computations usually contain indirect memory accesses and thus present a challenge to optimizing compilers, as well as to source analysis tools such PBound. In Section 3 we describe our current handling of computations containing indirect addresses.

Williams et al. [18] introduce the Roofline performance model approach, in which an upper bound on performance of a kernel is set depending on the kernel's operational intensity. Operational intensity is defined as "operations per byte of DRAM traffic". While the ideas and implementation of PBound were conceived many years ago, they are similar in spirit to the Roofline model. As far as we know, PBound is the first tool that relies on source code analysis to partly automate the generation of performance models of applications.

A wide selection of performance tools, including PAPI [6], TAU [15], HPCToolkit [5], Scalasca [2], Kojak [19], PerfSuite [7], gprof [3], CATCH [1], and Active Harmony [16] can be used to collect, process, analyze, and visualize performance data. Because our focus is on source-based performance modeling, we do not enumerate all the runtime modeling approaches and tools available.

3 Implementation

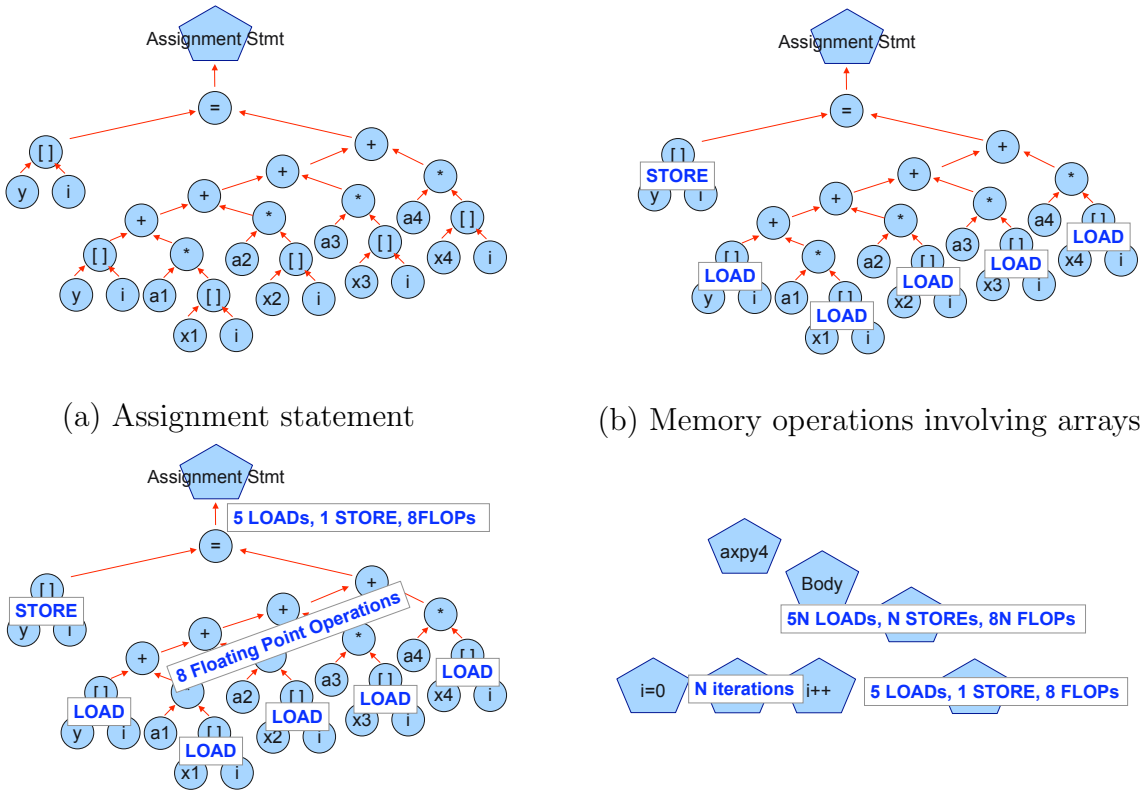
Our implementation is based on the ROSE compiler framework, which provides parsing and abstract syntax tree (AST) generation and traversals for C, C++, and Fortran [13,14]. ROSE can be used to build source-to-source transformation tools that support various compiler analyses and optimizations. For parsing C and C++, ROSE relies on the commercial EDG parser. The EDG parse trees are converted into a higher-level, ROSE-specific SAGE abstract syntax tree (AST). The leaf nodes of the AST include variable references or constants, while the internal nodes include various statements (e.g., declarations, assignment statements), function definitions, or expressions. ROSE provides mechanisms for manipulating the AST through tree traversals while ensuring syntactic correctness. The AST can be modified through interfaces provided by ROSE. Nodes in the AST can be decorated with either inherited or synthesized attributes. ROSE can also be used with external analysis tools.

PBound's implementation uses the ROSE AST traversal mechanism to count Integer and floating-point memory and arithmetic operations. Optionally, PBound determines whether instructions can be fused into one instruction such as fused multiply-add. Loops are optionally vectorized to the extent supported by the architecture. Architectural information, such as the presence of fused instructions and vector length for single-instruction multiple-data (SIMD) operations, is provided via a configuration file (discussed in more detail in Section 3.1). The generated parameterized expressions are stored in a log file and also (optionally) inserted into the generated source code. As briefly mentioned in Section 1.1, PBound optionally performs *slicing* when generating the instrumented version of the code to make the bounds computation as easy and as cheap as possible. When slicing is enabled, the generated code contains only those statements whose results are used in the generated performance metrics. The current analysis for determining the dependencies between the metric expressions and statements is limited to simple cases, but we plan to extend it with context-sensitive more versatile dependence analysis. The generated metric expressions can be inserted either as comments or as calls to a function that records the values of current memory access and operation counts. Currently, C and C++ input codes are supported.

Figure 2(a) shows a highly simplified AST for the assignment statement contained in the loop body in the example code in Figure 1(a). Our tool traverses the AST in a *TopDown-BottomUp* manner, which means that first traversal proceeds from root to leaves, then from the leaves up. As the traversal heads downward, context information in the form of inherited attributes are passed down the tree. This helps identify regions of interest such as function definitions and avoids the traversal of system header files.

Next, the AST is traversed from the leaves to the root. Only the portions of interest that are identified in the preceeding top-down pass are processed by our tool. The following actions take place at each type of node (we highlight the interesting actions performed at the relevant subset of the nodes):

- **Variable references.** A variable reference is in effect a memory operation. Integer and floating-point memory operations are counted separately. If a variable is a scalar, we assume that it can be held in the registers and repeated accesses to it do not result in multiple loads. For now, the tool does not perform register allocation to determine whether the number of scalars in a function exceeds the number of registers. Accesses



(a) Assignment statement

(b) Memory operations involving arrays

(c) Floating-point operations

(d) For loops

Figure 2: Simplified AST fragments corresponding to the example code in Figure 1.

to arrays (and point array references) are treated as loads from memory if the load is on the right-hand side (RHS) of an assignment statement or a store if it is on the left-hand side (LHS). Figure 2(b) shows the loads and stores that are identified for the array memory operations in the assignment statement. The figure ignores the scalars for the sake of clarity. Currently, array index calculations are treated as free; in other words, they are not counted. The effects of doing so are more pronounced when an array is indexed by the contents of another array, for example, $A[B[i]]$. In such cases, only the load or store of the top level array A is counted and not the load of array B. PBound will be modified to account for both memory accesses.

- **Binary expression.** At each binary expression, the memory accesses from the RHS and LHS of the expression are summed up. Depending on the data types of the RHS and LHS, an appropriate number of integer or floating-point operations are considered necessary to perform the operation. Because the binary operation may be a parent in an expression that has other binary operations, the integer and floating-point operations required to calculate the RHS and LHS are added to the operations required for the current binary operation. Figure 2(c) shows that 8 floating-point operations are performed in the AST fragment. At present, all binary operations are assumed to take the same number of cycles. We are planning extensions to allow more accu-

rate cost estimates for different operations that normally have higher latencies, such as floating-point division, square root and other standard mathematical functions.

- **Assignment statement.** At each assignment statement, all the memory and computational operations of the child nodes are summed and passed on to the statement's parent. The statement itself does not result in any computation or memory operations. The example assignment statement in Figure 2 contains a total of 5 load operations, 1 store operation, and 8 floating-point operations.

- **Basic block.** The memory and computational operations of all the child nodes (usually assignment statements) are summed and passed on to the basic block's parent. The basic block itself does not contribute any computation or memory or arithmetic operations. In the example in Figure 2 , the for loop body (which is a basic block) contains only the assignment statement, resulting in a total of 5 load operations, 1 store, and 8 floating-point operations.

- **For loop statement.** At each loop statement, we first evaluate whether the iteration count of the loop can be determined statically. Currently, we handle only cases in which the loop iterates from 0 to some constant N with an increment (or decrement) of 1. In the future, this requirement will be relaxed through the use of the Omega Library, which solves counting solutions expressed using Presburger Arithmetic [12]. If the loop is *summable*, then the computational and memory operation count of the body is *multiplied* by the iteration count value to generate the computational and memory operation measurements for the entire for loop. If the loop iteration count or the body's computational and memory operation requirements are expressions and not integers, a corresponding multiplication expression is created for the loop. Figure 2(d) shows the expressions that are created. The expressions are then saved for insertion into the argument list of the `pboundLogInsert` function call.

3.1 Configuration File

In order to achieve good accuracy on a variety of architectures, PBound can take into account a configuration file that contains information about different pertinent hardware capabilities. Table 2 shows some of the PBound configuration options and their default values. While this list provides an adequate starting point, we are continuously exploring new architecture-specific options that would ensure that PBound produces useful and accurate upper bounds.

Table 2: Configuration parameters.

Category	Parameter	Default Value (Type)
Vector operations	vector_ops	true (boolean)
	vector_length	2 (int)
Fused operations	fp_mul_add	true (boolean)
	fp_mul_sub	true (boolean)
	fp_div_add	true (boolean)
	fp_div_sub	true (boolean)

The first class of options is used to account for potential SIMD operations. If SIMD operations are enabled, the processing of the for loop is modified, resulting in different expressions for memory and computational operation counts. At present, PBound considers a loop to be simdizable if the following conditions are met.

1. The types of the different operations involved are the same.
2. The alignment of the values being operated on are the same in the vector register.
3. The loop increment (or decrement) is 1.

If the loop is determined to be simdizable, the number of computational and memory operations is divided by the vector length.

If fused operations are enabled, we implement the following greedy approach to estimating the number of potential fused operations. When a binary operation is encountered during AST traversal, it is inserted into a set. If the current binary operation can be fused with an operation already present in the set, then the two binary operations are treated as fused, and their computational requirements are not summed. Instead, the appropriate fused operation's computational requirements are used.

4 Experimental Validation

We conducted experiments using three applications (summarized in Figure 3): the triad operation from the Stream [8] benchmark (with vector size $2,000,000$); a generalized AXPY operation shown earlier in Section 1.1 (with vector size $22,612$); and a sparse matrix-vector multiplication kernel (using compressed sparse row storage and an input matrix from the MatrixMarket repository [10], with 90,449 rows and columns and 1,921,955 nonzeros).

Table 3: Description of validation benchmarks.

Benchmark	Description
Stream triad	$z_i \leftarrow x_i + \alpha y_i$
Generalized AXPY	$y_i \leftarrow y_i + \alpha_1 x_{1_i} + \alpha^2 x_{2_i} + \cdots + \alpha^n x_{n_i}$
Sparse matrix-vector product	$\forall_{A_{i,j}} \neq 0 : y_i \leftarrow y_i + A_{i,j} \cdot x_j$

To validate the statically generated operation counts, we compared them with experiments performed on a multicore Intel Xeon workstation and a Blue Gene/P.

To show the effects of configuration options on the operation of PBound, we created four configurations: basic, SIMD-enabled, fused-enabled, and SIMD- and fused-enabled. The size of the SIMD vector was 2 (which is the correct value for both the Xeon and the BG/P for computations involving doubles). Figure 4 lists the absolute values of the loads, stores, and floating -point operations performed by the benchmarks under these configurations.

Table 4: PBound output for different configuration parameters

Kernel	Configuration	Loads	Stores	FLOP
axpy	Basic	113,066	22,612	180,896
	SIMD	56,533	11,306	90,448
	Fused	113,066	22,612	90,448
	SIMD+Fused	56,533	11,306	45,224
stream	Basic	4,000,002	2,000,000	4,000,000
	SIMD	2,000,001	1,000,000	2,000,000
	Fused	4,000,002	2,000,000	2,000,000
	SIMD+Fused	2,000,001	1,000,000	1,000,000
spmv	Basic	5,765,867	180,898	4,024,808
	SIMD	2,882,934	90,449	2,012,404
	Fused	5,765,867	180,898	2,012,404
	SIMD+Fused	2,882,934	90,449	1,006,202

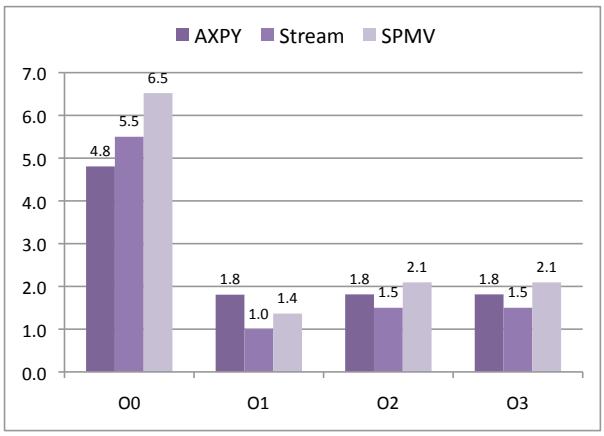
4.1 Intel Xeon Results

The Intel workstation has dual quad-core E5462 Xeon processors (eight cores in total) clocked at 2.8 GHz (1600 MHz FSB) with 32 KB L1 cache, 12 MB of L2 cache (6 MB shared per core pair), and 2 GB of DDR2 FBDIMM RAM, running Linux kernel version 2.6.25 (x86-64), and using Intel compilers version 10.1. We validated the statically generated performance bounds by using TAU (Tuning and Analysis Utilities) performance system, which is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, Java, and Python [15]. We profiled the benchmark applications by (1) instrumenting the portions of interest, (2) compiling the instrumented application, (3) executing the compiled application with specified performance counters, and (4) examining the performance counters. TAU relies on the PAPI [6] interface to performance counters, including native hardware counters on Xeon. Table 5 summarizes the counters we used to profile the benchmarks. Two runs were needed to obtain the values for all the counters: one for the standard PAPI counters and one for the native counters.

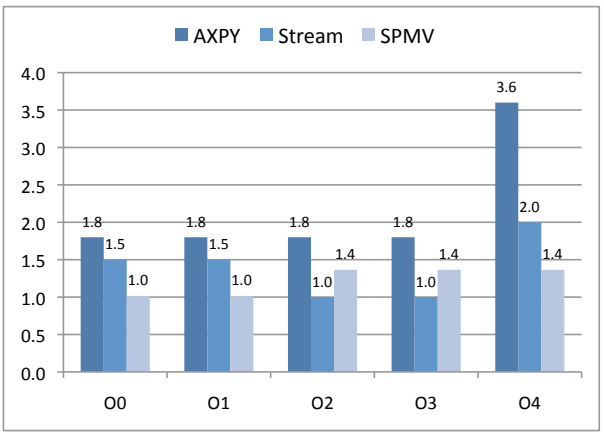
Table 5: Description of the counters used on the Intel Xeon

Counters	Description
UNHALTED_CORE_CYCLES	Core cyles
TOT_CYC	Core cyles
FP_INS	Floating-point inst. retired
INST_RETIRED:LOADS	Load instructions retired
INST_RETIRED:STORES	Store instructions retired
UOPS_RETIRED:MACRO_FUSED	Fused instructions retired
SIMD_COMP_INST_RETIRED:PACKED_DOUBLE	SIMD instructions retired

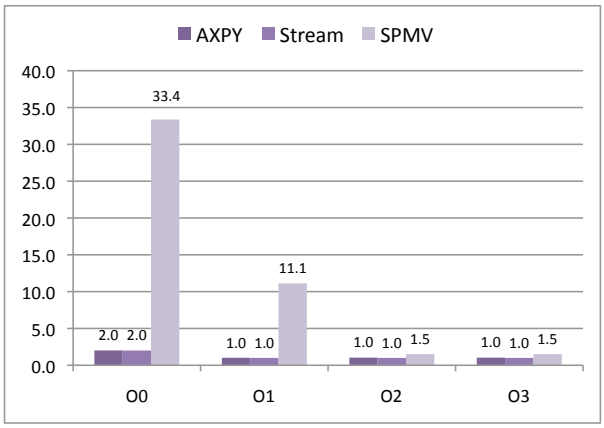
Figure 3(a) shows the ratio of actual loads performed by each benchmark for different optimization levels w.r.t. the static prediction. Similarly, Figure 3(c) shows the ratio of



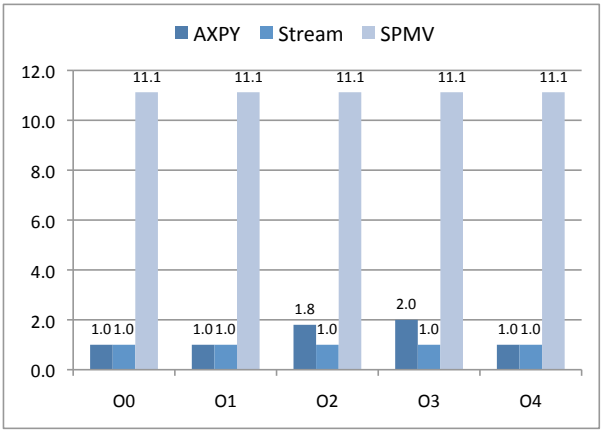
(a) Xeon loads



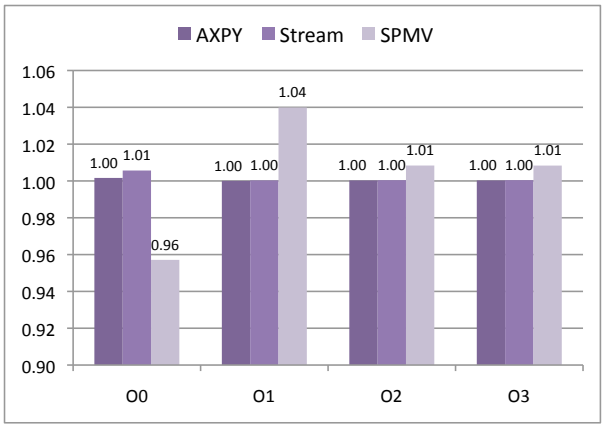
(b) Blue Gene/P loads.



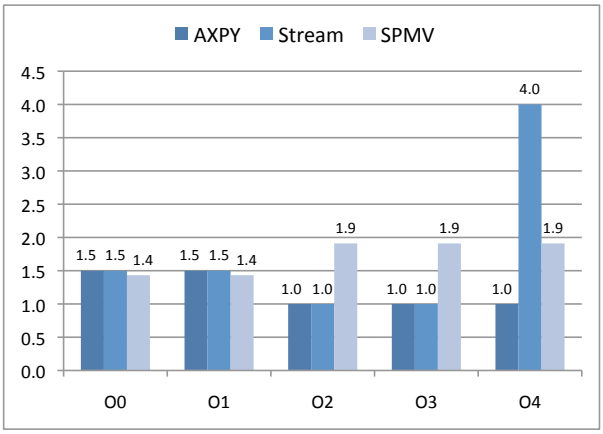
(c) Xeon stores



(d) Blue Gene/P stores



(e) Xeon FLOPs



(f) Blue Gene/P FLOPs

Figure 3: Validation results on an Intel Xeon workstation and Blue Gene/P.

stores performed by each benchmark for different optimization levels w.r.t. the predicted value. Figure 3(e) shows the equivalent ratio for the number of floating-point operations. The bounds values in the comparison with codes compiled with the "-O0" and "-O1" options were computed by setting the SIMD configuration options to false because the Intel compiler does not perform simdization at those levels. The bounds values in the comparison with codes compiled with the "-O2" and "-O3" options were computed by setting the SIMD configuration options to true and the vector length option to 2 (fused operations were set to false in all cases because they are not available on the Xeon). A ratio of one indicates a perfect match between the predicted and measured values. A value above one indicates that expected compiler optimizations (e.g., SIMD code generation or blocked stores) were not performed and that the generated code was not as efficient as expected. A ratio of less than one means that the predicted value is not an accurate ideal estimate and thus cannot be used reliably in computing an upper bound to performance (since the execution data was better than the predicted "ideal" value). There was only one such value (for the sparse matrix-vector product and runtime data based on compiling with "-O0"), which we must investigate further in order to determine which of the assumptions in our counting approach caused the observed discrepancy. For most applications, however, the highest level of optimization is of greatest interest. The results for "-O3" indicate that actual loads and stores were within a factor of two of the ideal bound. The floating-point operation counts were accurate to within a fraction of a percent.

Predicting wallclock time. In addition to raw counts validation, we computed wallclock time estimates based on several simple performance models that integrate the static predictions with hardware characteristics for different memory access patterns. We obtained hardware parameters such as bandwidth to various levels of memory by running LMbench [9] on the Xeon. We then computed the expected (predicted) time based on peak FLOP counts ($4FLOPS/cycle \times 0.3573ns/cycle \times 1^{-9}$ seconds). The first set of bars in Figure 4 show the ratio of predicted vs. measured wall-clock time based on peak FLOP counts using code compiled with "-O3". Clearly, the measured times are much larger than the predicted times.

Next, we evaluated several simple models based on adding the floating-point operation costs based on the peak FLOP rate to the time required to load all values from different levels of cache or memory (assuming no prefetching and overlapping stores). Specifically, T1 is the time for performing the computation and loading values from L1, T2 is the time for performing the computation and loading values from L2, and T3 is the time for performing the computation and loading values from memory. On a single core, the LMbench-measured read bandwidth from memory is 3585 MB/s, so we have $3585/2800 = 1.28$ bytes per cycle. The theoretical peak is 4 floating-point operations per cycle. Thus, any computation that requires more than $1.28/4 = 0.32$ bytes per FLOP is memory bound. The bytes per FLOP values (based on PBound-generated expressions) for the main loop in each of the three applications are : 0.625 for AXPY, 1 for Stream, and 1.5 for SPMV. Thus, they are all memory bound but to a different degree. This is reflected in the validation results in Figure 4: the most accurate AXPY model is T3 (loads from memory), indicating little reuse in either L1 or L2; the most accurate Stream model is T2, indicating that data is successfully being prefetched into L2; and the most accurate prediction for SPMV is T1, indicating a significant

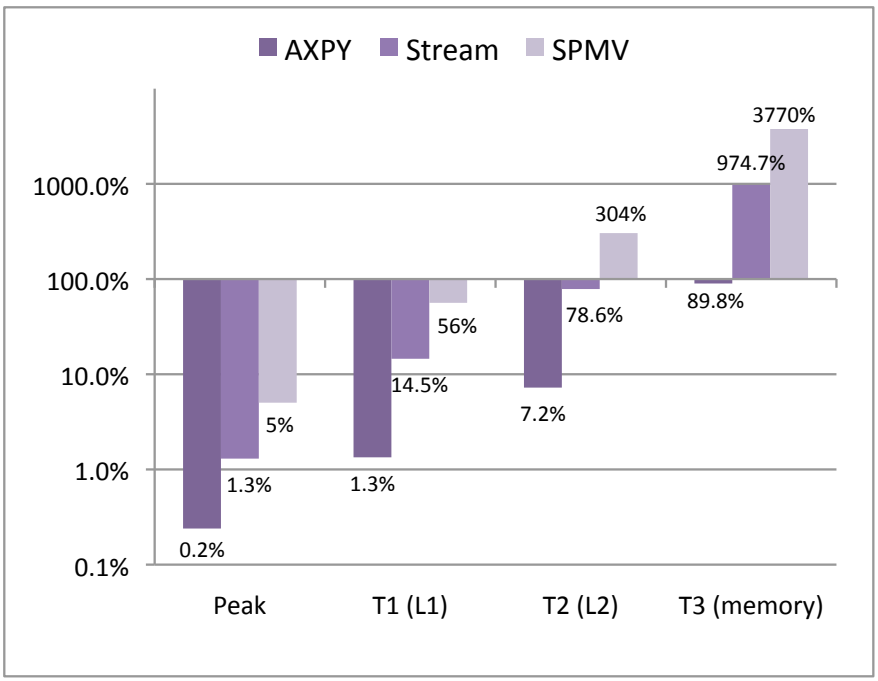


Figure 4: Efficiency estimates based on peak performance values and several statically computed predictions (for a fixed problem size): ratio of predicted time vs. measured time.

proportion of L1 reads.

Our simple models do not capture potential reuse or the prefetching capabilities of the hardware. To enable construction of more realistic performance models automatically, we must incorporate dataflow analyses and implement additional PBound configuration parameters for taking prefetching into account.

Example bound computation. In addition to producing models for predicting wallclock time, the PBound-generated expressions can be used to compute more realistic bounds on the achievable performance. For example, for memory-limited computations, the peak GFLOP/s rate can be expressed as **UB** $GFLOP/s =$ **N** $FLOPs/Byte *$ **Pm** $Bytes/sec * 1.e-9$, where **N** is the ratio of the number of FLOPs to the number of bytes for each loop body, and **Pm** is the peak memory reads bandwidth, which on the Xeon is 3.585GB/s for one core (for simplicity, we do not consider the cost of stores). Table 6 shows the best achievable GFLOP/s rate computed with the PBound-based static information and compares it to the theoretical peak performance of a single core (11.2 GFLOP/s), designated by **Pc**.

4.2 Blue Gene/P Results

Each compute node of the Blue Gene/P is equipped with four cores, each one a 850 MHz IBM PowerPC 450 processors with a dual floating-point unit. While L1 cache (32 KB) and L2 cache (4 MB) are private for each processor, L3 cache (8 MB) is shared on each Blue Gene/P node. The total memory per node is 2 GB. The operating system on Blue Gene/P is based on Linux 2.6.16 (ppc64). We used the IBM XL C/C++ V9.0 compiler. On the

Table 6: Upper bound on the achievable GFLOP/s rate.

Application	**N** FLOPs/Byte	**UB** GFLOP/s	**UB / Pc**
AXPY	1.60	6.01	53.70%
Stream	1.00	3.76	33.56%
SPMV	1.67	2.51	22.38%

Table 7: Counters used on the Blue Gene/P

BGP_PU0_FPU_QUADWORD_LOADS	BGP_PU0_FPU_ADD_SUB_1
BGP_PU0_FPU_QUADWORD_STORES	BGP_PU0_FPU_MULT_1
BGP_PU0_FPU_OTHER_LOADS	BGP_PU0_FPU_DIV_1
BGP_PU0_FPU_OTHER_STORES	BGP_PU0_FPU_FMA_2
BGP_PU0_FPU_ADD_SUB_2	BGP_PU0_FPU_MULT_2
BGP_PU0_FPU_FMA_4	

Blue Gene/P, we used the Hardware Performance Monitor (HPM) toolkit [11] to access the values of the native hardware counters.[1] First, we manually instrumented the input code with calls to the library version of the HPM toolkit. Then, we executed each application and examined the values of the counters summarized in Table 7. Some of the counters reflect the loads and stores performed, while the rest reflect various floating-point operations.

Figure 3(b) shows the ratio of loads performed by each benchmark for different optimization levels w.r.t. the predicted count. Similarly, Figure 3(d) shows the ratio of stores performed by each benchmark for different optimization levels w.r.t. the predicted metric. Figure 3(f) shows the ratio of floating-point operations w.r.t. the statically predicted number of floating-point operations. In the comparison for codes compiled with "-O4", we set the SIMD and fused options to true in computing the static counts, in order to reflect a fully optimized code. As the results indicate, for some of the codes (AXPY and Stream), the compiler did not generate SIMD or fused floating-point instructions. For all other optimization levels, these PBound configuration options were set to false. Equivalent to the Xeon experiments, ratios close to one indicate accurate predictions. For the Blue Gene/P the agreement between prediction and measurement was not as close as for the Xeon, but in all cases the prediction was better than the measurement, implying that the static predictions can reliably produce upper performance bounds.

[1]TAU is also available on the BG/P, but most standard PAPI counters are not supported, and we deemed HPM easier to use because it does not require setting any environment variables, while allowing 256 different counters to be measured simultaneously.

5 Conclusions and Future Work

We have described a new tool for automatically generating parameterized memory and computational metrics from C/C++ source code. Our initial validation with three small applications shows that the static operation counts can provide useful insight into the characteristics of performance-critical numerical kernels.

We consider the current PBound implementation a good first step toward automated performance bounds modeling. Much work remains, some of which we briefly summarize here. We plan to add support for Fortran in the near future by leveraging the recent addition of Fortran parsing capabilities in ROSE. We will continue expanding the list of possible configuration options to further improve the accuracy of the operation estimates. We will also incorporate more rigorous analysis that can help create more accurate memory performance models, for example, by measuring reuse distance and incorporating it into the bounds calculations. So far we have focused on single-core performance; the next step is to incorporate multicore scenarios by leveraging existing polyhedral analysis approaches. We are also working on enabling less conservative slicing and thus making bounds computations cheaper. In the long term, we plan to investigate the generation of *parallel* performance bounds for MPI codes by employing new context-sensitive MPI analyses.

Acknowledgments. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] L. DeRose and F. Wolf. CATCH: A call-graph based automatic tool for capture of hardware performance metrics for MPI and OpenMP applications. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 167–176, London, UK, 2002. Springer-Verlag.

[2] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. Scalable parallel trace-based performance analysis. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (2006)*, pages 303–312, Bonn, Germany, 2006.

[3] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, New York, 1982. ACM Press.

[4] W. D. Gropp, D. K. Kaushik, D. Keyes, and B. F. Smith. Toward realistic performance bounds for implicit CFD codes. In D. Keyes, A. Ecer, J. Periaux, N. Satofuka, and P. Fox, editors, *Proceedings of Parallel CFD '99*, pages 233–240. Elsevier Scientific Publishing Company, 1999.

[5] HPCToolkit Web Page. `http://hpctoolkit.org/`.

[6] Innovating Computing Laboratory, University of Tennessee. Performance application programming interface. http://icl.cs.utk.edu/papi/, 2004.

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

[8] J. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/, 2006.

[9] L. McVoy and C. Staelin. LMbench - Tools for performance analysis. http://www.bitmover.com/lmbench/.

[10] NIST. Matrixmarket web page. http://math.nist.gov/MatrixMarket/.

[11] C. Pospiech. *Hardware Performance Monitor (HPM) Toolkit Users Guide*. International Business Machines Corp., 3.2.2 edition, June 2008. http://domino.research.ibm.com/comm/research_projects.nsf/pages/hpct.hardwareperf.html/\$FILE/HPM_ug.pdf.

[12] W. Pugh. The Omega Project web page. http://www.cs.umd.edu/projects/omega/.

[13] D. Quinlan. ROSE web page. http://rosecompiler.org.

[14] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, Aug. 2003.

[15] S. S. Shende and A. D. Malony. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[16] C. Tapus, I.-H. Chung, and J. K. Hollingsworth. Active Harmony: Towards automated performance tuning. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–11, Los Alamitos, CA, 2002. IEEE Computer Society Press.

[17] R. Vuduc, A. Gyulassy, J. Demmel, and K. Yelick. Memory hierarchy optimizations and performance bounds for sparse A^TAx. In *ICCS 2003: Workshop on Parallel Linear Algebra*, 2003.

[18] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[19] F. Wolf and B. Mohr. Automatic performance analysis of hybrid mpi/openmp applications. *Journal of Systems Architecture: The EUROMICRO Journal*, 49(10-11):421–439, 2003.