The Bench++ Benchmark Suite

Measuring C++ code performance

By Joseph M. Orost

Joe is district manager for AT&T Lab Software Technology Services. He can be contacted at joseph.orost@att.com or http://www.research.att.com/~orost/.

Investigation into optimizing compiler design requires a suite of measurement programs to determine what kind of job the compiler is doing in translating the source program into target machine code. My initial search of the Internet for such a suite written in C++ failed to find anything significant. Therefore, I decided to create my own set of benchmarks for C++, thus producing "Bench++."

Unlike most other benchmark suites, Bench++ is designed to measure the performance of the code generated by C++ compilers, not hardware performance. Therefore, Bench++ is based on the Ada PIWG suite (see "PIWG Measurement Methodology," by D. Roy, *Ada Letters*, X(3), 1990), which was also designed to test individual language features, with some small applications and "traditional" benchmarks included. The tests are not just recoded Ada programs -- they are all reengineered into new C++ implementations, including the appropriate use of object-oriented features.

The discussion that follows deals with the version of the Bench++ 1.0. This version consists of more than 17,000 C++ source lines and contains more than 120 tests. Bench++ is freely available at http://www.research.att.com/~orost/bench_plus_plus.tar.Z, and is designed to be highly portable. No modification of the Bench++ test software is necessary. A configuration script, along with *#ifdef*s capture and automate most system dependencies. Timing routines are provided for UNIX, MacOS, and Windows NT.

Since practically all language features are tested, only modern C++ compilers can compile the Bench++ suite. Templates and exceptions must be supported. For g++, this means that Version 2.7.1 or better must be used.

Bench++ Suite Structure

The Bench++ suite measures the run-time performance for three test groups:

- Traditional benchmarks (Dhrystone, Whetstone, Hennessy).
- Applications (Tracker, Orbit, Kalman, Centroid).
- Language features (*if-then-else*, procedure/method calls, new/delete, exceptions, and so on).

The traditional benchmarks are designed to measure C++'s effectiveness on benchmarks traditionally coded in other languages. These benchmarks have been reengineered into C++, making use of object-oriented features where appropriate. These tests are artificial, geared to a narrow domain. Nonetheless, they provide an interesting comparison of compilers when run on the same hardware.

The Dhrystone (A000091) was originally written in Ada, and based on statement frequencies found in an examination of hundreds of high-level language programs. The original version was published in the *Communications of the ACM* in October 1984 (see "Dhrystone: A Synthetic Systems Programming Benchmark," by R.P. Weicker *CACM*, 27(10), October 1984). Versions written in Pascal, C, and Ada are available on the Internet. Analysis of the original program shows that it spends far too much time comparing 30-character strings.

The C++ version is based on Version 2.1 of the Ada and C code. Implementation of the *String* class was based on the libg++ implementation. Implementation of the discriminant record using inheritance and virtual functions was suggested by Professor Alex Borgida.

The Whetstone (A000092) attempts to model a "typical" mathematical processing workload. It was first described in Algol 60 in 1976 (see "A Synthetic Benchmark," by H.J. Curnow and B.A. Wichmann, *The Computer Journal*, 19(1), February 1976). It can be found in Ada, Fortran, and C on the Internet. Despite its limitations and age, it is still one of the most widely used benchmarks for measuring arithmetic performance.

The Hennessy benchmarks (A000094a-j) are named for Stanford University professor John L. Hennessy. They are a collection of short tests that are frequently used as benchmarks. Some simple correctness checks are included. They consist of:

- A000094a: Perm: Permutation Program, heavily recursive.
- A000094b: Towers: Towers of Hanoi.
- A000094c: Queens: Eight Queens problem solved 50 times.
- A000094d/e: IntMM/MM: Integer and float matrix multiplication, using a template class.
- A000094f: Puzzle: Simulate the solving of a puzzle.
- A000094g: Quick: Recursive quicksort.
- A000094h: Bubble: Bubble sort.
- A000094i: Tree: Tree insertion sort.
- A000094j: FFT: Fast Fourier Transform.
- A000094k: Ack: Ackerman's function *ack(3,6)* computed 10 times.

The current C++ version was reengineered from the Ada version, which originally came from various Fortran and Pascal implementations. Recursion is heavily used.

Four applications are available in Bench++ 1.0:

- B000002b/3b/4b: Path tracker using a covariance matrix (3 versions).

- B000010: NASA Orbit Determination.
- B000011: JIAWG Kalman Benchmark.
- B000013: Integer Tracker Centroid Algorithm.

All of these benchmarks were originally coded in JOVIAL and Ada. The three versions of the path tracker use *float*, *double*, and a combination of *float* and *integer*, respectively. The original fixed-point version of the benchmark has not been reproduced in C++ (I would appreciate an appropriate *fixed_point* class for implementation of this benchmark).

The language feature tests are designed to measure the implementation efficiency of individual language features. The language feature tests are divided into nine classes -- D, E, F, G, H, L, O, P, and S.

D tests measure dynamic allocation overhead with and without initialization of an array.

- D000001: Allocation of an array of 1000 integers using *malloc(3)*. The array is not initialized. The space is then freed using *free(3)*.
- D000002: Similar to D1, but the entire array is initialized. Typically, D2 runs an order of magnitude slower than D1.
- D000003: Similar to D1, but the proper C++ operators *new* and *delete* are used instead of *malloc(3)* and *free(3)*.
- D000004: Similar to D3 but the entire array is initialized.
- D000005: Similar to D1, but using the stack allocation routine, *alloca(3)*.
- D000006: Similar to D5, but the entire array is initialized.

E tests measure the overhead involved in throwing and catching C++ exceptions. Typically there is a delicate balance between fast exceptions and fast procedure calls, and fast procedure calls are much more important (see the P tests). For comparability with the Ada PIWG, there is no E000005 in Bench++.

- E000001: Time to raise and handle an exception locally.
- E000002: Exception is thrown in a class method (that has its own exception handler) and caught in the caller.
- E000003: Similar to E2, but exception is thrown nested three deep in calls.
- E000004: Similar to E3, but exception is nested four deep.
- E000006: Similar to E4, but exception is declared at each level.
- E000007: Similar to E4, but exception is caught and rethrown at each level.
- E000008: Simulates exceptions nested four deep using *setjmp*/*longjmp*.

F tests assess the impact of various coding styles on C++ run-time performance.

- F000001: Time to set a *bool* flag using a logical equation *flag = local < global;*.
- F000002: Similar to F1 but uses an *if* statement; see Listing One
- F000003: Time to test a global using a two-way *if*/*else if* statement.
- F000004: Similar to F3, but uses a 2-case *switch* statement.
- F000005: Time to test a global using a 10-way *if*/*else if* statement.

- F000006: Similar to F5, but uses a 10-case *switch* statement.
- F000007: Similar to F6, but uses a sparse switch.
- F000008: Similar to F5, but uses a 10-way virtual function class.

G tests measure the *iostream* class implementation. There is a large degree of variability in such tests because on some systems they measure simple buffer moves, but on other systems they may involve actual physical I/O. Also, operating system involvement may introduce even more overhead and inconsistency.

- G000001: *iostream.getline* and *gcount* reading 20 characters. A scratch file of 10 lines is written, then read and rewound repeatedly. Time per line is reported.
- G000002: *iostream.>> (char)* called 21 times per line. Same input as in G1. Time per line is reported.
- G000003: *iostream.<<* for 20-*char string* and *newline*s. Time per line is reported.
- G000004: *iostream.<< (char)* called 21 times. Time per line is reported.
- G000005: *istrstream.>>* an *int* variable from a local string.
- G000006: Similar to G5 but using a *float* variable.
- G000007: Time to open and close an existing file. The timing here is highly OS dependent, and the elapsed time is typically much higher than the CPU time (reported as a ratio).

H tests measure the effect of minimizing data space by using bit arrays and bit fields, and also assess the cost of performing type conversions.

- H000001: Bit array class operations. Time to perform "&", "|", "!", and "^" (and, or, not, and xor) on an entire bit-packed array of 16 bits. The bit array class was suggested by Bjarne Stroustrup.
- H000002: Byte array class operations. Similar to H1, but each *bool* object is stored as a byte. Comparing to H1 gives an indication of how cleverly bit array operations are implemented.
- H000003: Similar to H1, but the operations are performed component-wise in a loop. This would be necessary if the class implementation did not provide entire array operations.
- H000004: Similar to H2, but the operations are performed component-wise (like in H3).
- H000005: Time to move an integer from one variable to another using *j = *(int *)(&i);*. A smart optimizer can do this using a single register move.
- H000006: Time to move an array of 10 *float*s to a 10-component class object.
- H000007: Time to store and extract bit-fields that are declared as *int* and *bool* class members. The test comprises 12 access, five stores, and one class object copy.
- H000008: Time to store and extract bit fields and bit arrays that are declared as (nested) class members.
- H000009: Time to perform a class pack and unpack. Two similar classes are defined, one using bit fields and one without. Conversion operators are defined between them and are used to pack and then unpack an object.

L tests measure loop overhead.

- L000001: Simple *for* loop time: *for(int i = 1; i <= 100; i++) {}*. Time reported is once through the loop.
- L000002: Simple *while* loop time: *while(i <= 100) {}*. Time reported is once through the loop.
- L000003: Simple loop w/break time: *for(;;) { i++; if(i > 100) break; }*. Time reported is for once through the loop.
- L000004: Measures compiler's choice to unroll a small loop of five iterations. A time of zero indicates that the unrolling occurred.

O tests measure optimizer performance. These tests are based on Fortran benchmarks written in the late 1980s by M. Klerer and H. Liu (see "A New Benchmark Test to Estimate Optimization Quality of Compilers," *ACM SIGPLAN Not., 23(3)*, March 1988, and "Benchmark Test to Estimate Optimization Quality of Compilers," *ACM SIGPLAN Not., 23(10)*, October 1988). Each test is run in two parts: The test with the "a" suffix is designed to be optimized by the compiler. The corresponding "b" test is hand-optimized, and should be compared with the "a" test. Better optimizers should get similar times for each of the respective "a" and "b" tests.

- O000001a/b: Tests the Constant Propagation optimization, including calls to math functions with constant arguments.
- O000002a/b: Tests the Local Common Subexpression Elimination optimization, including duplicate calls to math functions.
- O000003a/b: Tests the Global Common Subexpression Elimination optimization.
- O000004a/b: Tests the Unnecessary Copy Elimination optimization.
- O000005a/b: Tests the Code Motion optimization, including loop-invariant calls to math functions.
- O000006a/b: Tests the Induction Variable optimization.
- O000007a/b: Tests the Strength Reduction optimization, including calls to math functions that can be simplified.
- O000008a/b: Tests the Dead Code Elimination optimization.
- O000009a/b: Tests the Loop Jamming optimization.
- O000010a/b: Tests the Redundant Code Elimination optimization.
- O000011a/b: Tests the Unreachable Code Elimination optimization.

P tests measure procedure call related overhead.

- P000001: Procedure call and return time. Procedure is local and has no parameters. Result may approach zero if compiler performs automatic inlining.
- P000002: Similar to P1, but procedure is not inlinable. Note that automatic inlining is defeated by coding a recursive call to the same procedure in an exception handler. However, this does not guarantee that inlining is actually defeated. In fact, when writing this article, I discovered that the CC compiler on Solaris is actually inlining the procedure, and is therefore breaking the benchmark. This has been fixed in Version 1.1 by using separate compilation.

- The introduction of the exception handler in the procedure may introduce extra overhead for some implementations. However, such undesirable extra overhead is a good side effect of this benchmarking effort, as it needs to be measured anyway.
- P000003: Similar to P2, but the procedure is declared as a static class method.
- P000004: Similar to P3, but the procedure is declared to be "inline," and does not catch exceptions. Compare to P1. If the time is not zero, then the compiler has disobeyed the inline declaration.
- P000005: Similar to P3, but the procedure has one *int* parameter.
- P000006: Similar to P5, but uses *int **.
- P000007: Similar to P5, but uses *int* &.
- P000008: Similar to P2, but the procedure is called through a procedure variable.
- P000010: Similar to P5, but has 10 *int* parameters.
- P000011: Similar to P5, but has 20 *int* parameters.
- P000012: Similar to P10, but each of the 10 parameters is a class object with three *int* members.
- P000013: Similar to P12, but has 20 parameters.
- P000020: Similar to P3, but not declared *static*. Has one implicit *this* parameter.
- P000021: Similar to P20, but declared *virtual*.
- P000022: Similar to P21, but declared *const*.
- P000023: Similar to P22, but procedure is called 100 times in a loop to see if the lookup code is optimized. Time reported is for one call.

S tests measure how well the C++ compiler generates code for object-oriented constructs. These tests are based on the OOPACK benchmarks written by Kuck & Associates (see "OOPACK: A Benchmark for Comparing OOP vs. C-style Programming," http://www.kai.com/oopack/oopack.html). Each test is run in two parts, the test with the "a" suffix is written in object-oriented C++ style. The corresponding "b" test is coded in C style, and should be compared with the "a" test. Better compilers should get similar times for each of the respective "a" and "b" tests.

- S000001a/b: Time to compute the maximum of a vector. Measures how well a compiler inlines a simple conditional.
- S000002a/b: Time to perform a double matrix multiplication. Measures how well a compiler propagates constants and hoists simple invariants.
- S000003a/b: Time to perform a vector inner product (using iterators in the OOP version). Measures how well a compiler inlines short-lived small objects.
- S000004a/b: Time to perform a complex phase shift. Measures how well a compiler eliminates temporaries.

Similar to the prior set, the following group of tests also measures the optimization of object-oriented constructs. These tests are based on the Stepanov benchmark, created by Alexander Stepanov (see "Stepanov Benchmark Results," http://www.kai.com/benchmarks/ stepanov/), the author of the Standard Template Library (STL). They code the same double array addition operation using more and more abstraction.

- S000005m: Abstraction Level 0: Uses a simple Fortran-style *for* loop.
- S000005l: Abstraction Level 1: Uses pointers and an STL-style accumulate template function with plus function object.
- S000005k: Abstraction Level 2: Same as level 1, and wraps the double type in a class.
- S000005j: Abstraction Level 3: Same as level 1, and wraps the pointers in a class.
- S000005i: Abstraction Level 4: Same as level 2, and wraps the pointers in a class.
- S000005h: Abstraction Level 5: Same as level 1, and uses a reverse-iterator adapter.
- S000005g: Abstraction Level 6: Same as level 2, and uses a reverse-iterator adapter.
- S000005f: Abstraction Level 7: Same as Level 1, and uses wrapped pointers in a reverse-iterator adapter.
- S000005e: Abstraction Level 8: Same as Level 2, and uses wrapped pointers in a reverse-iterator adapter.
- S000005d: Abstraction Level 9: Same as Level 5, and wrapped in another reverse-iterator adapter.
- S000005c: Abstraction Level 10: Same as Level 6, and wrapped in another reverse-iterator adapter.
- S000005b: Abstraction Level 11: Same as Level 7, and wrapped in another reverse-iterator adapter.
- S000005a: Abstraction Level 12: Same as Level 8, and wrapped in another reverse-iterator adapter.

Bench++ Suite Design

The Bench++ suite is designed using the "dual loop" paradigm and coded to work in spite of sophisticated code optimizers.

Listing Two is C++ pseudocode that shows the structure of the dual loop used by the Bench++ suite. As shown, a "null" loop is executed first and its overhead is measured. The time difference between the two loops divided by the number of iterations yields the timing of the feature being measured. Care must be taken when using this approach, as studies have shown that the dual loop approach can exhibit timing variations (up to 12 percent) due to word alignment on some architectures (see "Timing Variations in Dual Loop Benchmarks," by N. Altman and N. Weiderman, *Ada Letters*, *VIII(3)*, May 1989).

The *Trick_Optimizer()* routine is simply a call to a static class method with a global variable as a parameter. The routine simply increments the global. After each loop is executed the global is tested to make sure that it was incremented properly.

The *nr_iterations* in the control and test loop is determined dynamically. It typically starts out at 100 (but can start at another value chosen by the individual test writer), and gets multiplied by powers of four (4, 16, 64, 256, and so on) until the *feature_overhead* measurement is significant with respect to the CPU time clock tick.

An additional advantage of running the tests multiple times are that any possible demand-paging overheads are factored out, as the page faults are encountered the first time around the loop, and should not occur the next time (because most of the tests are very small).

Bench++ Suite Architecture

One of the main difficulties in designing a benchmark has to do with redundant and dead code elimination in optimizing compilers. The tests must be written in such a way that a "smart" optimizer is not able to remove the feature being measured because the result is not being used in the program.

The Bench++ approach is based on the PIWG approach, which is, in turn, based on the ideas of Bassman et al. in 1985 (see "An Approach for Evaluating the Performance Efficiency of Ada Compilers," by M.J. Bassman, G.A. Fisher, and A. Gargaro, *Proceedings of the 1985 International Ada Conference (Ada in Use)*, Cambridge University Press, May 1985). Each Bench++ program uses a class *break_optimization* to prevent the optimizer from removing the entire test. Procedure *iteration::increment* performs an increment of its argument, which is always passed as *break_optimization::global*. In addition, the names of the control and test procedures are bound as function arguments, and called through function pointers at run time. These factors combined should prevent the optimizer from deleting stores to global variables, which each test uses to prevent its results from being deleted. Listing Three illustrates the overall structure of a typical Bench++ test. In addition, some tests use checksums or recursion counts to make sure that the computations are being computed.

The *iteration* class provides timing services to the benchmark suite. It relies on implementation-dependent routines for CPU time measurement. If elapsed time is substituted for the CPU time because no CPU clock routine is available, then inaccuracies might result. This is because on most systems there are always some background processes running that result in process interruption and rescheduling. This nonprocess related timing can be added to the control loop or the test loop causing unpredictable results. In addition, using elapsed time instead of CPU time will foil the "erroneous result" check that the elapsed time is less than the CPU time.

Bench++ Results

Bench++ 1.0 was run on a variety of platforms and compilers. I report some of the more interesting results here.

The A000097 program provides comparison ratios of the various tests with other tests in the same suite, using the same compiler and target processor. Here are some of the more interesting ratios using g++ 2.7.2 on a Sparc Ultra-2.

Reference: D000005
Other: D000003
Ratio: 10.18

Description: *new*/*delete* versus *alloca*
Discussion: *alloca* is much faster than *new*/*delete*. Too bad you can't use it in a portable fashion.

Reference: E000003
Other: E000004
Ratio: 1.0
Description: Four-deep exception handed versus three-deep.
Discussion: Exception handling in g++ has the same overhead no matter how many levels of procedure calls need be "unwound."

Reference: F000006
Other: F000008
Ratio: 0.84
Description: 10-way virtual function class versus 10-way switch.
Discussion: Using C++ virtual functions is actually slightly better than using switch statements. This is good news!

Reference: G000001
Other: G000002
Ratio: 10.99
Description: 20-*iostream.>> versus* 20-*char iostream.getline* & *gcount.*
Discussion: Reading characters as strings will greatly enhance performance.

Reference: S000001b
Other: S000001a
Ratio: 2.63
Description: C++ style *Max* versus C style.
Discussion: g++ performs poorly on all of the C++-style and Stepanov tests. In this case the code for the C-style loop is unrolled eight times, while the C++-style loop is not, and extra unnecessary code for the compare is generated.

g++ versus CC

The A000096 program provides comparison ratios of one complete Bench++ run with another run. Here are some of the more interesting ratios comparing g++ 2.7.2 on a Sparc Ultra-2 to CC Version SC4.0 on the same processor. The options specified for CC are "-*native -libmil -fast -O4 -cg92.*" The options for g++ are "-*O3 -fhandle-exceptions -funroll-loops -ffast-math -mv8.*" Ratio values greater than 1.0 mean than CC generates faster code than g++.

Test: A000094i
Description: Hennessy Tree Insertion Sort.
Ratio: 2.45
Discussion: CC expands the insert function inline.

Test: B000002b
Description: Path tracker using a covariance matrix (*float*).
Ratio: 7.23
Discussion: CC optimizes out the computation of the prediction and smoothed matrices. B000003b and B000004b are optimized similarly, and have ratios of 7.27 and 6.64, respectively.

Test: D000005
Description: Allocation of 1000 integers using *alloca(3).*
Ratio: 2.98
Discussion: g++ doesn't inline the array constructor.

Test: E000001
Description: Time to raise and handle an exception locally.
Ratio: 0.35
Discussion: Exception handling in g++ seems to be much more efficient in general than that of CC. Similar ratios can be observed on all of the "E" tests.

Test: F000002
Description: Time to set a "*bool*" flag using an *if* statement.
Ratio: 1.98
Discussion: CC does a much better job of instruction scheduling to avoid load delays.

Test: F000003
Description: Time to test a global using a two-way *if*/*else if* statement.
Ratio: 0.43
Discussion: CC preloads *a_one* and stores into global even when all of the *if* conditions fail (which is the case in this test). By not performing this optimization, g++ actually generates more efficient code.

Test: G000001
Description: *iostream*,.*getline*, and *gcount* reading 20 characters.
Ratio: 0.31
Discussion: The g++ *iostream* implementation is more efficient than that of CC for input. The same is true for G000005 and G000006, whose ratios are 0.36 and 0.40, respectively.

Test: H000008
Description: Store/extract bit fields declared as nested class members.
Ratio: 1.98
Discussion: The CC generated code is 28 instructions, while that of g++ is 43 instructions, including extra code to reload scan and *a_big_one* from memory, and bogus code to branch around unreachable exception throwing code.

Test: L000001
Description: Simple *for* loop time.
Ratio: 2.19

Discussion: g++ unrolls the loop five times. However, it seems that the Ultra-2 likes the code better not unrolled! (Other machines behave just the opposite here.)

Test: L000003
Description: Simple *while* loop time.
Ratio: 9.73
Discussion: g++ generates bogus code to branch around unreachable exception throwing code, messing up the pipeline.

Test: L000004
Description: Measures compiler's choice to unroll a small loop of five iterations.
Ratio: 0.07
Discussion: g++ unrolls the loop completely.

Test: O000005a
Description: Tests the Code Motion optimization.
Ratio: 2.02
Discussion: g++ calls a *sqrt* subroutine. CC generates the *fsqrtd* instruction inline.

Test: O000007b
Description: Tests the Strength Reduction optimization.
Ratio: 2.08
Discussion: CC keeps *n* in a register (and in memory), while g++ loads it always from memory.

Test: O000010b
Description: Tests the Redundant Code Elimination optimization.
Ratio: 2.02
Discussion: g++ generates bogus code to branch around unreachable exception throwing code. CC does a better job of instruction scheduling to avoid load delays.

Test: P000002
Description: Procedure call and return time. Procedure is local and has no parameters, and is not inlinable.
Ratio: 4.29
Discussion: CC manages to inline the supposed "noninlinable" procedure in spite of the preventative measures taking by using both recursion and exceptions. Looks like the compilers are getting very smart, indeed! Tests P3, P5, P6, P7, and P10 are all affected by this. This is fixed in Version 1.1.

Test: S000002b
Description: Time to perform a double matrix multiplication. C-style. Measures how well a compiler propagates constants and hoists simple invariants.
Ratio: 2.84
Discussion: CC unrolls the inner loop three times (four iterations per trip). g++ only unrolls it twice. Also, CC optimizes the index arithmetic much better than g++. It

performs loop invariant strength reduction, and only generates 21 instructions to handle the four loop iterations, to g++'s 20 instructions for two iterations.

Test: S000004a
Description: Time to perform a vector inner product. OOP style using iterators. Measures how well a compiler inlines short-lived small objects.
Ratio: 2.52
Discussion: CC unrolls the loop once, and generates 81 instructions for the two iterations of the loop. g++ generates 50 instructions for one loop iteration, including unnecessary memory loads and stores, and branches around code to throw exceptions.

Test: S000004b
Description: Time to perform a vector inner product. C style.
Ratio: 5.09
Discussion: CC unrolls the loop twice (three iterations per trip), and generates 47 instructions for the three iterations of the loop. g++ unrolls the loop once (two iterations per trip) and generates 41 instructions for the two loop iterations, including unnecessary memory loads and stores, and branches around code to throw exceptions.

Test: S000005l
Description: C++ Style Abstraction: Level 1.
Ratio: 5.09
Discussion: CC generates eight instructions for the inner loop, and keeps a copy of the result in a register (and in memory). g++ generates 12 instructions, including branches around code to throw exceptions. It reloads result from memory.

C++ versus Ada

Since Bench++ is based on the Ada PIWG suite, I am able to compare the efficiency of the code generated for similar constructs. The Ada PIWG suite was run on a VAX 8650 with VAX/Ada 2.1-28. The Bench++ suite was run on a Sparc Ultra-2 with CC Version SC4.0. As expected, the Sparc Ultra-2 beats the VAX hands down, with the average ratio (geometric mean) being 0.0335 or almost 30 times faster. Unexpectedly, there are some tests that perform much worse than average. Here is a discussion of each one:

Test: A000094b
Description: Hennessy Benchmark: Towers -- Highly Recursive.
Ratio: 0.008 (over 4 times better than average).
Discussion: See the discussion of P1.

Test: B000002b
Description: Path tracker using a covariance matrix.
Ratio: 0.003 (over 11 times better than average).
Discussion: Ada requires both variable value and array bounds checking. C has no such requirement, and therefore, while the generated code can be much faster, there is no way to check for either out-of-bounds values, or out-of-bounds array subscripts, without using

special tools. The other path tracker tests (B000003b and B000004b) exhibit similar performance differences (approximately 17 and 7 times better than average, respectively).

Test: D000001
Description: Allocation of an array of 1000 integers using *malloc(3)*. The array is not initialized. The space if then freed using *free(3)*.
Ratio: 0.51 (over 15 times worse than average).
Discussion: This is not really a fair comparison. The Ada version allocates space on the stack, and the compiler inlines the allocation code. On the other hand, there is no way to do this in C++, so users must resort to a less efficient allocation method, or use the nonportable *alloca(3)* call, which is not available in all implementations. The Bench++ test D000005 is the fairer comparison here, using this nonportable *alloca(3)* call, and runs 1.5 times better than average when compared to the Ada D000001.

Test: D000003
Description: Similar to D1, but the proper C++ operators *new* and *delete* are used instead of *malloc(3)* and *free(3)*.
Ratio: 0.55 (over 16 times worse than average).
Discussion: See the discussion of D1.

Test: E000001
Description: Time to raise and handle an exception locally.
Ratio: 0.14 (over 4 times worse than average).
Discussion: VAX Ada has a more efficient implementation for all of the exception handling tests than CC does. However, CC does a relatively much better job on procedure calls.

Test: H000001
Description: Bit array class operations.
Ratio: 41.5 (over 1200 times worse than average).
Discussion: Packed bit arrays are a feature designed into most Ada compilers, and are well optimized in most implementations. Simulating a bit array using a *bit_ref* class along with a *bit_array* class does not generate very efficient code at all.

Test: P000001
Description: Procedure call and return time. Procedure is local and has no parameters.
Ratio: 0.003 (over 11 times better than average).
Discussion: VAXen are not noted for their efficient procedure calls. In addition, the extra code to get efficient exception handling adds to the cost of a procedure call in Ada. CC properly optimizes the more-used procedure call over exception efficiency. The other "P" tests show similar results, except for P4, where inlining cancels out the procedure call overhead, and P12 and P13 where the class object with three-*int*s is copied in C++ and passed by reference in Ada implementations.

Other C++ Benchmark Studies

I have found only three other publications that discuss C++ benchmarks.

Brad Calder, et al., from the University of Colorado published "Quantifying Behavioral Differences between C and C++ Programs" in the *Journal of Programming Language* 2(1994) in which they study the run-time characteristics of C and C++ programs. The programs are quite large, not easily portable, and not all generally available. They did not place them into a common timing framework. I feel that my work is significantly different than theirs because the Bench++ suite is generally available, easy to compile and run, and the tests are small enough to ease the analysis of the generated code.

Scott W. Haney, from the Lawrence Livermore National Laboratory, published "Is C++ Fast Enough for Scientific Computing?" in *Computers in Physics* (November/December 1994) and compares OOP-designed codes written in C++ with the corresponding Fortran codes. He tests only a very few C++ features, and his tests are similar to our S tests, which were based on the OOPACK suite by Kuck & Associates. While his tests are generally available (on the Internet), they are not easy to port, as they use an extensive set of preprocessor macros that are difficult to understand, and which cause the error messages from C++ compilers to be almost meaningless. On the other hand, my suite covers more of the language, includes "traditional" benchmarks, and includes similar OOP versus C-style tests. My results with g++ tend to support Haney's conclusion that C++ is not ready to be used for scientific computing. However, my findings show that the commercial compilers seem to do much better than g++ in optimizing object-oriented code, though there is still plenty of room for improvements.

Arch D. Robison from Kuck & Associates published "The Abstraction Penalty for Small Objects in C++" at the 1996 Parallel Object-Oriented Methods and Applications Conference (abstract available at http:// www.acl.lanl.gov/Pooma96/abstracts/ robison.html). He discusses the OOPACK, Haney, and Stepanov benchmarks (two of which can be found in the Bench++ S-tests), and what you can do to workaround the major abstraction penalty imposed by the currently available compilers. He then goes on to tout the Kuck & Associates compiler, which seems to do a much better job of optimizing C++ code constructs than the other currently available commercial compilers.

Bench++ Futures

It is my intention to maintain and enhance the Bench++ suite, as driven by the C++ community. More application programs that are self contained and modified to work in the Bench++ timing framework are also welcomed. If you are willing to partner with me in developing a Java version of these benchmarks, please let me know.

I'm also looking for suggestions for additional C++ language feature tests (preferably including the code for testing). Please send all such usability and enhancement suggestions to me at joseph.orost@att.com.

DDJ

Listing One

```
if(local < global) { flag = true;
} else {
 flag = false;
}
```

Back to Article

Listing Two

```
// Control loopstart = cpu_time_clock();
for(int i = 1; i <= nr_iterations; i++) {
 Trick_Optimizer();
}
stop = cpu_time_clock();
control_loop_overhead = stop - start;
```

// Test Loop start = cpu_time_clock(); for(int i = 1; i <= nr_iterations; i++) { Trick_Optimizer(); feature; } stop = cpu_time_clock(); test_loop_overhead = stop - start; feature_overhead = (test_loop_overhead - control_loop_overhead) / nr_iterations;

Back to Article

Listing Three

```
#include "a000010.h" // duration#include "a000021.h" //
break_optimization
#include "a000031.h" // iteration
#include "a000047.h" // bench_plus_plus_timer_generic
```

void TESTNAME_control(void) { iteration::increment(break_optimization::global); // Code here to mask overhead effects of the code to be measured. } void TESTNAME_test(void) { iteration::increment(break_optimization::global); // Code to be measured goes here } void TESTNAME(void) // main procedure { scale scale = automatic; bench_plus_plus_timer_generic(scale, TESTNAME_control, TESTNAME_test, "TESTNAME", "TEST-CLASS", "Description: line 1", " line 2", " line 3"); }

Back to Article