
Measurement and Analysis of C and C++ Performance

Hemant G. Rotithor
Kevin W. Harris
Mark W. Davis

As computer languages and architectures evolve, many more challenges are being presented to compilers. Dealing with these issues in the context of the Alpha Architecture and the C and C++ languages has led Compaq's C and C++ compiler and engineering teams to develop a systematic approach to monitor and improve compiler performance at both run time and compile time. This approach takes into account five major aspects of product quality: function, reliability, performance, time to market, and cost. The measurement framework defines a controlled test environment, criteria for selecting benchmarks, measurement frequency, and a method for discovering and prioritizing opportunities for improvement. Three case studies demonstrate the methodology, the use of measurement and analysis tools, and the resulting performance improvements.

Optimizing compilers are becoming ever more complex as languages, target architectures, and product features evolve. Languages contribute to compiler complexity with their increasing use of abstraction, modularity, delayed binding, polymorphism, and source reuse, especially when these attributes are used in combination. Modern processor architectures are evolving ever greater levels of internal parallelism in each successive generation of processor design. In addition, product feature demands such as support for fast threads and other forms of external parallelism, integration with smart debuggers, memory use analyzers, performance analyzers, smart editors, incremental builders, and feedback systems continue to add complexity. At the same time, traditional compiler requirements such as standards conformance, compatibility with previous versions and competitors' products, good compile speed, and reliability have not diminished.

All these issues arise in the engineering of Compaq's C and C++ compilers for the Alpha Architecture. Dealing with them requires a disciplined approach to performance measurement, analysis, and engineering of the compiler and libraries if consistent improvements in out-of-the-box and peak performance on Alpha processors are to be achieved. In response, several engineering groups working on Alpha software have established procedures for feature support, performance measurement, analysis, and regression testing.

The operating system groups measure and improve overall system performance by providing system-level tuning features and a variety of performance analysis tools. The Digital Products Division (DPD) Performance Analysis Group is responsible for providing official performance statistics for each new processor measured against industry-standard benchmarks, such as SPECmarks published by the Standard Performance Evaluation Corporation and the TPC series of transaction processing benchmarks from the Transaction Processing Performance Council. The DPD Performance Analysis Group has established rigorous methods for analyzing these benchmarks and provides performance regression testing for new software versions.

Similarly, the Alpha compiler back-end development group (GEM) has established performance improvement and regression testing procedures for SPECmarks; it also performs extensive run-time performance analysis of new processors, in conjunction with refining and developing new optimization techniques. Finally, consultants working with independent software vendors (ISVs) help the ISVs port and tune their applications to work well on Alpha systems.

Although the effort from these groups does contribute to competitive performance, especially on industry-standard benchmarks, the DEC C and C++ compiler engineering teams have found it necessary to independently monitor and improve both run-time and compile-time performance. In many cases, ISV support consultants have discovered that their applications do not achieve the performance levels expected based on industry-standard benchmarks. We have seen a variety of causes: New language constructs and product features are slow to appear in industry benchmarks, thus these optimizations have not received sufficient attention. Obsolete or obsolescent source code remaining in the bulk of existing applications causes default options/switches to be selected that inhibit optimizations. Many of the most important optimizations used for exploiting internal parallelism make assumptions about code behavior that prove to be wrong. Bad experiences with compiler bugs induce users to avoid optimizations entirely. Configuration and source-code changes made just before a product is released can interfere with important optimizations.

For all these reasons, we have used a systematic approach to monitor, improve, and trade off five major aspects of product quality in the DEC C and DIGITAL C++ compilers. These aspects are function, reliability, performance, time to market, and cost. Each aspect is chosen because it is important in isolation and because it trades off against each of the other aspects. The objective of this paper is to show how the one characteristic of performance can be improved while minimizing the impact on the other four aspects of product quality.

In this paper, we do not discuss any individual optimization methods in detail; there is a plethora of literature devoted to these topics, including a paper published in this *Journal*.¹ Nor do we discuss specific compiler product features needed for competitive support on individual platforms. Instead, we show how the efforts to measure, monitor, and improve performance are organized to minimize cost and time to market while maximizing function and reliability. Since all these product aspects are managed in the context of a series of product releases rather than a single release, our goals are frequently expressed in terms of relationships between old and new product versions.

For example, for the performance aspects, goals along the following lines are common:

- Optimizations should not impose a compile-speed penalty on programs for which they do not apply.
- The use of unrelated compiler features should not degrade optimizations.
- New optimizations should not degrade reliability.
- New optimizations should not degrade performance in any applications.
- Optimizations should not impose any nonlinear compile-speed penalty.
- No application should experience run-time speed regressions.
- Specific benchmarks or applications should achieve specific run-time speed improvements.
- The use of specific new language features should not introduce compile-speed or run-time regressions.

In the context of performance, the term *measurement* usually refers to crude metrics collected during an automated script, such as compile time, run time, or memory usage. The term *analysis*, in contrast, refers to the process of breaking down the crude measurement into components and discovering how the measurement responds to changing conditions. For example, we analyze how compile speed responds to an increase in available physical memory. Often, a comprehensive analysis of a particular issue may require a large number of crude measurements. The goal is usually to identify a particular product feature or optimization algorithm that is failing to obey one of the product goals, such as those listed above, and repair it, replace it, or amend the goal as appropriate. As always, individual instances of this approach are interesting in themselves, but the goal is to maximize the overall performance while minimizing the development cost, new feature availability, reliability, and time to market for the new version.

Although some literature²⁻⁴ discusses specific aspects of analyzing and improving performance of C and C++ compilers, a comprehensive discussion of the practical issues involved in the measurement and analysis of compiler performance has not been presented in the literature to our knowledge. In this paper, we provide a concrete background for a practitioner in the field of compilation-related performance analysis.

In the next section, we describe the metrics associated with the compiler's performance. Following that, we discuss an environment for obtaining stable performance results, including appropriate benchmarks, measurement frequency, and management of the results. Finally, we discuss the tools used for performance measurement and analysis and give examples of the use of those tools to solve real problems.

Performance Metrics

In our experience, ISVs and end users are most interested in the following performance metrics:

- **Function.** Although function is not usually considered an aspect of performance, new language and product features are entirely appropriate to consider among potential performance improvements when trading off development resources. From the point of view of a user who needs a particular feature, the absence of that feature is indistinguishable from an unacceptably slow implementation of that feature.
- **Reliability.** Academic papers on performance seldom discuss reliability, but it is crucial. Not only is an unreliable optimization useless, often it prejudices programmers against using any optimizations, thus degrading rather than enhancing overall performance.
- **Application absolute run time.** Typically, the absolute run time of an application is measured for a benchmark with specific input data. It is important to realize, however, that a user-supplied benchmark is often only a surrogate for the maximum application size.
- **Maximum application size.** Often, the end user is not trying to solve a specific input set in the shortest time; instead, the user is trying to solve the largest possible real-world problem within a specific time. Thus, trends (e.g., memory bandwidth) are often more important than absolute timings. This also implies that specific benchmarks must be retired or upgraded when processor improvements moot their original rationale.
- **Price/Performance ratio.** Often, the most effective competitor is not the one who can match our product's performance, but the one who can give acceptable performance (see above) with the cheapest solution. Since compiler developers do not contribute directly to server or workstation pricing decisions, they must use the previous metrics as surrogates.
- **Compile speed.** This aspect is primarily of interest to application developers rather than end users. Compile speed is often given secondary consideration in academic papers on optimization; however, it can make or break the decision of an ISV considering a platform or a development environment. Also, for C++, there is an important distinction between ab initio build speed and incremental build speed, due to the need for template instantiation.
- **Result file size.** Both the object file and executable file sizes are important. This aspect was not a particular problem with C, but several language features of C++ and its optimizations can lead to explosive growth in result file size. The most obvious problems are the need for extensive function inlining

and for instantiation of templates. In addition, for debug versions of the result files, it is essential to find a way to suppress repeated descriptions of the type information for variables in multiple modules.

- **Compiler dynamic memory use.** Peak usage, average usage, and pattern of usage must be regulated to keep the cost of a minimum development configuration low. In addition, it is important to ensure that specific compiler algorithms or combinations of them do not violate the usage assumptions built into the paging system, which can make the system unusable during large compilations.

Crude measurements can be made for all or most of these metrics in a single script. When attempting to make a significant improvement in one or more metrics, however, the change often necessarily degrades others. This is acceptable, as long as the only cases that pay a penalty (e.g., in larger dynamic memory use) are the compilations that benefit from the improved run-time performance.

As the list of performance metrics indicates, the most important distinction is made between compile-time and run-time metrics. In practice, we use automated scripts to measure compile-time and run-time performance on a fairly frequent (daily or weekly during development) basis.

Compile-Time Performance Metrics

To measure compile-time performance, we use four metrics: compilation time, size of the generated objects, dynamic memory usage during compilation, and template instantiation time for C++.

Compilation Time The compilation time is measured as the time it takes to compile a given set of sources, typically excluding the link time. The link time is excluded so that only compiler performance is measured. This metric is important because it directly affects the productivity of a developer. In the C++ case, performance is measured ab initio, because our product set does not support incremental compilation below the granularity of a whole module. When optimization of the entire program is attempted, this may become a more interesting issue. The UNIX shell timing tools make a distinction between user and system time, but this is not a meaningful distinction for a compiler user. Since compilation is typically CPU intensive and system time is usually modest, tracking the sum of both the user and the system time gives the most realistic result. Slow compilation times can be caused by the use of $O(n^2)$ algorithms in the optimization phases, but they can also be frequently caused by excessive layering or modularity due to code reuse or excessive growth of the in-memory representation of the program during compilation (e.g., due to inlining).

Size of Generated Objects Excessive size of generated objects is a direct contributor to slow compile and link times. In addition to the obvious issues of inlining and template instantiation, duplication of the type and naming information in the symbolic debugging support has been a particular problem with C++. Compression is possible and helps with disk space, but this increases link time and memory use even more. The current solution is to eliminate duplicate information present in multiple modules of an application. This work requires significant support in both the linker and the debugger. As a result, the implementation has been difficult.

Dynamic Memory Usage during Compilation Usually modern compilers have a multiphase design whereby the program is represented in several different forms in dynamic memory during the compilation process. For C and C++ optimized compilations, this involves at least the following processes:

- Retrieving the entire source code for a module from its various headers
- Preprocessing the source according to the C/C++ rules
- Parsing the source code and representing it in an abstract form with semantic information embedded
- For C++, expanding template classes and functions into their individual instances
- Simplifying high-level language constructs into a form acceptable to the optimization phases
- Converting the abstract representation to a different abstract form acceptable to an optimizer, usually called an intermediate language (IL)
- Expanding some low-level functions inline into the context of their callers
- Performing multiple optimization passes involving annotation and transformation of the IL
- Converting the IL to a form symbolically representing the target machine language, usually called code generation
- Performing scheduling and other optimizations on the symbolic machine language
- Converting the symbolic machine language to actual object code and writing it onto disk

In modern C and C++ compilers, these various intermediate forms are kept entirely in dynamic memory. Although some of these operations can be performed on a function-by-function basis within a module, it is sometimes necessary for at least one intermediate form of the module to reside in dynamic memory in its entirety. In some instances, it is necessary to keep multiple forms of the whole module simultaneously.

This presents a difficult design challenge: how do we compile large programs using an acceptable amount of virtual and physical memory? Trade-offs change constantly as memory prices decline and paging algorithms of operating systems change. Some optimizations even have the potential to expand one of the intermediate representations into a form that grows faster than the size of the program ($O(n \times \log(n))$, or even $O(n^2)$). In these cases, optimization designers often limit the scope of the transformation to a subset of an individual function (e.g., a loop nest) or use some other means to artificially limit the dynamic memory and computation requirements. To allow additional headroom, upstream compiler phases are designed to eliminate unnecessary portions of the module as early as possible.

In addition, the memory management systems are designed to allow internal memory reuse as efficiently as possible. For this reason, compiler designers at Compaq have generally preferred a zone-based memory management approach rather than either a malloc-based or a garbage-collection approach. A zoned memory approach typically allows allocation of varying amounts of memory into one of a set of identified zones, followed by deallocation of the entire zone when all the individual allocations are no longer needed. Since the source program is represented by a succession of internal representations in an optimizing compiler, a zoned-based memory management system is very appropriate.

The main goals of the design are to keep the peak memory use below any artificial limits on the virtual memory available for all the actual source modules that users care about, and to avoid algorithms that access memory in a way that causes excessive cache misses or page faults.

Template Instantiation Time for C++ Templates are a major new feature of the C++ language and are heavily used in the new Standard Library. Instantiation of templates can dominate the compile time of the modules that use them. For this reason, template instantiation is undergoing active study and improvement, both when compiling a module for the first time and when recompiling in response to a source change. An improved technique, now widely adopted, retains pre-compiled instantiations in a library to be used across compilations of multiple modules.

Template instantiation may be done at either compile time or during link time, or some combination.⁵ DIGITAL C++ has recently changed from a link-time to a compile-time model for improved instantiation performance. The instantiation time is generally proportional to the number of templates instantiated, which is based on a command-line switch specification and the time required to instantiate a typical template.

Run-Time Performance Metrics

We use automated scripts to measure run-time performance for generated code, the debug image size, the production image size, and specific optimizations triggered.

Run Time for Generated Code The run time for generated code is measured as the sum of user and system time on UNIX required to run an executable image. This is the primary metric for the quality of generated code. Code correctness is also validated. Comparing run times for slightly differing versions of synthetic benchmarks allows us to test support for specific optimizations. Performance regression testing on both synthetic benchmarks and user applications, however, is the most cost-effective method of preventing performance degradations. Tracing a performance regression to a specific compiler change is often difficult, but the earlier a regression is detected, the easier and cheaper it is to correct.

Debug Image Size The size of an image compiled with the debug option selected during compilation is measured in bytes. It is a constant struggle to avoid bloat caused by unnecessary or redundant information required for symbolic debugging support.

Production Image Size The size of a production (optimized, with no debug information) application image is measured in bytes. The use of optimization techniques has historically made this size smaller, but modern RISC processors such as the Alpha microprocessor require optimizations that can increase code size substantially and can lead to excessive image sizes if the techniques are used indiscriminately. Heuristics used in the optimization algorithms limit this size impact; however, subtle changes in one part of the optimizer can trigger unexpected size increases that affect I-cache performance.

Specific Optimizations Triggered In a multiphase optimizing compiler, a specific optimization usually requires preparatory contributions from several upstream phases and cleanup from several downstream phases, in addition to the actual transformation. In this environment, an unrelated change in one of the upstream or downstream phases may interfere with a data structure or violate an assumption exploited by a downstream phase and thus generate bad code or suppress the optimizations. The generation of bad code can be detected quickly with automated testing, but optimization regressions are much harder to find.

For some optimizations, however, it is possible to write test programs that are clearly representative and can show, either by some kind of dumping or by comparative performance tests, when an implemented optimization fails to work as expected. One

commercially available test suite is called NULLSTONE,⁶ and custom-written tests are used as well.

In a collection of such tests, the total number of optimizations implemented as a percentage of the total tests can provide a useful metric. This metric can indicate if successive compiler versions have improved and can help in comparing optimizations implemented in compilers from different vendors. The optimizations that are indicated as not implemented provide useful data for guiding future development effort.

The application developer must always consider the compile-time versus run-time trade-off. In a well-designed optimizing compiler, longer compile times are exchanged for shorter run times. This relationship, however, is far from linear and depends on the importance of performance to the application and the phase of development.

During the initial code-development stage, a shorter compile time is useful because the code is compiled often. During the production stage, a shorter run time is more important because the code is run often. Although most of the above metrics can be directly measured, dynamic memory use can only be indirectly observed, for example, from the peak stack use and the peak heap use. As a result, our tests include benchmarks that potentially make heavy use of dynamic memory. Any degradation in a newer compiler version can be deduced from observing the compilation of such test cases.

Environment for Performance Measurement

In this section, we describe our testing environment, including hardware and software requirements, criteria for selecting benchmarks, frequency of performance measurement, and tracking the results of our performance measurements.

Compiler performance analysis and measurement give the most reliable and consistent results in a controlled environment. A number of factors other than the compiler performance have the potential of affecting the observed results, and the effect of such perturbations must be minimized. The hardware and software components of the test environment used are discussed below.

Experience has shown that it helps to have a dedicated machine for performance analysis and measurement, because the results obtained on the same machine tend to be consistent and can be meaningfully compared with successive runs. In addition, the external influences can be closely controlled, and versions of system software, compilers, and benchmarks can be controlled without impacting other users.

Several aspects of the hardware configuration on the test machine can affect the resulting measurements. Even within a single family of CPU architectures at comparable clock speeds, differences in specific imple-

mentations can cause significant performance changes. The number of levels and the sizes of the on-chip and board-level caches can have a strong effect on performance in a way that depends on algorithms of the application and the size of the input data set. The size and the access speed of the main memory strongly affect performance, especially when the application code or data does not fit into the cache. The activity on a network connected to the test system can have an effect on performance; for example, if the test sources and the executable image are located on a remote disk and are fetched over a network. Variations in the observed performance may be divided into two parts: (1) system-to-system variations in measurement when running the same benchmark and (2) run-to-run variation on the same system running the same benchmark.

Variation due to hardware resource differences between systems is addressed by using a dedicated machine for performance measurement as indicated above. Variation due to network activity can be minimized by closing all the applications that make use of the network before the performance tests are started and by using a disk system local to the machine under test. The variations due to cache and main memory system effects can be kept consistent between runs by using similar setups for successive runs of performance measurement.

In addition to the hardware components of the setup described above, several aspects of the software environment can affect performance. The operating system version used on the test machine should correspond to the version that the users are likely to use on their machines, so that the users see comparable performance. The libraries used with the compiler are usually shipped with the operating system. Using different libraries can affect performance because newer libraries may have better optimizations or new features. The compiler switches used while compiling test sources can result in different optimization trade-offs. Due to the large number of compiler options supported on a modern compiler, it is impractical to test performance with all possible combinations.

To meet our requirements, we used the following small set of switch combinations:

1. **Default Mode.** The default mode represents the default combination of switches selected for the compiler when no user-selectable options are specified. The compiler designer chooses the default combination to provide a reasonable trade-off between compile speed and run speed. The use of this mode is very common, especially by novices, and thus is important to measure.
2. **Debug Mode.** In the debug mode, we test the option combination that the programmer would select when debugging. Optimizations are typically turned off, and full symbolic information is generated about the

types and addresses of program variables. This mode is commonly specified during code development.

3. **Optimize/Production Mode.** In the optimize/production mode, we select the option combination for generating optimized code (`-O` compiler option) for a production image. This mode is most likely to be used in compiling applications before shipping to customers.

We prefer to measure compile speed for debug mode, run speed for production mode, and both speeds for the default mode. The default mode is expected to lose only modest run speed over optimize mode, have good compile speed, and provide usable debug information.

Criteria for Selecting Benchmarks

Specific benchmarks are selected for measuring performance based on the ease of measuring interesting properties and the relevance to the user community. The desirable characteristics of useful benchmarks are

- It should be possible to measure individual optimizations implemented in the compiler.
- It should be possible to test performance for commonly used language features.
- At least some of the benchmarks should be representative of widely used applications.
- The benchmarks should provide consistent results, and the correctness of a run should be verifiable.
- The benchmarks should be scalable to newer machines. As newer and faster machines are developed, the benchmark execution times diminish. It should be possible to scale the benchmarks on the machines, so that useful results can still be obtained without significant error in measurement.

To meet these diverse requirements, we selected a set of benchmarks, each of which meets some of the requirements. We grouped our benchmarks in accordance with the performance metrics, that is, as compile-time and run-time benchmarks. This distinction is necessary because it allows us to fine-tune the contents of the benchmarks under each category. The compile-time and run-time benchmarks may be further classified as (1) synthetic benchmarks for testing the performance of specific features or (2) real applications that indicate typical performance and combine the specific features.

Compile-Time Benchmarks Examples of synthetic compile-time benchmarks include the `#define` intensive preprocessing test, the array intensive test, the comment intensive test, the declaration processing intensive test, the hierarchical `#include` intensive test, the `printf` intensive test, the empty `#include` intensive test, the arithmetic intensive test, the function definition intensive test (needs a large memory), and the instantiation intensive test.

Real applications used as compile-time benchmarks include selected sources from the C compiler, the DIGITAL UNIX operating system, UNIX utilities such as `awk`, the X window interface, and C++ class inheritance.

Run-Time Benchmarks Synthetic run-time benchmarks contain tests for individual optimizations for different data type, storage types, and operators. One run-time suite called NULLSTONE⁶ contains tests for C and C++ compiler optimizations; another test suite called Bench++⁷ has tests for C++ features such as virtual function calls, exception handling, and abstraction penalty (the Haney kernels test, the Stepanov benchmark, and the OOPACK benchmark⁸).

Run-time benchmarks of real applications for the C language include some of the SPEC tests that are closely tracked by the DPD Performance Group. For C++, the tests consist of the groff word processor processing a set of documents, the EON ray tracing benchmark, the Odbsim-a database simulator from the University of Colorado, and tests that call functions from a search class library.

Acquiring and Maintaining Benchmarks

We have established methods of acquiring, maintaining, and updating benchmarks. Once the desirable characteristics of the benchmarks have been identified, useful benchmarks may be obtained from several sources, notably a standards organization such as SPEC or a vendor such as Nullstone Corporation. The public domain can provide benchmarks such as EON, groff, and Bench++. The use of a public-domain benchmark may require some level of porting to make the benchmark usable on the test platform if the original application was developed for use with a different language dialect, e. g., GNU's `gcc`.

Sometimes, customers encounter performance problems with a specific feature usage pattern not anticipated by the compiler developers. Customers can provide extracts of code that a vendor can use to reproduce these performance problems. These code extracts can form good benchmarks for use in future testing to avoid reoccurrence of the problem.

Application code such as extracts from the compiler sources can be acquired from within the organization. Code may also be obtained from other software development groups, e. g., the class library group, the debugger group, and the operating system group.

If none of these sources can yield a benchmark with a desirable characteristic, then one may be written solely to test the specific feature or combination.

In our tests of the DIGITAL C++ compiler, we needed to use all the sources discussed above to obtain C++ benchmarks that test the major features of the language. The public-domain benchmarks sometimes required a significant porting effort because of com-

patibility issues between different C++ dialects. We also reviewed the results published by other C++ compiler vendors.

Maintaining a good set of performance measurement benchmarks is necessary for evolving languages such as C and C++. New standards are being developed for these languages, and standards compatibility may make some of a benchmark's features obsolete. Updating the database of benchmarks used in testing involves

- Changing the source of existing benchmarks to accommodate system header and default behavior changes
- Adding new benchmarks to the set when new compiler features and optimizations are implemented
- Deleting outdated benchmarks that do not scale well to newer machines

In the following subsection, we discuss the frequency of our performance measurement.

Measurement Frequency

When deciding how often to measure compiler performance, we consider two major factors:

- It is costly to track down a specific performance regression amid a large number of changes. In fact, it sometimes becomes more economical to address a new opportunity instead.
- In spite of automation, it is still costly to run a suite of performance tests. In addition to the actual run time and the evaluation time, and even with significant efforts to filter out noise, the normal run-to-run variability can show phantom regressions or improvements.

These considerations naturally lead to two obvious approaches to test frequency:

- Measuring at regular intervals. During active development, measuring at regular intervals is the most appropriate policy. It allows pinpointing specific performance regressions most cheaply and permits easy scheduling and cost management. The interval selected depends on the amount of development (number of developers and frequency of new code check-ins) and the cost of the testing. In our tests, the intervals have been as frequent as three days and as infrequent as 30 days.
- Measuring on demand. Measurement is performed on demand when significant changes occur, for example, the delivery of a major new version of a component or a new version of the operating system. A full performance test is warranted to establish a new baseline when a competitor's product is released or to ensure that a problem has been corrected.

Both strategies, if implemented purely, have problems. Frequent measurement can catch problems early but is

resource intensive, whereas an on-demand strategy may not catch problems early enough and may not allow sufficient time to address discovered problems. In retrospect, we discovered that the time devoted to more frequent runs of existing tests could be better used to develop new tests or analyze known results more fully.

We concluded that a combination strategy is the best approach. In our case all the performance tests are run prior to product releases and after major component deliveries. Periodic testing is done during active development periods. The measurements can be used for analyzing existing problems, analyzing and comparing performance with a competing product, and finding new opportunities for performance improvement.

Managing Performance Measurement Results

Typically, the first time a new test or analysis method is used, a few obvious improvement opportunities are revealed that can be cheaply addressed. Long-term improvement, however, can only be achieved by going beyond this initial success and addressing the remaining issues, which are either costly to implement or which occur infrequently enough to make the effort seem unworthy. This effort involves systematically tracking the performance issues uncovered by the analysis and judging the trends to decide which improvement efforts are most worthwhile.

Our experience shows that rigorously tracking all the performance issues resulting from the analyses provides a long list of opportunities for improvement, far more than can be addressed during the development of a single release. It thus became obvious that, to deploy our development resources most effectively, we needed to devise a good prioritization scheme.

For each performance opportunity on our list, we keep crude estimates of three criteria: usage frequency, payoff from implementation, and difficulty of implementation. We then use the three criteria to divide the space of performance issues into equivalence classes. We define our criteria and estimates as follows:

- Usage frequency. The usage frequency is said to be *common* if the language feature or code pattern appears in a large fraction of source modules or *uncommon* if it appears in only a few modules. When the language feature or code pattern appears in most modules for a particular application domain predominantly, the usage frequency is said to be skewed. The classic example of *skewed* usage is the complex data type.
- Payoff from implementation. Improvement in an implementation is estimated as high, moderate, or small. A *high* improvement would be the elimination of the language construct (e.g., removal of unnecessary constructors in C++) or a significant fraction of their overhead (e.g., inlining small func-

tions). A *moderate* improvement would be a 10 to 50 percent increase in the speed of a language feature. A *small* improvement such as loop unrolling is worthwhile because it is common.

- Difficulty of implementation. We estimate the resource cost for implementing the suggested optimization as difficult, straightforward, or easy. Items are classified based on the complexity of design issues, total code required, level of risk, or number and size of testing requirements. An *easy* improvement requires little up-front design and no new programmer or user interfaces, introduces little breakage risk for existing code, and is typically limited to a single compiler phase, even if it involves a substantial amount of new code. A *straightforward* improvement would typically require a substantial design component with multiple options and a substantial amount of new coding and testing but would introduce little risk. A *difficult* improvement would be one that introduces substantial risk regardless of the design chosen, involves a new user interface, or requires substantial new coordination between components provided by different groups.

For each candidate improvement on our list, we assign a triple representing its priority, which is a Cartesian product of the three components above:

$$\text{Priority} = (\text{frequency}) \times (\text{payoff}) \times (\text{difficulty})$$

This classification scheme, though crude and subjective, provides a useful base for resource allocation. Opportunities classified as common, high, and easy are likely to provide the best resource use, whereas those issues classified as uncommon, small, and difficult are the least attractive. This scheme also allows management to prioritize performance opportunities against functional improvements when allocating resources and schedule for a product release.

Further classification requires more judgment and consideration of external forces such as usage trends, hardware design trends, resource availability, and expertise in a given code base. Issues classified as common and high but difficult are appropriate for a major achievement of a given release, whereas an opportunity that is uncommon and moderate but easy might be an appropriate task for a novice compiler developer.

So-called “nonsense optimizations” are often controversial. These are opportunities that are almost nonexistent in human-written source code, for example, extensive operations on constants. Ordinarily they would be considered unattractive candidates; however, they can appear in hidden forms such as the result of macro expansion or as the result of optimizations performed by earlier phases. In addition, they often have high per-use payoff and are easy to implement, so it is usually worthwhile to implement new nonsense optimizations when they are discovered.

Management control and resource allocation issues can arise when common, high, or easy opportunities involve software owned by groups not under the direct control of the compiler developers, such as headers or libraries.

Tools and Methodology

We begin this section with a discussion of performance evaluation tools and their application to problems. We then briefly present the results of three case studies.

Tools and Their Application to Problems

Tools for performance evaluation are used for either measurement or analysis. Tools for measurement are designed mainly for accurate, absolute timing. Low overhead, reproducibility, and stability are more important than high resolution. Measurement tools are primarily used in regression testing to identify the existence of new performance problems. Tools for analysis, on the other hand, are used to isolate the source code responsible for the problem. High, relative accuracy is more important than low overhead or stability here. Analysis tools tend to be intrusive: they add instrumentation to either the sources or the executable image in some manner, so that enough information about the execution can be captured to provide a detailed profile.

We have constructed adequate automated measurement tools using scripts layered over standard operating system timing packages. For compile-time measurement, a driver reads the compile commands from a file and, after compiling the source the specified number of times, writes the resulting timings to a file. Post-processing scripts evaluate the usability of the results (average times, deviations, and file sizes) and compare the new results against a set of reference results. For compile-time measurement, the default, debug, and optimize compilation modes are all tested, as previously discussed.

These summarized results indicate if the test version has suffered performance regressions, the magnitude of these regressions, and which benchmark source is exhibiting a regression. Analysis of the problem can then begin.

The tools we use for compile-speed and run-time analysis are considerably more sophisticated than the measurement tools. They are generally provided by the CPU design or operating system tools development groups and are widely used for application tuning as well as compiler improvements. We have used the following compile-speed analysis tools:

- The compiler's internal `-show statistics` feature gives a crude measure of the time required for each compiler phase.

- The `gprof` and `hiprof` tools are supplied in the development suites for DIGITAL UNIX. Both operate by building an instrumented version of the test software (the compiler itself in our case). The `gprof` tool works with the compiler, the linker, and the loader; it is available from several UNIX vendors. `Hiprof` is an Atom tool⁹⁻¹¹ available only on DIGITAL UNIX; it does not require compiler or linker support.

The benchmark exhibiting the performance problem can then be compiled with the profiling version of the compiler, and the compilation profile can be captured. Using the display facilities of the tool, we can analyze the relevant portions of the execution profile. We can then compare this profile with that of the reference version to localize the problem to a specific area of compiler source. Once this information is available, a specific edit can be identified as the cause and a solution can be identified and implemented. Another round of measurement is needed to verify the repair is effective, similar to the procedure for addressing a functional regression.

- When the problem needs to be pinpointed more accurately than is possible with these profiling tools, we use the `IProbe` tool, which can provide instruction-by-instruction details about the execution of a function.¹⁴

We have used the following tools or processes for run-time analysis:

- We apply `hiprof` and `gprof` in combination, and the `IProbe` tool as described above, to the run-time behavior of the test program rather than to its compilation.
- We analyze the `NULLSTONE` results by examining the detailed log file. This log identifies the problem and the machine code generated. This analysis is usually adequate since the tests are generally quite simple.
- If more detailed analysis is needed, e.g., to pinpoint cache misses, we use the highly detailed results generated by the Digital Continuous Profiling Infrastructure (DCPI) tool.^{12,13} DCPI can display detailed (average) hardware behavior on an instruction-by-instruction basis. Any scheduling problems that may be responsible for frequent cache misses can be identified from the DCPI output, whereas they may not always be obvious from casually observing the machine code.
- Finally, we use the estimated schedule dump and statistical data optionally generated by the `GEM` back end.¹ This dump tells us how instructions are scheduled and issued based on the processor architecture selected. It may also provide information about ways to improve the schedule.

In the rest of this section, we discuss three examples of applying analysis tools to problems identified by the performance measurement scripts.

Compile-Time Test Case

Compile-time regression occurred after a new optimization called base components was added to the GEM back end to improve the run-time performance of structure references. Table 1 gives compile-time test results that compare the ratios of compile times using the new optimized back end to those obtained with the older back end. The results for the iostream test indicate a significant degradation of 25 percent in the compile speed for optimize mode, whereas the performance in the other two modes is unchanged.

To analyze this problem, we built hiprof versions of the two compilers and compiled the iostream benchmark to obtain its compilation profile. Figures 1a and 1b show the top contributions in the flat hiprof profiles from the two compilers. These profiles indicate that the number of calls made to `cse` and `gem_il_peep` in the new version is greater than that of the old one and that these calls are responsible for performance degradation. Figures 2a and 2b show the call graph profiles for `cse` for the two compilers and show the calls made by `cse` and the contributions of each component

called by `cse`. Since these components are included in the GEM back end, the problem was fixed there.

Run-Time Test Cases

For the run-time analysis, we used two different test environments, the Haney kernels benchmark and the NULLSTONE test run against `gcc`.

Haney Kernels The Haney kernels benchmark is a synthetic test written to examine the performance of specific C++ language features. In this run-time test case, an older C++ compiler (version 5.5) was compared with a new compiler under development (version 6.0). The Haney kernels results showed that the version 6.0 development compiler experienced an overall performance regression of 40 percent. We isolated the problem to the real matrix multiplication function. Figure 3 shows the execution profile for this function.

We then used the DCPI tool to analyze performance of the inner loop instructions exercised on version 6.0 and version 5.5 of the C++ compiler. The resulting counts in Figures 4a and 4b show that the version 6.0 development compiler suffered a code scheduling regression. The leftmost column shows the average cycle counts for each instruction executed. The reason for this regression proved to be that a test

Table 1
Ratios of CPU (User and System) Compile Times (Seconds) of the New Compiler to Those of the Old Compiler

File Name	Debug Mode	Default Mode	Optimize Mode
	Options -O0 -g		-O4 -g0
a1amch2	0.970	0.970	0.930
collevo1	0.910	0.780	0.740
d_inh	0.970	0.960	0.960
e_rvirt_yes	0.970	0.980	0.960
interfaceparticle	0.880	0.790	0.730
iostream	0.990	0.980	1.250
pistream	0.890	0.760	0.790
t202	0.970	0.970	1.130
t300	0.980	0.960	1.040
t601	1.010	1.020	1.010
t606	1.000	1.020	1.020
t643	1.020	1.010	1.000
test_complex_excepti	0.960	0.890	0.830
test_complex_math	0.970	0.950	0.950
test_demo	0.950	0.830	0.780
test_generic	1.000	1.020	1.100
test_task_queue6	0.970	0.920	0.960
test_task_rand1	0.950	0.890	0.890
test_vector	0.970	0.920	1.120
vectorf	0.890	0.790	0.850
Averages	0.961	0.920	0.952

```
granularity: cycles; units: seconds; total: 48.96 seconds

%   cumulative      self      self      total
time  seconds  seconds  calls  ms/call  ms/call  name
2.8   1.37      1.37    10195   0.13     0.13    cse [12]
2.6   2.66      1.29    219607  0.01     0.01    gem_il_peep [31]
2.6   3.93      1.27    515566  0.00     0.00    gem_fi_ud_access_resource [67]
2.4   5.09      1.17    481891  0.00     0.00    gem_vm_get_nz [37]
2.3   6.23      1.14    713176  0.00     0.00    _OtsZero [75]
. . .
```

(a) Hiprof Profile Showing Instructions Executed with the New Compiler

```
granularity: cycles; units: seconds; total: 27.49 seconds

%   cumulative      self      self      total
time  seconds  seconds  calls  ms/call  ms/call  name
3.0   0.83      0.83    143483  0.01     0.01    gem_il_peep [40]
2.7   1.58      0.75    614350  0.00     0.00    _OtsZero [64]
2.5   2.26      0.68     8664    0.08     0.08    cse [16]
1.7   2.71      0.45    465634  0.00     0.00    gem_fi_ud_access_resource [86]
1.6   3.14      0.43    423144  0.00     0.00    gem_vm_get_nz [36]
. . .
```

(b) Hiprof Profile Showing Instructions Executed with the Old Compiler

Figure 1
Hiprof Profiles of Compilers

for pointer disambiguation outside the loop code was not performed properly in the version 6.0 compiler. The test would have ensured that the pointers *a* and *t* were not overlapping.

We traced the origin of this regression back to the intermediate code generated by the two compilers. Here we found that the version 6.0 compiler used a more modern form of array address computation in the intermediate language for which the scheduler had not yet been tuned properly. The problem was fixed in the scheduler, and the regression was eliminated.

Initial NULLSTONE Test Run against gcc We measured the performance of the DEC C compiler in compiling the NULLSTONE tests and repeated the performance measurement of the gcc 2.7.2 compiler and libraries on the same tests. Figures 5a and 5b show the results of our tests. This comparison is of interest because gcc is in the public domain and is widely used, being the primary compiler available on the public-domain Linux operating system. Figure 5a shows the tests in which the DEC C compiler performs at least 10 percent better than gcc. Figure 5b indicates the optimiza-

```
[12] 14.1      1.37      5.55    10195+9395    cse [12]
      2.63    134485/134485    test_for_cse [42]
      0.63    134485/134485    update_operands [92]
      0.59    102760/102760    test_for_induction [97]
      0.34    121243/121243    gem_df_move [136]
      0.32    12127/12127     push_effect [149]
. . .
```

(a) Hierarchical Profile for cse with the New Compiler

```
[16] 10.5      0.68      2.19      8664+7593    cse [16]
      1.04    96554/96554    test_for_cse [56]
      0.30    66850/66850    test_for_induction [104]
      0.29    96554/96554    update_operands [106]
      0.12    87176/87176    move [215]
      0.09    7863/7863     pop_effect [267]
. . .
```

(b) Hierarchical Profile for cse with the Old Compiler

Figure 2
Hierarchical Call Graph Profiles for cse

```

void rmatMulHC(Real * t,
  const Real * a,
  const Real * b,
  const int M, const int N, const int K)
{
  int i, j, k;
  Real temp;

  memset(t, 0, M * N * sizeof(Real));

  for (j = 1; j <= N; j++)
  {
    for (k = 1; k <= K; k++)
    {
      temp = b[k - 1 + K * (j - 1)];
      if (temp != 0.0)
      {
        for (i = 1; i <= M; i++)
          t[i - 1 + M * (j - 1)] +=
            temp * a[i - 1 + M * (k - 1)];
      }
    }
  }
}

```

Figure 3
Haney Loop for Real Matrix Multiplication

tion tests in which the DEC C compiler shows 10 percent or more regression compared to gcc.

We investigated the individual regressions by looking at the detailed log of the run and then examining the machine code generated for those test cases. In this case, the alias optimization portion showed that the regressions were caused by the use of an outmoded standard¹⁵ as the default language dialect (`-std0`) for DEC C in the DIGITAL UNIX environment. After we retested with the `-ansi_alias` option, these regressions disappeared.

We also investigated and fixed regressions in instruction combining and if optimizations. Other regressions, which were too difficult to fix within the existing schedule for the current release, were added to the issues list with appropriate priorities.

Conclusions

The measurement and analysis of compiler performance has become an important and demanding field. The increasing complexity of CPU architectures and the addition of new features to languages require the development and implementation of new strategies for testing the performance of C and C++ compilers. By employing enhanced measurement and analysis techniques, tools, and benchmarks, we were able to address these challenges. Our systematic framework for compiler performance measurement, analysis, and prioritization of improvement opportunities should serve as an excellent starting point for the practitioner in a situation in which similar requirements are imposed.

References and Notes

1. D. Blickstein et al., "The GEM Optimizing Compiler System," *Digital Technical Journal*, vol. 4, no. 4 (Special issue, 1992): 121–136.
2. B. Calder, D. Grunwald, and B. Zorn, "Quantifying Behavioral Differences Between C and C++ Programs," *Journal of Programming Languages*, 2 (1994): 313–351.
3. D. Detlefs, A. Dosser, and B. Zorn, "Memory Allocation Costs in Large C and C++ Programs," *Software Practice and Experience*, vol. 24, no. 6 (1994): 527–542.
4. P. Wu and F. Wang, "On the Efficiency and Optimization of C++ Programs," *Software Practice and Experience*, vol. 26, no. 4 (1996): 453–465.
5. A. Itzkowitz and L. Foltan, "Automatic Template Instantiation in DIGITAL C++," *Digital Technical Journal*, vol. 10, no. 1 (this issue, 1998): 22–31.
6. NULLSTONE Optimization Categories, URL: <http://www.nullstone.com/htmls/category.htm>, Nullstone Corporation, 1990–1998.
7. J. Orost, "The Bench++ Benchmark Suite," December 12, 1995. A draft paper is available at http://www.research.att.com/~orost/bench_plus_plus/paper.html.
8. C++ Benchmarks, Comparing Compiler Performance, URL: <http://www.kai.com/index.html>, Kuck and Associates, Inc. (KAI), 1998.
9. *ATOM: User Manual* (Maynard, Mass.: Digital Equipment Corporation, 1995).
10. A. Eustace and A. Srivastava, "ATOM: A Flexible Interface for Building High Performance Program Analysis Tools," Western Research Lab Technical Note TN-44, Digital Equipment Corporation, July 1994.
11. A. Eustace, "Using Atom in Computer Architecture Teaching and Research," *Computer Architecture Technical Committee Newsletter*, IEEE Computer Society, Spring 1995: 28–35.
12. J. Anderson et al., "Continuous Profiling: Where Have All the Cycles Gone?" SRC Technical Note 1997-016, Digital Equipment Corporation, July 1997; also in *ACM Transactions on Computer Systems*, vol. 15, no. 4 (1997): 357–390.
13. J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos, "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," 30th Symposium on Microarchitecture (Micro-30), Raleigh, N.C., December 1997.
14. *Guide to IPROBE, Installing and Using* (Maynard, Mass.: Digital Equipment Corporation, 1994).
15. B. Kernighan and D. Richie, *The C Programming Language* (Englewood Cliffs, N.J.: Prentice-Hall, 1978).

```

rmatMulHC__XPfPCfPCfiii:
. . .
3181 0x120014894 0:88270000 lds $f1, 0(t6)
70 0x120014898 0:a3e70080 ldl zero, 128(t6)
6204 0x12001489c 0:89460000 lds $f10, 0(t5)
3396 0x1200148a0 0:58011041 muls $f0,$f1,$f1
13 0x1200148a4 0:47e60412 bis zero, t5, a2
0 0x1200148a8 0:40a09005 addl t4, 0x4, t4
3058 0x1200148ac 0:20c60010 lda t5, 16(t5)
3157 0x1200148b0 0:40a80db4 cmple t4, t7, a4
0 0x1200148b4 0:20e70010 lda t6, 16(t6)
7265 0x1200148b8 0:59411001 adds $f10,$f1,$f1
12784 0x1200148bc 0:9826fff0 sts $f1, -16(t5)
3207 0x1200148c0 0:8967fff4 lds $f11, -12(t6)
0 0x1200148c4 0:8986fff4 lds $f12, -12(t5)
6604 0x1200148c8 0:580b104b muls $f0,$f11,$f11
13054 0x1200148cc 0:598b100b adds $f12,$f11,$f11
13188 0x1200148d0 0:9966fff4 sts $f11, -12(t5)
3205 0x1200148d4 0:89a7fff8 lds $f13, -8(t6)
0 0x1200148d8 0:89c6fff8 lds $f14, -8(t5)
6388 0x1200148dc 0:580d104d muls $f0,$f13,$f13
12862 0x1200148e0 0:59cd100d adds $f14,$f13,$f13
12687 0x1200148e4 0:99a6fff8 sts $f13, -8(t5)
3134 0x1200148e8 0:89e7fffc lds $f15, -4(t6)
0 0x1200148ec 0:8a06fffc lds $f16, -4(t5)
6357 0x1200148f0 0:580f104f muls $f0,$f15,$f15
12705 0x1200148f4 0:5a0f100f adds $f16,$f15,$f15
12748 0x1200148f8 0:99f2000c sts $f15, 12(a2)
. . .

```

(a) DCPI Profile for This Execution with Version 6.0

```

rmatMulHC__XPfPCfPCfCiCiCi:
. . .
6351 0x1200194d0 0:88270000 lds $f1, 0(t6)
0 0x1200194d4 0:40a09005 addl t4, 0x4, t4
3131 0x1200194d8 0:89460000 lds $f10, 0(t5)
0 0x1200194dc 0:40a80db4 cmple t4, t7, a4
3215 0x1200194e0 0:20e70010 lda t6, 16(t6)
17968 0x1200194e4 0:58011041 muls $f0,$f1,$f1
0 0x1200194e8 0:20c60010 lda t5, 16(t5)
12870 0x1200194ec 0:59411001 adds $f10,$f1,$f1
12727 0x1200194f0 0:9826fff0 sts $f1, -16(t5)
3228 0x1200194f4 0:8967fff4 lds $f11, -12(t6)
0 0x1200194f8 0:8987fff8 lds $f12, -8(t6)
6233 0x1200194fc 0:89a7fffc lds $f13, -4(t6)
3209 0x120019500 0:580b104b muls $f0,$f11,$f11
0 0x120019504 0:89c6fff4 lds $f14, -12(t5)
3127 0x120019508 0:580c104c muls $f0,$f12,$f12
0 0x12001950c 0:89e6fff8 lds $f15, -8(t5)
3174 0x120019510 0:580d104d muls $f0,$f13,$f13
0 0x120019514 0:8a06fffc lds $f16, -4(t5)
6791 0x120019518 0:59cb100b adds $f14,$f11,$f11
3168 0x12001951c 0:59ec100c adds $f15,$f12,$f12
3066 0x120019520 0:5a0d100d adds $f16,$f13,$f13
6258 0x120019524 0:9966fff4 sts $f11, -12(t5)
3134 0x120019528 0:9986fff8 sts $f12, -8(t5)
3200 0x12001952c 0:99a6fffc sts $f13, -4(t5)
3168 0x120019530 0:f69fffe7 bne a4, 0x1200194d0
. . .

```

(b) DCPI Profile with Counts with Version 5.5

Figure 4
DCPI Profiles of the Inner Loop

NULLSTONE SUMMARY PERFORMANCE IMPROVEMENT REPORT			
Nullstone Release 3.9b2			
Threshold: Nullstone Ratio Increased by at least 10%			
	Baseline Compiler	Comparison Compiler	
Compiler	GCC 2.7.2	DEC Alpha C 5.7-123 bl36	
Architecture	DEC Alpha	DEC Alpha	
Model	3000/300	3000/300	
	Optimization	Sample Size	Improvements
	Alias Optimization (by type)	102 tests	6 tests
	Alias Optimization (const-qualified)	11 tests	0 tests
	Alias Optimization (by address)	52 tests	19 tests
	Bitfield Optimization	3 tests	3 tests
	Branch Elimination	15 tests	15 tests
	Instruction Combining	2510 tests	2026 tests
	Constant Folding	56 tests	56 tests
	Constant Propagation	15 tests	8 tests
	CSE Elimination	2600 tests	2353 tests
	Dead Code Elimination	306 tests	278 tests
	Integer Divide Optimization	92 tests	15 tests
	Expression Simplification	181 tests	120 tests
	If Optimization	69 tests	13 tests
	Function Inlining	39 tests	39 tests
	Induction Variable Elimination	4 tests	3 tests
	Strength Reduction	2 tests	1 tests
	Hoisting	38 tests	18 tests
	Loop Unrolling	16 tests	11 tests
	Loop Collapsing	3 tests	3 tests
	Loop Fusion	2 tests	2 tests
	Unswitching	2 tests	1 tests
	Block Merging	1 tests	1 tests
	Cross Jumping	4 tests	2 tests
	Integer Modulus Optimization	92 tests	26 tests
	Integer Multiply Optimization	99 tests	3 tests
	Address Optimization	26 tests	20 tests
	Pointer Optimization	15 tests	9 tests
	Printf Optimization	3 tests	3 tests
	Forward Store	3 tests	3 tests
	Value Range Optimization	30 tests	0 tests
	Tail Recursion	4 tests	2 tests
	Register Allocation	4 tests	1 tests
	Narrowing	3 tests	0 tests
	SPEC Conformance	2 tests	0 tests
	Static Declarations	1 tests	1 tests
	String Optimization	4 tests	4 tests
	Volatile Conformance	90 tests	0 tests
	Total Performance Improvements >= 10%	6499 tests	5065 tests

Figure 5a
 NULLSTONE Results Comparing gcc with DEC C Compiler, Showing All Improvements of Magnitude 10% or More

NULLSTONE SUMMARY PERFORMANCE REGRESSION REPORT			
Nullstone Release 3.9b2			
Threshold: Nullstone Ratio Decreased by at least 10%			
	Baseline Compiler	Comparison Compiler	
Compiler	GCC 2.7.2	DEC Alpha C 5.7-123 b136 no restrict	
Architecture Model	DEC Alpha 3000/300	DEC Alpha 3000/300	
	Optimization	Sample Size	Regressions
Alias Optimization (by type)		102 tests	64 tests
Alias Optimization (const-qualified)		11 tests	9 tests
Alias Optimization (by address)		52 tests	7 tests
Instruction Combining		2510 tests	204 tests
Constant Propagation		15 tests	1 tests
CSE Elimination		2600 tests	32 tests
Integer Divide Optimization		92 tests	32 tests
Expression Simplification		181 tests	34 tests
If Optimization		69 tests	14 tests
Hoisting		38 tests	4 tests
Unswitching		2 tests	1 tests
Integer Modulus Optimization		92 tests	40 tests
Integer Multiply Optimization		99 tests	95 tests
Pointer Optimization		15 tests	1 tests
Tail Recursion		4 tests	2 tests
Narrowing		3 tests	2 tests
Total Performance Regressions >= 10%		6499 tests	542 tests

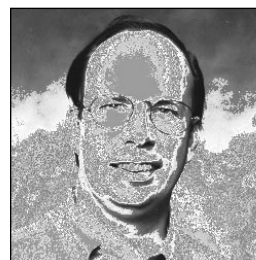
Figure 5b
NULLSTONE Results Comparing gcc with DEC C Compiler, Showing All Regressions of 10% or Worse

Biographies



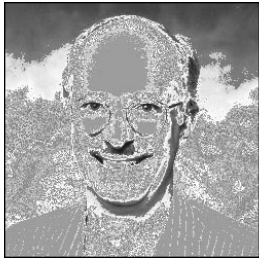
Hemant G. Rotithor

Hemant Rotithor received B. S., M. S., and Ph.D. degrees in electrical engineering in 1979, 1981, and 1989, respectively. He worked on C and C++ compiler performance issues in the Core Technology Group at Digital Equipment Corporation for three years. Prior to that, he was an assistant professor at Worcester Polytechnic Institute and a development engineer at Philips. Hemant is a member of the program committee of The 10th International Conference on Parallel and Distributed Computing and Systems (PDCS '98). He is a senior member of the IEEE and a member of Eta Kappa Nu, Tau Beta Pi, and Sigma Xi. His interests include computer architecture, performance analysis, digital design, and networking. Hemant is currently employed at Intel Corporation.



Kevin W. Harris

Kevin Harris is a consulting software engineer at Compaq, currently working in the DEC C and C++ Development Group. He has 21 years of experience working on high-performance compilers, optimization, and parallel processing. Kevin graduated Phi Beta Kappa in mathematics from the University of Maryland and joined Digital Equipment Corporation after earning an M.S. in computer science from the Pennsylvania State University. He has made major contributions to the DIGITAL Fortran, C, and C++ product families. He holds patents for techniques for exploiting performance of shared memory multiprocessors and register allocation. He is currently responsible for performance issues in the DEC C and DIGITAL C++ product families. He is interested in CPU architecture, compiler design, large- and small-scale parallelism and its exploitation, and software quality issues.



Mark W. Davis

Mark Davis is a senior consulting engineer in the Core Technology Group at Compaq. He is a member of Compaq's GEM Compiler Back End team, focusing on performance issues. He also chairs the DIGITAL Unix Calling Standard Committee. He joined Digital Equipment Corporation in 1991 after working as Director of Compilers at Stardent Computer Corporation. Mark graduated Phi Beta Kappa in mathematics from Amherst College and earned a Ph. D. in computer science from Harvard University. He is co-inventor on a pending patent concerning 64-bit software on OpenVMS.