PROGRAMMING LANGUAGES & TOOLS

Digital Technical Journal

digital™

Volume 10 Number 1
1998

Cover Design
This special issue of the *Journal* focuses on Programming Languages & Tools, specifically on compiler software. For the cover, we have chosen the alchemist who transforms common elements into precious gold to represent the compiler developer who transforms code to extract the highest performance possible for software applications.

The cover was designed by Lucinda O'Neill of the Compaq Industrial and Graphic Design Group.

The *Digital Technical Journal* is a refereed journal published quarterly by Compaq Computer Corporation, 550 King Street, LKG1-2/W7, Littleton, MA 01460-1289.

Hard-copy subscriptions can be ordered by sending a check in U.S. funds (made payable to Compaq Computer Corporation) to the published-by address. General subscription rates are $40.00 (non-U.S. $60) for four issues and $75.00 (non-U.S. $115) for eight issues. University and college professors and Ph.D. students in the electrical engineering and computer science fields receive complimentary subscriptions upon request. Compaq customers may qualify for gift subscriptions and are encouraged to contact their sales representatives.

Electronic subscriptions are available at no charge by accessing URL **http://www.digital.com/subscription**. This service will send an electronic mail notification when a new issue is available on the Internet.

Single copies and back issues can be ordered by sending the requested issue's volume and number and a check for $16.00 (non-U.S. $18) each to the published-by address. Recent issues are also available on the Internet at **http://www.digital.com/dtj**.

Compaq employees may order subscriptions through Readers Choice at URL **http://webrc.das.dec.com.**

Inquiries, address changes, and complimentary subscription orders can be sent to the *Digital Technical Journal* at the published-by address or the electronic mail address, **ctj@compaq.com**. Inquiries can also be made by calling the *Journal* office at 978-506-6858.

Comments on the content of any paper and requests to contact authors are welcomed and may be sent to the managing editor at the published-by or electronic mail address.

ISSN 0898-901X

Documentation Number EC-P9706-18

Book production was done by Quantic Communications, Inc.

December 1998

A letter to readers of the *Digital Technical Journal*

This issue is the last *Digital Technical Journal* to be published. Since 1985, the *Journal* has been privileged to publish information about significant engineering accomplishments for DIGITAL, including standards-setting network and storage technologies, industry-leading VAX systems, record-breaking Alpha microprocessors and semiconductor technologies, and advanced application software and performance tools. The *Journal* has been rewarded by continual growth in the number of readers and by their expressions of appreciation for the quality of content and presentation.

The editors thank the engineers who somehow made the time to write, the engineering managers who supported them, the consulting engineers and professors who reviewed manuscripts and made the process a learning experience for all of us, and, of course, the readers who are the reason the *Journal* came into existence 13 years ago.

With kind regards,

Jane Blake

Jane Blake
Managing Editor

Kathleen Stetson

Kathleen Stetson
Editor

Helen Patterson

Helen Patterson
Editor

Digital Technical Journal Volume 10 Number 1

Contents

Introduction

C. Robert Morgan
Senior Consulting Engineer and Technical Program Manager, Core Technology Group

The complexity of high-performance systems and the need for ever-increased performance to be gained from those systems creates a challenge for engineers, one that requires both experience and innovation in the development of software tools. The papers in this issue of the *Journal* are a few selected examples of the work performed within Compaq and by researchers worldwide to advance the state of the art. In fact, Compaq supports relevant research in programming languages and tools.

Compaq has been developing high-performance tools for more than thirty years, starting with the Fortran compiler for the DIGITAL PDP-10, introduced in 1967. Later compilers and tools for VAX computer systems, introduced in 1977, made the VAX system one of the most usable in history. The compilers and debugger for VAX/VMS are exemplary. With the introduction of the VAX successor in 1992, the 64-bit RISC Alpha systems, Compaq has continued the tradition of developing advanced tools that accelerate application performance and usability for system users. The papers, however, represent not only the work of Compaq engineers but also that of researchers and academics who are working on problems and advanced techniques of interest to Compaq.

The paper on characterization of system workloads by Casmira, Hunter, and Kaeli addresses the capture of basic data needed for the development of tools and high-performance applications. The authors' work focuses on generating accurate profile and trace data on machines running the Windows NT operating system. Profiling describes the point in the program that is most frequently executed. Tracing describes the commonly executed sequence of instructions. In addition to helping developers build more efficient applications, this information assists designers and implementers of future Windows NT systems.

Every compiler consists of two components: the front end, which analyzes the specific language, and the back end, which generates optimized instructions for the target machine. An efficient compiler is a balance of both components. As languages such as C++ evolve, the compiler front end must also evolve to keep pace. C++ has now been standardized, so evolutionary changes will lessen. However, compiler developers must continue to improve front-end techniques for implementing the language to ensure ever better application performance. An important feature of C++ compiler development is C++ templates. Templates may be implemented in multiple ways, with varying effects on application programs. The paper by Itzkowitz and Foltan describes Compaq's efficient implementation of templates. On a related subject, Rotithor, Harris, and Davis describe a systematic approach Compaq has developed for monitoring and improving C++ compiler performance to minimize cost and maximize function and reliability.

Improved optimization techniques for compiler back ends are presented in three papers. In the first of these, Reinig addresses the requirement in an optimizing compiler for an accurate description of the variables and fields that may be changed by an assignment operation, and describes an efficient technique used in the C/C++ compilers for gathering this information. Sweany, Carr, and Huber describe techniques for increasing execution speed in processors like the Alpha that issue multiple instructions simultaneously. The technique reorders the instructions in the program to increase the number of instructions that are simultaneously issued. Maximizing the performance of multiprocessor systems is the subject of the paper by Hall et al., which was previously published in IEEE *Computer* and updated with an addendum for this issue. The authors describe the SUIF compiler, which represents some of the best research in this area and has become the basis of one part of the ARPA compiler infrastructure project. Compaq assisted researchers by providing the DIGITAL Fortran compiler front end and an AlphaServer 8400 system.

As compilers become more effective in increasing application program performance, the ability to debug the programs becomes more difficult. The difficulty arises because the compiler gains efficiency by reordering and eliminating instructions. Consequently, the instructions for an application program are not easily identifiable as part of any particular statement. The debugger cannot always report to the application program where variables are stored or what statement is currently being executed. Application programmers have two choices: Debug an unoptimized version of the program or find some other technique for determining the state of the program. The paper by Brender, Nelson, and Arsenault reports an advanced development project at Compaq to provide techniques for the debugger to discover a more accurate image of the state of the program. These techniques are currently being added to Compaq debuggers.

One of the problems that tool developers face is increasing tool reliability. Tool developers, therefore, test the code. However, developers are often biased; they know how their programs operate, and they test certain aspects of the code but not others. The paper by McKeeman describes a technique called differential testing that generates correct random tests of tools such as compilers. The random nature of the tests removes the developers' bias. The tool can be used for two purposes: to improve existing tools and to compare the reliability of competitive tools.

The High Performance Technical Computing Group and the Core Technology Group within Compaq are pleased to help develop this issue of the *Journal*. Studying the work performed within Compaq and by other researchers worldwide is one way that we remain at the cutting edge of technology of programming language, compiler, and programming tool research.

Bob Morgan

Foreword

William C. Blake
Director, High Performance Technical Computing and Core Technology Groups

You might think that the cover of this issue of the *Digital Technical Journal* is a bit odd. After all, what could be the relevance of those ancient alchemists in the drawing to the computer-age topic of programming languages and tools? Certainly, both alchemists and programmers work busily on new tools. An even more interesting metaphorical connection is the alchemist and the compiler software developer as creators of tools that transform (transmute, in the strict sense of alchemy) the base into the precious. The metaphor does, however, break down. Unlike the myth and folklore of alchemy, the science and technology of compiler software development is a real and important part of processing a new solution or algorithm into the correct and highest performance set of actual machine instructions. This issue of the *Journal* addresses current, state-of-the-art work at Compaq Computer Corporation on programming languages and tools.

Gone are the days when programmers plied their craft "close to the machine," that is, working in detailed machine instructions. Today, system designers and application developers, driven by the pressures of time to market and technical complexity, must express their solutions in terms "close to the programmer" because people think best in ways that are abstract, language dependent, and machine independent. Enhancing the characteristics of an abstract high-level language, however, conflicts with the need for lower level optimizations that make the code run fastest. Computers still require detailed machine instructions, and the high-level programs close to the programmer must be correctly compiled into those instructions. This semantic gap between programming languages and machine instructions is central to the evolution of compilers and to microprocessor architectures as well. The compiler developer's role is to help close the gap by preserving the correctness of the compilation and at the same time resolving the trade-offs between the optimizations needed for improvements "close to the programmer" and those needed "close to the machine."

To put the work described in this *Journal* into context, it is helpful to think about the changes in compiler requirements over the past 15 years. It was in the early 1980s that the direction of future computer architectures changed from increasingly complex instruction sets, CISC, that supported high-level languages to computer architectures with much simpler, reduced instruction sets, RISC. Three key research efforts led the way: the Berkeley RISC processor, the IBM 801 RISC processor, and the Stanford MIPS processor. All three approaches dramatically reduced the instruction set and increased the clock rate. The RISC approach promised improvements up to a factor of five compared with CISC machines using the same manufacturing technology. Compaq's transition from the VAX to the Alpha 64-bit RISC architecture was a direct result of the new architectural trend.

As a consequence of these major architectural changes, compilers and their associated tools became significantly more important. New, much more complex compilers for RISC machines eliminated the need for the large, microcoded CISC machines. The complexities of high-level language processing moved from the petrified software of CISC microprocessors to a whole new generation of optimizing compilers. This move caused some to claim that RISC really stands for "Relegate Important Stuff to Compilers."

The introduction of the third-generation Alpha microprocessor, the 21264, demonstrates that the shift to RISC and Alpha system implementations and compilers served Compaq customers well by producing reliable, accurate, and high-performance computers. In fact, Alpha systems, which have the ability to process over a billion 64-bit floating-point numbers per second, perform at levels formerly attained only by specialized supercomputers. It is not surprising that the Alpha microprocessor is the most frequently used microprocessor in the top 500 largest supercomputing sites in the world.

After reading through the papers in this issue, you may wonder what is next for compilers and tools. As physical limits curtail the shrinking of silicon feature sizes, there is not likely to be a repeat of the performance gains at the microprocessor level, so attention will turn to compiler technology and computer architecture to deliver the next thousandfold increase in sustained application performance. The two principal laws that affect dramatic application performance improvements are Moore's Law and Amdahl's Law. Moore's Law states that performance will double each 18 months due to semiconductor process scaling; and Amdahl's Law expresses the diminishing returns of various system speedup enhancements. In the next 15 years, Moore's Law may be stopped by the physical realities of scaling limits. But Amdahl's Law will be broken as well, as improvements in parallel language, tool development, and new methods of achieving parallelism will positively affect the future of compilers and hence application performance. As you will see in papers in this issue, there is a new emphasis on increasing execution speed by exploiting the multiple instruction issue capability of Alpha microprocessors. Improvements in execution speed will accelerate dramatically as future compilers exploit performance improvement techniques using new capabilities evolved in Alpha. Compilers will deliver new ways of hiding instruction latency (reducing the performance gap between vector processors and RISC superscalar machines), improved unrolling and optimization of loops, instruction reordering and scheduling, and ways of dealing with parallel decomposition and data layout in nonuniform memory architectures. The challenges to compiler and tool developers will undoubtedly increase over time.

By not relying on hardware improvements to deliver all the increases in performance, compiler wizards are making their own contributions — always watchful of correctness first, then run-time performance, and, finally, speed and efficiency of the software development process itself.

Bill Blake

Tracing and Characterization of Windows NT–based System Workloads

Jason P. Casmira
David P. Hunter
David R. Kaeli

To optimize the design of pipelines, branch predictors, and cache memories, computer architects study the characteristics of benchmark programs by examining traces, i.e., samples of program execution. Since commercial desktop applications are increasingly dependent on services and application programming interfaces provided by the host operating system, the authors argue that traces from benchmark execution must capture operating system execution in addition to native application execution. Common benchmark-based workloads, however, lack operating system execution. This paper discusses the ongoing joint efforts of the Northeastern University Computer Architecture Research Laboratory and Compaq Computer Corporation's Advanced and Emerging Technologies Advanced Development Group to capture operating system–rich traces on Alpha-based machines running the Windows NT operating system. The authors describe the latest PatchWrx software toolset and demonstrate its trace-generating capabilities by characterizing numerous applications. Included is a discussion of the fundamental differences between using traces captured from common benchmark programs and using those captured on commercial desktop applications. The data presented demonstrates that operating system execution can dominate the overall execution time of desktop applications such as Microsoft Word, Microsoft Visual C/C++, and Microsoft Internet Explorer and that the characteristics of the operating system instruction stream can be quite different from those typically found in benchmarking workloads.

The computer architecture research community commonly uses trace-driven simulation in pursuing answers to a variety of design issues. Architects spend a significant amount of time studying the characteristics of benchmark programs by examining traces, i.e., samples taken from program execution. Popular benchmark programs include the SPEC[1] and the BYTEmark[2] benchmark test suites. Since the underlying assumption is that these programs generate workloads that represent user applications, today's computer designs have been optimized based on the characteristics of these benchmark programs.

Although the authors of popular benchmarks are well intentioned, the resulting workloads lack operating system execution and consequently do not represent some of the most prevalent desktop applications, e.g., Microsoft Word, Microsoft Visual C/C++, and Microsoft Internet Explorer. Such applications make heavy use of application programming interfaces (APIs), which in turn execute many instructions in the operating system. As a result, the overall performance of many desktop applications depends on efficient operating system interaction. Clearly operating system overhead can greatly reduce the benefits of a new computer design feature. Past architectural studies, however, have generally ignored operating system interaction because few tools can generate operating system–rich traces.

This paper discusses the ongoing joint efforts of Northeastern University and Compaq Computer Corporation to capture operating system–rich traces on DIGITAL Alpha-based machines running the Microsoft Windows NT operating system. We argue that for traces of today's workloads to be accurate, they must capture the operating system execution as well as the native application execution. This need to capture complete program trace information has been a driving force behind the development and use of software tools such as the PatchWrx dynamic execution-tracing toolset, which we describe in this paper.

The PatchWrx toolset was originally developed by Sites and Perl at Digital Equipment Corporation's Systems Research Center. They described PatchWrx, as developed for Windows NT version 3.5, in "Studies of

Windows NT Performance Using Dynamic Execution Traces."[3] The Northeastern University Computer Architecture Research Laboratory and Compaq's Advanced and Emerging Technologies Advanced Development Group continue to develop the toolset. We have updated the framework to operate under Windows NT version 4.0, added the ability to trace programs that have code sections larger than 4 megabytes (MB), added multiple trace buffer sizes, and developed additional postprocessing tools.

After briefly discussing related tracing tools, we describe the PatchWrx toolset and specify the new features we have added. We then analyze PatchWrx traces captured on Windows NT version 4.0, demonstrating the capabilities of the tool while illustrating the importance of capturing operating system–rich traces. In the final section, we summarize the paper, discuss the current limitations of the toolset, and suggest new directions for development and study.

Trace Generation Tools

Trace-driven simulation has been the method of choice for evaluating the merits of various architectural trade-offs.[4,5] Traces captured from the system under test are recorded and replayed through a model of the proposed design. Computer architecture researchers have proposed methodologies that capture both application and operating system references. These tools include hardware-based[6–10] and software-based[11–15] methods. Some of the issues involved in capturing operating system–rich traces are

1. Tracing overhead (system slowdown)
2. Accuracy (perturbation of the memory address space)
3. Completeness (capturing all desired information, e.g., the operating system reference stream)

Table 1 contains a list of 10 tracing tools that have been developed over the past 10 to 15 years. Although far from complete, this list provides a sample of the tools that have been used to generate input to a variety of trace-driven simulation studies. We have characterized each tool in terms of the three issues (criteria) previously mentioned. Table 1 lists the target platform(s) for each tracing tool.

Note that many of these tools cannot capture operating system activity. For those that can, their associated slowdown can significantly affect the accuracy of the captured trace. Of the tools that provide this capability, PatchWrx introduces the least amount of slowdown yet maintains the integrity of the address space. The next section discusses the PatchWrx toolset.

PatchWrx

PatchWrx is a dynamic execution-tracing toolset developed for use on the Alpha-based Microsoft Windows NT operating system. The toolset utilizes the Privileged Architecture Library (PAL) facility, also referred to as PALcode, of the Alpha microprocessor to perform tracing with minimal overhead.[22] PatchWrx can instrument, i.e., *patch,* all Windows NT application and system binary images, including the kernel, operating system services, drivers, and shared libraries. The PAL facility is a set of architected functions and instructions that provides a consistent interface to a set of complex system functions. These routines provide primitives for memory management, context switching, interrupts, and exceptions.

PatchWrx and the Alpha PAL Routines

The PatchWrx software tool is made possible through the PAL used by DIGITAL Alpha microprocessors. PAL routines have access to physical memory and internal hardware registers and operate with interrupts disabled. PALcode is loaded from disk at system boot time. We modified and extended the shrink-wrapped Alpha PALcode on a DIGITAL Alpha 21064–based system to support the PatchWrx operations. The mod-

Table 1
Sample of Tracing Tools

Name	Average Slowdown	Address Perturbation	Operating System Activity	Platform
ATOM[13]	10X to 100X	No	Yes	DIGITAL Alpha UNIX
ATUM[16]	20X	No	Yes	DIGITAL VAX OpenVMS
EEL[17]	10X to 100X	Yes	No	SPARC Solaris
Etch[18]	35X	Yes	No	Intel x86 Microsoft Windows NT V4.0
NT-Atom[19]	10X to 100X	No	No	DIGITAL Alpha Microsoft Windows NT V4.0
PatchWrx[3]	4X	No	Yes	DIGITAL Alpha Microsoft Windows NT V4.0
Pixie[20]	10X to 100X	Yes	No	DIGITAL MIPS ULTRIX
QPT[12]	10X to 100X	Yes	No	SPARC Solaris, DIGITAL ULTRIX
Shade[21]	6X	No	No	SPARC Solaris
SimOS[14]	10X to 50,000X	No	Yes	DIGITAL Alpha UNIX, SGI IRIX, SPARC Solaris

ified PatchWrx PAL routines serve two major purposes: (1) to reserve the trace buffer at system boot time and (2) to log trace entries at trace time.

One way that PatchWrx maintains a low operating overhead is to store the captured trace in a physical memory buffer, which is reserved at boot time. The size of the buffer can be varied depending on the amount of physical memory installed on the system. Since we use PAL routines to reserve this memory, the operating system is not aware that the memory exists because the PALcode performs all low-level system initialization before the operating system is started.

PatchWrx logs all trace entries in this buffer. Writing trace entries directly to physical memory has several advantages. First, writing to memory is much faster than writing to disk or to tape. Second, using physical memory allows tracing of the lowest levels of the operating system (i.e., the page fault handler) without generating page faults. Third, using physical memory allows tracing across multiple threads running in multiple address spaces regardless of which address space is currently running.

To enable PatchWrx to operate under Windows NT versions 3.51 and 4.0, we started with the PAL routines modified by Sites and Perl[3] and made additional modifications as required by the operating system versions. These modifications were concentrated in the process data structures. The PatchWrx-specific PAL routines are listed in Table 2. The first three routines are used for reading the trace entries from the buffer and for turning tracing on and off. The remaining five routines are used to log trace entries based on the type of instruction instrumented.

PatchWrx Image Instrumentation

Next we describe how we use PatchWrx to instrument Microsoft Windows NT images. Patching the operating system involves the instrumentation of all the binary images, including applications, operating system executables, libraries, and kernel. Once patching is complete, trace entries are logged by means of PAL routines as images execute.

We define a patched instruction as an instruction within an image's code section that is overwritten with an unconditional branch (BR) to a patch. The target of the BR contains the *patch section*. The patch section includes the trap (CALL_PAL) to the appropriate PAL routine that logs a trace entry corresponding to the type of instruction patched and the return branch to the original target.

PatchWrx does not modify the original binary images; instead, it generates new images that contain patches. This operation preserves the original images on the system in case they need to be restored. Instrumentation involves replacing all branching instructions of type unconditional branch, conditional branch (e.g., branch if equal to zero [BEQ]), branch to subroutine (BSR), function return (RET), jump (JMP), and jump to subroutine (JSR) within an image's code section with unconditional branches to a patch section. If loads and stores are also traced, PatchWrx replaces these instructions (e.g., load sign-extended longword [LDL]) with unconditional branches to the patch section, where the original load or store instruction is copied. A return branch is also needed to return control flow to the instruction subsequent to the original load. When PatchWrx encounters this patch, the tool records the register value of the original load or store instruction in the trace log. The patch section contains all the patches for the image and is added to the rewritten image. Figure 1 shows examples of patched instructions. PatchWrx replaces only branch instructions within an image to reduce the type and number of entries logged in the trace buffer. Using these traced branches, the tool can later reconstruct the basic blocks they represent.

As shown in Figure 1, PatchWrx replaces BR and JMP instructions with BR instructions that transfer control to the patch section. The original BR or JMP instruction is repeated in the patch section for the purpose of recording the value of the target register (if necessary) into the trace buffer when the patched image is executed. This register value is necessary for reconstructing the traced instruction stream. PatchWrx

Table 2
PatchWrx-specific PAL Routines

PAL Routines	Function
PWRDENT	Read a trace entry from trace memory
PWPEEK	Read an arbitrary location (for debug)
PWCTRL	Initialize, turn tracing on/off
PWBSR	Record a branch to subroutine
PWJSR	Record a jump/call/return
PWLDST	Record a load/store base register value
PWBRT	Record a conditional branch taken bit
PWBRF	Record a conditional branch fall-through bit

	ORIGINAL CODE	PATCHED CODE	
EXAMPLE 1	⋮	⋮	
	JMP ZERO,(R19)	~~JMP ZERO,(R19)~~ BR PATCH.001	
	⋮	⋮	
		PATCH.001:	CALL_PAL PWJSR
			JMP ZERO,(R19)
EXAMPLE 2	⋮	⋮	
	JSR R26,(R19)	~~JSR R26,(R19)~~ BSR R26,PATCH.002	
	⋮	⋮	
		PATCH.002:	CALL_PAL PWJSR
			JMP ZERO,(R19)
EXAMPLE 3	⋮	⋮	
	BEQ R3,TARGET.003	~~BEQ R3,TARGET.002~~ BR PATCH.003	
	⋮	BACK.003	
		⋮	
		PATCH.003:	BEQ R2,PATCH.003T
			CALL_PAL PWBRF
			BR BACK.003
		PATCH.003T:	CALL_PAL PWBRT
			BR TARGET.003
EXAMPLE 4	⋮	⋮	
	LDL R20,4(R16)	~~LDL R20,4(R16)~~ BR PATCH.004	
	⋮	BACK.004	
		⋮	
		PATCH.004:	CALL_PAL PWLDST
			LDL R20,4(R16)
			BR BACK.004

Figure 1
Instruction Patch Examples

replaces JSR and BSR instructions with BSR patches. This replacement preserves the return address (RA) register field value, which contains the return address for the subroutine. Again, the original instruction is repeated in the patch section for register value recording during tracing to help facilitate reconstruction.

Conditional branches have a larger and more complex patch than the other branch types because the original condition is duplicated and resolved within the patch. The taken or fall-through path generates a bit value when logged within the taken or fall-through trace entry. The return branch in the patch section is a replica of the original conditional branch.

As explained earlier, for all patches, PatchWrx replaces the original branch with a patch unconditional branch. Since Alpha instructions are equal in size, this replacement process allows patching without increasing the code size within the image. Although the code size remains unchanged, the image size will increase in proportion to the number of patches added. This image size change becomes an issue for dynamically linked library (DLL) images.

Patching Dynamic Link Libraries

The Microsoft Windows NT operating system provides a memory management system that allows sharing between processes.[23] For example, two processes that edit text files can share the text editor application image that has been mapped into memory. When the first process invokes the editor, the operating system loads the application into memory and maps the process's virtual address space to it. When the second process invokes the editor, rather than load another editor image, the operating system maps the second process's virtual address space to the physical pages that contain the editor. Of course, both processes contain local storage for private data.

DLLs are loaded into memory and shared in this manner. When patches are added to a DLL, the size of the image increases. When this image is mapped to physical memory (as per its preferred base load address), the larger image may overlap with another image having a base address within the new range. This image overlap can prevent the operating system from booting properly: some environment DLLs will conflict in memory because they perform calls directly into other DLLs at fixed offsets. To resolve this issue, we *rebase*[24] the preferred base load addresses of the patched DLLs, which modifies the base load addresses of each patched DLL to eliminate conflicts. Rebasing affects the address accuracy of the patched system, though we are able to readjust the addresses during reconstruction. An increase in the paging activity may also be observed since the additional code may cross page boundaries.

The original version of the PatchWrx toolset was developed on Microsoft Windows NT version 3.5. When versions 3.51 and 4.0 were released, several modifications were made to the image format. In completing the 3.51- and 4.0-compatible versions of PatchWrx, we had to address this issue. One change that affected how we patch was the placement of the Import Address Table (IAT) into the front of the initial code section of executable binary images. This table is used to look up the addresses of DLL procedures used (i.e., imported) by the executable binary. In developing the current generation of PatchWrx, we had to make modifications to use image header fields that had previously remained unused or reserved, indicating the executable code sections that contained data areas.

Another issue that we addressed in the recent modifications to PatchWrx was long branches. The original version of PatchWrx replaces all branch, jump, call, and return instructions with either BR or BSR instructions to the patch section. Since the PatchWrx tool has no information about machine state during the patching phase, it is impossible to utilize other branching instructions (e.g., JMP or JSR instructions) to provide this branch-to-patch transition. Register and register-indirect branching instructions would require perturbing the machine state. Therefore, the developers could use only program counter (PC)–based offset branching instructions.

As discussed previously, in replacing a control flow instruction with a patch branch, PatchWrx uses a BR or BSR instruction in which the offset field is set to branch to the corresponding patch within the image's patch section. The Alpha architecture branching instructions use the format shown in Figure 2.

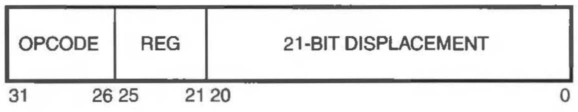


Figure 2
Alpha Branch Instruction Format

The branch target virtual address computation for this format is newPC = (oldPC + 4) + (4 * sign-extended(21-bit branch displacement)). The register field holds the return address for BSRs. With this branch format and target virtual address computation, the Alpha architecture provides a branch target range of 4 MB from an instruction's current PC.

Several applications that run today on Microsoft Windows NT version 4.0 are sufficiently large that the displacement between a control flow instruction to be patched and the patch location within the patch section exceeds this 4-MB limit. (Recall that since we want to avoid moving code or data sections, the patch section is placed at the end of the image.) To address this problem, we developed two new branch instructions for use with PatchWrx. These new branches were not implemented in the instruction set architecture of the Alpha architecture. Instead, we used PALcode to implement them. The two new branches are designated long branch (LBR) and long branch subroutine (LBSR). Figure 3 illustrates the format of these two instructions.

The computation of the target virtual address is newPC = (oldPC + 4) + (4 * sign-extended(25-bit branch displacement)) for LBR branches and newPC = (oldPC + 4) + (32 * zero-extended(20-bit branch displacement)) for LBSR branches. PatchWrx uses LBRs when patching any control flow instruction that has a displacement greater than 4 MB. PatchWrx uses LBSRs similarly for control flow instructions that must preserve the register field value.

When an LBR or LBSR instruction is executed within the image code section, a trap to PALcode occurs. Normally, CALL_PAL instructions have one of several defined function fields that cause a corresponding PAL routine to be executed. The two long branch instructions have function fields that do not belong to any of the defined CALL_PAL instructions and therefore force an illegal instruction exception within the PALcode. This PALcode flow has been modified to detect if a long branch has been encountered.

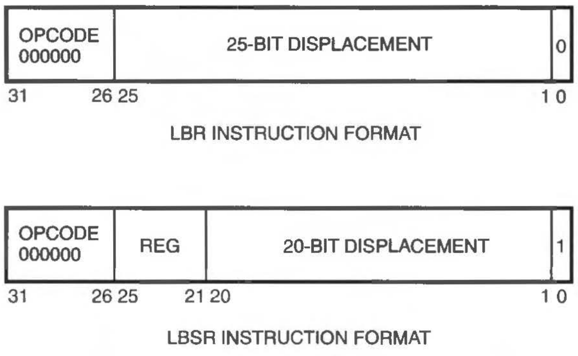


Figure 3
PALcode Long Branch Instruction Formats

As shown in Figure 3, both long branch types have the same PALcode operation code (opcode) value of 000000. To distinguish between the two types, the least significant bit in the instruction word is set to 0 for LBRs and to 1 for LBSRs. This bit is not included as a usable bit for the displacement fields of either branch type. Consequently, each LBR has a 25-bit displacement field and each LBSR has a 20-bit field. With a 25-bit usable displacement field, the PALcode performs the LBR target address computation, allowing a ±64-MB range.

Since each LBSR instruction has a 20-bit displacement field, whereas the original Alpha architecture branch displacement field is 21 bits, the target instruction address computation for LBSR instructions is performed differently than for standard branches within the PALcode. As shown in the address computation equation, the 20-bit displacement is multiplied by 32 rather than by 4 (as for the LBR branch). Notice that the 20-bit displacement is always zero extended. The computation provides the LBSR instruction with a displacement of +32 MB.

This computation procedure has two implications. First, LBSR instructions can only be used to branch from an image code section to an image's patch section. Second, branches into the patch section are either BR or BSR instructions (or their long displacement counterparts). PatchWrx uses only BR or LBR instructions to return from the patch section to the original branch target within a code section; BSR and LBSR instructions are never used. Therefore, restricting LBSR instructions to use positive displacements does not present a problem.

The LBSR displacement multiplier value of 32 does present some restrictions, however. The multiplier value of 4 used in the original Alpha instruction set architecture represents the instruction word length of 4 bytes. Thus, normal branch instruction target addresses must be aligned on a 4-byte boundary. By using the multiplier value of 32 for LBSR instructions, LBSR target addresses are restricted to align on a 32-byte (i.e., eight-instruction) boundary. Since all LBSR targets reside within the patch section, this restriction does not pose a problem. If an LBSR is to be inserted into the image code section and the next available patch target address is not aligned properly, PatchWrx can insert no operation (NOP) instruction words and advance the next available patch target address until the necessary alignment is achieved. PatchWrx never executes the NOPs; they are inserted for alignment purposes only. Although inserting these NOP instructions increases the image size, we have implemented several optimizations into the instrumentation algorithm to minimize this increase. For example, a queue is used to hold LBSRs that do not align. As LBR patches are committed, PatchWrx probes the queue to determine if any LBSRs align from their origin to the newly available patch target offset.

Trace Capture

The PatchWrx toolset allows the user to turn tracing on and off and thus capture any portion of workload execution. The tracing tool is also responsible for copying trace entries from the physical memory buffer to disk. Copying the trace buffer to disk is performed after tracing has stopped so that the time required to perform the copy does not introduce any overhead during trace capture.

PatchWrx logs a trace entry for each patch encountered during program execution. As it executes instructions within the code section, PatchWrx encounters an unconditional PatchWrx branch. Instead of branching to the original target, the patched branch transfers control to the image's patch section. Within the patch section, a PatchWrx PALcall traps to the PAL routine corresponding to the patch type and logs a trace entry to the trace buffer. The PAL routine then returns to the instruction following the CALL_PAL instruction. PatchWrx uses an unconditional branch to transfer control from the patch section back to the original target within an image code section. During the execution of the PatchWrx PAL routine, necessary machine state information is recorded and logged in the trace buffer. This allows for the capture of register contents, process ID information, etc., which are used later during trace reconstruction.

The trace capture facility captures the dynamic execution of a workload running on the system. To reconstruct the trace after it has been captured, the tracing tool must also capture a snapshot of the base load addresses of all active images on the system. This snapshot serves as the virtual address map used in reconstructing the trace. Each active process and its associated libraries is loaded into a separate address space, which may be different than the preferred load address as specified statically in the image header. If each image was loaded into memory at its preferred base address, the virtual address map would not be necessary to perform reconstruction. Instead, PatchWrx could map target addresses from the trace buffer using the base address values contained in the static image headers.

The type of trace record that PatchWrx logs into the trace buffer depends on the type of branch or low-level PAL function being traced. Figure 4 shows the trace record formats. The first three trace entry formats consist of an 8-bit opcode and a 24-bit time stamp. The time stamp is the low-order 24 bits of the CPU cycle counter. The 32-bit field of these three formats depends on the type of trace entry logged. The first format is used for target virtual addresses for all unconditional direct and indirect branches, jumps, calls, returns, interrupts, and returns from interrupts. The 32-bit field of the second format is used to record the base register value for traced load and store instructions and stack pointer values that are flushed into the trace buffer during system calls and returns. The 32-bit field of the third format is used for logging the current active process ID at a context swap.

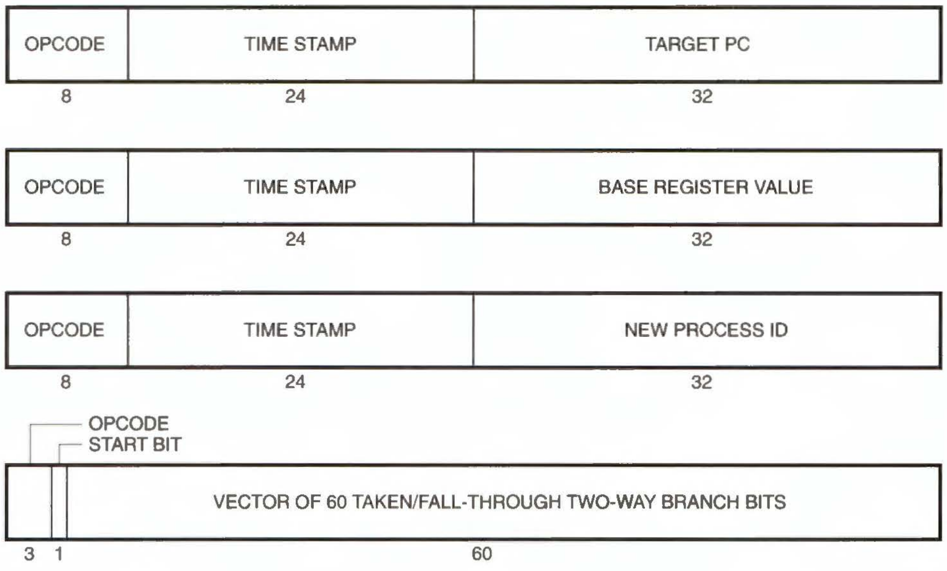


Figure 4
Trace Entry Formats

The fourth trace entry type is used for tracing conditional branches. It uses a 3-bit opcode and up to 60 taken/fall-through bits. A start bit is used to determine how many bits are active. The start bit is set to 1 if a conditional branch is taken and to 0 if the branch is not taken. This recording scheme allows a compact encoding of conditional branch trace entries. During trace reconstruction, PatchWrx uses conditional branch trace entries to reconstruct the correct instruction flow when conditional branches are encountered and to provide concise information about when to deliver interrupts in loops.

Trace Reconstruction

The reconstruction phase is the final step in generating a full instruction stream of traced system activity. As shown in Figure 5, trace reconstruction requires several resources in order to generate an accurate instruction stream of all traced system activity.

Trace reconstruction reads and initializes the heading of the captured trace, which includes a time stamp, the name of the user who captured the trace, and any important system configuration information, e.g., the operating system version number. Next, reconstruction reads the first four raw trace records, which are automatically entered whenever tracing is turned on. These records contain the first target virtual address, the active process ID, the value of the stack pointer, and the first taken/fall-through record to be used (such records always precede the branches they represent). PatchWrx uses this information to initialize the necessary data structures of the reconstruction process.

Using the first target virtual address and process ID pair from the captured trace, trace reconstruction consults the virtual address map to determine in which image the instruction falls (based on its dynamic base load address) and where that image is physically located on the system. The tool consults the patched image to determine the actual instruction at the target address, records this instruction, and then reads the next instruction from the patched image. This process continues until reconstruction encounters either a conditional branch or an unconditional branch. A conditional branch causes the tool to check the first active bit of the current taken/fall-through entry to determine subsequent control flow; the process then continues at that address. If an unconditional branch is encountered, reconstruction records the entry and checks it against the next captured trace entry. If the two entries match, the tool outputs the recorded instructions to an instruction stream file, consults the captured trace entry for the next target instruction virtual address, and repeats the procedure until the entire captured trace has been processed.

Since PatchWrx captures interrupts and other low-level system activities (e.g., page faults) in the trace, these activities must also be reconstructed. When PatchWrx logs an interrupt into the trace buffer, the corresponding target virtual address in the captured record represents the address of the first instruction *not* executed when the interrupt was taken. PatchWrx flushes the currently active taken/fall-through entry to the memory buffer and initializes a new taken/fall-through entry. This new entry will be responsible for

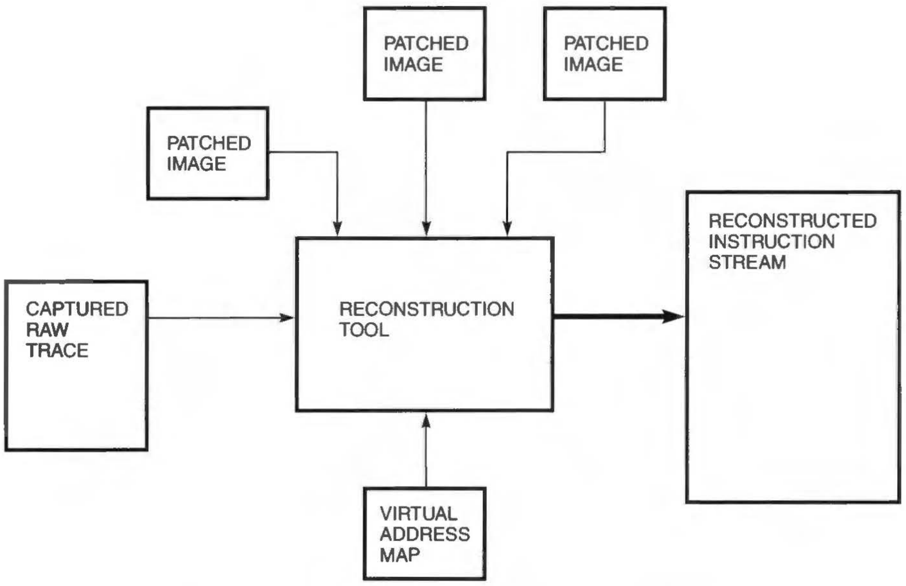


Figure 5
Instruction Stream Reconstruction Resources

the conditional branches encountered beginning with the interrupt service routine. The address of the first instruction within the interrupt service routine is then logged in the trace.

During reconstruction, the reconstruction tool looks for the interrupt's first unexecuted instruction address to know which instruction to stop at when reconstructing the instruction stream. The tool then begins reconstructing the instruction stream, including the interrupt handler stream. If the unexecuted instruction is within a loop, trace reconstruction utilizes the taken/fall-through entry convention. On taking the interrupt, the active taken/fall-through record is flushed and another record is started. This process allows the tool to continue to reconstruct iterations of the loop until all the taken/fall-through bits are exhausted.

Operating System–Rich Workload Characterization

As presented in the study by Lee et al.,[18] desktop applications and benchmarks share some workload characteristics, but applications alone do not represent full system behavior. To investigate and address system design issues, computer architects should use operating system–rich traces.

To illustrate this point, we present a sample of the various workload characteristics that exist in a set of benchmark and desktop applications specially selected to study the differences in the use of the operating system and related services. The first characteristic we discuss is the amount of time each benchmark or desktop application spends within three domains:

1. Application-only domain (e.g., winword.exe and excel.exe)
2. DLL domain—Win32 user (e.g., kernel32.dll, user32.dll, and ntdll.dll)
3. Operating system domain—Win32 kernel, kernel, system processes, system idle process (e.g., Win32K.sys, ntoskrnl.exe, drivers, and the spooler)

Examining these times provides insight into a workload's use of each domain. We also examine DLL and system service usage on an image basis for each workload. This breakdown helps us more clearly identify the dependence between the workload and the system services provided by the Windows NT operating system.

We also present the instruction mix of each workload with and without the inclusion of the operating system execution. Understanding the differences in instruction composition in the presence of system activity further highlights the behavior lacking in application-only traces, such as increases in branch and memory instructions, when compared to application-only workloads. We present the average basic block lengths for each domain of execution (application-only, DLL, operating system) separately and then in combination. This metric reveals which workload domain dominates the branching behavior. Casmira's work provides a more complete description of these differences across a wider set of workload characteristics.[25]

Workload Descriptions

We performed all the experiments reported on in this paper on a DIGITAL Alpha platform running the Microsoft Windows NT version 4.0 operating system. We captured the traces on a 150-megahertz Alpha 21064 processor. The system configuration included 80 MB of physical memory. Table 3 lists the workloads we examined.

Table 3
Workload Description

Workload	Description
fourier	BYTEmark benchmark; a numerical analysis routine for calculating series approximations of waveforms
neural	BYTEmark benchmark; a small, functional back-propagation network simulator
go	SPEC95 *Go!* game benchmark
li	SPEC95 Lisp interpreter benchmark
cdplay	Microsoft CD Player playing a music CD
fx!32	DIGITAL FX!32 V1.1 interpreting/translating included OpenGL sample x86 application
ie	Microsoft Internet Explorer V2.0 following a series of web page links
vc50	Microsoft Visual C/C++ V5.0 compiling a 3,000-line C program
word	Microsoft Word97 V7.0, spell-checking a 15-page document

The fourier and neural workloads are from the BYTEmark benchmark test suite: the neural workload is a small array-based floating-point test; the fourier workload is designed to measure transcendental and trigonometric floating-point unit performance.

The go and li workloads are from the SPEC95 integer benchmark suite: the go workload is a simulation of the game *Go!*, with the computer playing against itself; the li workload is a Lisp interpreter. All the workloads use the standard inputs provided with the benchmarks and are compiled with the default optimization level using the native Alpha version of Microsoft C/C++ version 5.0.

The cdplay workload is the Microsoft CD Player application included in Microsoft Windows NT version 4.0. The device was traced while playing a music CD using default playing options (e.g., playing all the songs in order).

The fx!32 workload is the DIGITAL FX!32 version 1.1 emulator/translator provided by Compaq's DIGITAL Alpha Migration Tools Group.[26] We ran the robot arm OpenGL sample Intel-based application in the foreground during trace capture.

The ie workload is the standard Microsoft Internet Explorer version 2.0 workload included in Microsoft Windows NT version 4.0. The ie workload was traced while traversing four links through the Sony home web page, arriving finally at the Sony PlayStation Store web page. The trace was captured on May 4, 1998; pages may have changed since this date. The history cache and the web link cache were both empty when the trace was captured.

The vc50 workload is the Microsoft C/C++ version 5.0 compiler compiling a 3,000-line C source code file. We used the command line interface, and we used the default optimization levels and other parameters, which best represented the common usage of the compiler.

The word workload is Microsoft Word from the Microsoft Office97 desktop application suite for the Alpha processor used to capture a manual spell check of a 15-page Microsoft Word document. The standard Microsoft Word dictionary was employed.

To provide a clear and representative comparison of workload behavior, we captured several traces. For all scenarios, full traces of each workload captured approximately 5 to 10 seconds of execution, filling the 45-MB trace buffer. To characterize workload behavior, each experiment was run with the benchmark or application as the only activity on the system. Each workload was run in the foreground.

To ensure that the traces captured were representative of the overall workload behavior, we captured multiple traces. We chose different points during execution for tracing to allow comparison between different portions of the selected scenarios. To investigate the variability present in selected workloads, we traced additional scenarios. A second Microsoft Word trace was captured with the application performing an autoformat operation of the same document used in the first trace of the spell-check operation, and we captured a second Microsoft Internet Explorer trace, repeating the Sony links but with the links cached. We captured a second trace of FX!32 using the included *boggle* sample game (for comparison against using the OpenGL application input). Additionally, the FX!32 translator was traced while it optimized a native Intel x86 application's profile. To condense the number of memory pages occupied by an image, Microsoft designed the new linker to allow data to reside within the code regions. Hookway and Herdeg[26] provide an explanation of the DIGITAL FX!32 emulation and translation/optimization procedures. Casmira discusses these scenarios and others.[25]

Domain Mix

To illustrate the inherent differences between benchmark and desktop application behavior, we break down the captured trace in terms of three mutually exclusive domains. These domains are (1) application, (2) DLL, and (3) operating system. The application domain represents the set of executed instructions that are within the traced application's executable image.

The DLL domain represents the instructions executed by the application of interest's process but excludes the application's executable image. This domain is made up of the DLLs, system services, and drivers that the application may access during execution. The operating system domain includes instructions executed by the kernel or other system support service executable images, and all associated DLL and driver images. These are the processes, images, and libraries that are always present and running on the system. Figure 6 displays the breakdown of instructions into these three domains. The x-axis lists the workloads, and the y-axis presents the percent composition of the captured trace. Note that the four benchmarks, i.e., fourier, neural, go, li, spend at least 95 percent of their execution within their application image. Both the fourier and the neural benchmarks spend about 99 percent of their execution within their application image. The go and li benchmarks do exhibit some operating system activity, but this activity is due to the I/O generated as go displays output as it progresses and as li reads input from its standard input file.

The operating system dominates the execution in the cdplay workload. The Microsoft CD Player application is I/O bound, relying heavily on the necessary services provided by the operating system and the DLLs to access the CD hardware. While waiting for I/Os to complete, the system activity is composed almost completely of the kernel idle loop performing busy waiting (recall that each workload investigated is the only application running on the system, so there is no other work to be done during these periods).

The fx!32 workload spends nearly all its execution time operating within DLLs. The robot arm Intel x86 OpenGL sample that the DIGITAL FX!32 application is interpreting heavily exercises the graphics display libraries and console display services.

The ie workload is more evenly distributed across the three domains. The moderate amount of operating system activity is due to the network and screen display I/O and also to the Microsoft Internet Explorer's caching of the pages it touches to local disk files. The DLL activity is generated by operating system services for screen and file I/O and by network service library routines. The application image coordinates the usage of these routines, and network and display I/O, which is frequently encountered during the operations of selecting and opening web links. This coordination accounts for the high percentage of application domain execution exhibited by ie, as shown in Figure 6.

The vc50 workload spends nearly all its execution time within its application image. This phase of the compiler is responsible for performing the parsing and lexical analysis of the source code file. There is some use of DLLs through invoking library routines to load included header files. The operating system activity,

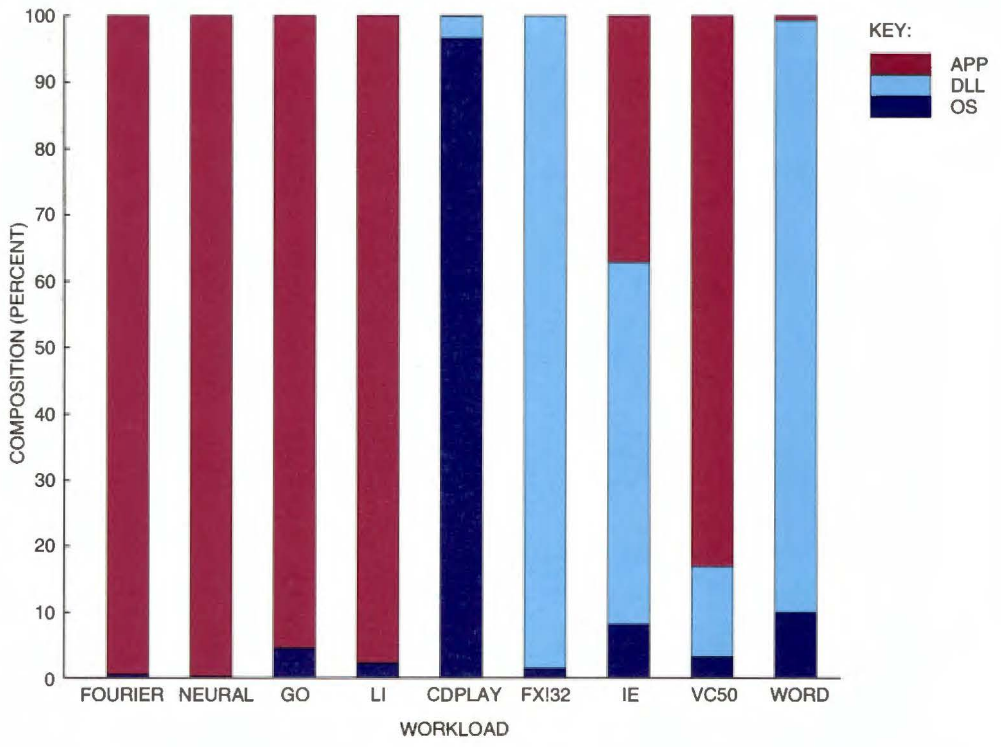


Figure 6
Domain Execution Mix

although small, is present; all I/O must be accessed by means of a system service.

The Microsoft Word spell-checking service is provided by means of a DLL included with the application. Thus for the word workload, this DLL handles both the search through the document and the successive dictionary lookups. Operating system services are required for accessing portions of the file residing on disk (not in memory pages), for displaying the search and compare results to the user, and for performing the user-driven I/O associated with accepting/rejecting word replacement choices (prompted by the spell-checking tool).

Figure 6 shows the consistent pattern of instruction domains that the four benchmarks follow in contrast to the variability in the instruction mix domain of the desktop application workloads. Even though there is slight operating system activity for go and li (attributable to I/O services), the benchmarks spend practically all their execution within their application images; no DLL use is visible. Clearly these benchmarks do not utilize system services to the level observed in the commercial desktop workloads. With the exception of the CD player, the commercial desktop applications examined use DLLs more heavily than they do operating system services. This is especially true in the fx!32 and word workloads, which carry out the tasks captured in the trace by means of DLL routines.

Characterization of Image Usage

To investigate the domains present in the trace at the image level, we identified the top five most heavily used images, based on the number of instructions executed in each image. First, an explanation of some of the more frequently used system executables and DLLs is in order. Table 4 lists the names of the commonly used images and a brief description of each.

We present the image usage of the nine traces. This characterization includes all the images (e.g., executables, DLLs, services, and drivers) listed in Table 5. The data helps demonstrate several points. First, commercial desktop workloads spend a lot more time in DLLs than benchmarks do. Consequently, we can project that the number of procedure calls in desktop applications will be higher than the number of calls in benchmarks. Second, real applications depend not only on system DLLs but also on their local DLLs. We see this behavior explicitly with the Microsoft Word application.

Instruction Mix

Although understanding the domain mix and image usage helps identify differences between benchmarks and desktop applications, we would like to look deeper within each domain to see inherent differences that affect design decisions. Figure 7 shows the application-only instruction mix (i.e., the instruction mix for only the application and application-specific DLLs) for each workload. Each entry in the legend represents a class of instructions found within the application domain. The y-axis denotes the percent composition of the trace; the workloads are displayed on the x-axis.

Note that the instruction mix for the fx!32 workload is zero. This value is a result of the lack of execution within the application image itself. Referring back to Table 5 and the domain instruction mix, note that nearly all the workload execution is within DLLs (some execution is within ntoskrnl.exe). The remaining workloads consist mainly of load, store, conditional branch, and arithmetic and logic unit (ALU) logic operations. No overriding characteristic differentiates benchmarks and desktop applications. Note the significant variability in the instruction mix among the different benchmarks and among the different desktop applications.

Figure 8 shows the instruction mix of the entire trace. The first and most noticeable difference between the application domain and full-trace instruction mix figures is the increase in instruction types present in the trace. Nine instruction classes were present in the application domain instruction mixes, while 17 are present in the full-system traces. Worth noting is the presence of 6 CALL_PAL instruction types (all use the same opcode, but invoke 6 different PAL routines) in the full traces. Since each executed CALL_PAL instruction causes a trap that takes on the order of tens of cycles to complete, we can conclude that this is a

Table 4
Common System Images

Name	Description
ntoskrnl.exe	Windows NT operating system kernel core
hal.dll	Hardware Abstraction Library (HAL), which is responsible for the underlying hardware interface
kernel32.dll	Main kernel library
win32k.sys	Kernel-mode device driver
gdi32.dll	Graphics display interface library
ntdll.dll	Library routines provided to each client process on the Windows NT system
MSVCRT.dll	Microsoft C/C++ run-time library
s3.dll	Graphics adapter library for the test platform
qv.dll	Graphics adapter library for the test platform

Table 5
The Five Most Frequently Used Images in Each Application or Benchmark

Workload	Image Name (Percentage of Total Number of Instructions Executed within the Image)					
fourier	bytecpu.exe (99.5%)	winsrv.dll (0.2%)	win32k.sys (0.1%)	ntoskrnl.exe (0.1%)	user32.dll (0.02%)	Other (0.08%)
neural	bytecpu.exe (99.7%)	winsrv.dll (0.2%)	ntoskrnl.exe (0.03%)	win32k.sys (0.03%)	ntdll.dll (0.02%)	Other (0.02%)
go	go.exe (95.5%)	win32k.sys (2.0%)	ntoskrnl.exe (1.0%)	hal.dll (0.4%)	qv.dll (0.1%)	Other (1.0%)
li	li.exe (97.7%)	win32k.sys (1.0%)	ntoskrnl.exe (0.6%)	user32.dll (0.1%)	qv.dll (0.1%)	Other (0.5%)
cdplay	ntoskrnl.exe (81.8%)	hal.dll (14.7%)	win32k.sys (1.1%)	tcpip.sys (0.4%)	winsrv.dll (0.3%)	Other (1.7%)
fx!32	hal.dll (42.5%)	s3.dll (24.6%)	OPENGL32.DLL (12.2%)	MSVCRT.dll (11.7%)	GLU32.dll (2.7%)	Other (6.3%)
ie	iexplore.exe (37.2%)	win32k.sys (19.3%)	ntoskrnl.exe (17.5%)	Fastfat.sys (6.1%)	ntdll.dll (6.0%)	Other (13.9%)
vc50	c1.exe (83.1%)	ntoskrnl.exe (10.5%)	MSVCRT.dll (2.8%)	Ntfs.sys (1.2%)	win32k.sys (1.1%)	Other (1.3%)
word	MSSP232.DLL (36.4%)	MSGREN32.DLL (34.0%)	ntoskrnl.exe (10.2%)	win32k.sys (7.7%)	hal.dll (4.0%)	Other (7.7%)

significant insight into the system's inherent run-time latency, not visible with application-only workloads.

Next note the striking similarities in instruction mix for the four benchmarks in Figures 7 and 8. Benchmarks do not interact with the operating system in any significant manner. The desktop application workloads, however, show significant differences between the application domain and the complete trace instruction mixes.

The number of store instructions for the cdplay workload *decreases* from about 11 percent to approximately 1 percent. The number of BSR instructions *increases* from 1 percent to about 6 percent. Most interesting for this application is the decrease in the number of ALU operations from almost 30 percent to about 2 percent, while the number of CALL_PAL instructions increases from 0 to 21 percent. Referring to Figure 6, the domain execution mix plots clearly show why the differences for this workload are so large when the system activity is included—more than 95 percent of the workload trace is operating system execution.

Considering the latency incurred by executing CALL_PAL instructions, clearly an optimization that concentrates on improving ALU operations based on the application domain instruction mixes would have a much smaller impact on the true system performance. The measured difference in instruction mix underscores the importance not only of using real workloads for trace-driven simulations but also of including the operating system behavior in order to see the full picture.

The fx!32 complete trace instruction mix is, of course, completely different from the application instruction mix of Figure 7, in which no instructions were executed within the fx!32 application image. Both the ie and the word workloads introduce CALL_PAL instructions when including the operating system. The ie instruction mix shows an increase in jumps, calls, and returns, which most likely reflects the increase in subroutine calls for system services. The word instruction mix experiences a reduction in load instructions from approximately 52 percent to 35 percent. This decrease can be attributed to the increase in ALU operations present when operating system activity is included.

The results presented in Figures 7 and 8 reinforce the points that benchmarks do not represent true desktop workloads and that the desktop workloads display significantly different characteristics when viewed in the presence of system activity.

Average Basic Block Length

Including the operating system activity in our traces yields an overall increase in the percentage of control flow instructions present. Figure 9 shows a consequence of this fact. In this figure, we present the average basic block length for each workload, on a per-domain basis. The ALL bar is the average basic block length across all domains; OS denotes the operating system instructions only; DLL denotes the workload's DLL instructions only; APPDLL denotes the combined application and DLL instructions; and APP denotes the application instructions only.

Inspecting the four benchmarks, we notice little difference between the application-only basic block length and the overall basic block length. Referring to our domain instruction mix figure, recall that the benchmarks spend about 95 percent of their execution

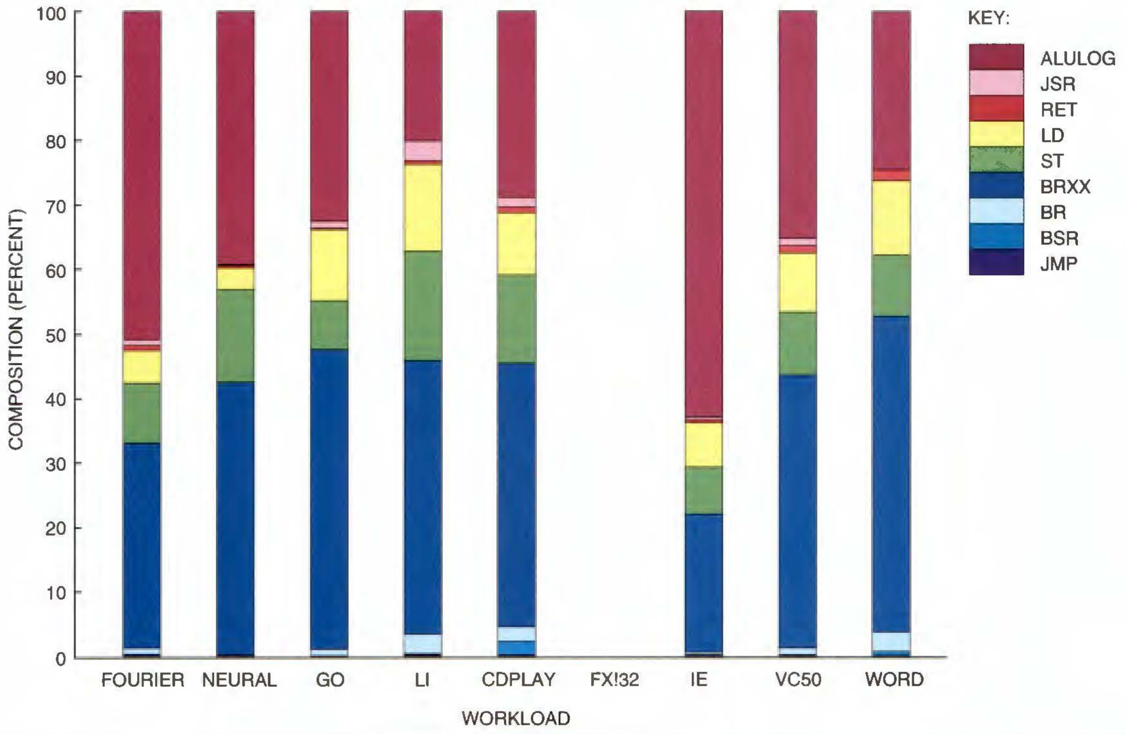


Figure 7
Application-only Instruction Mix

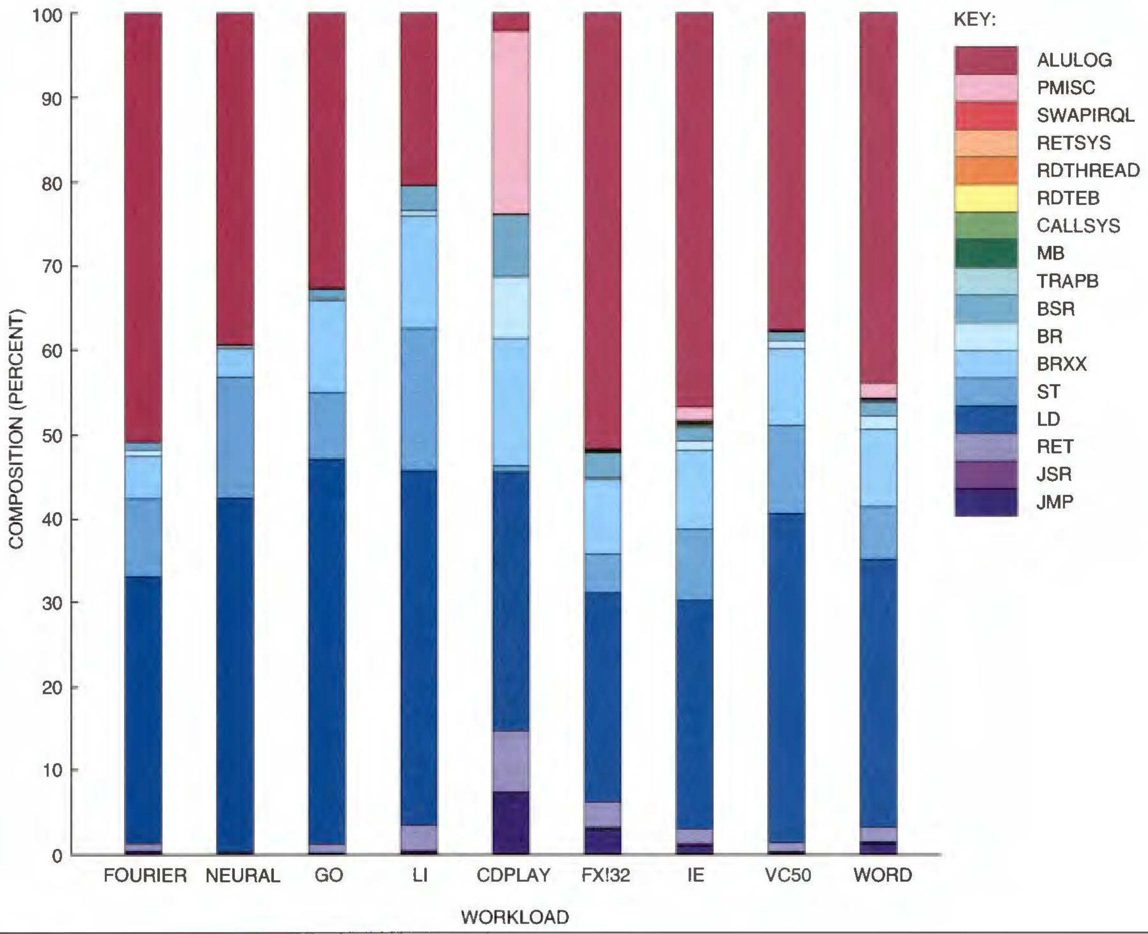


Figure 8
Complete Trace Instruction Mix

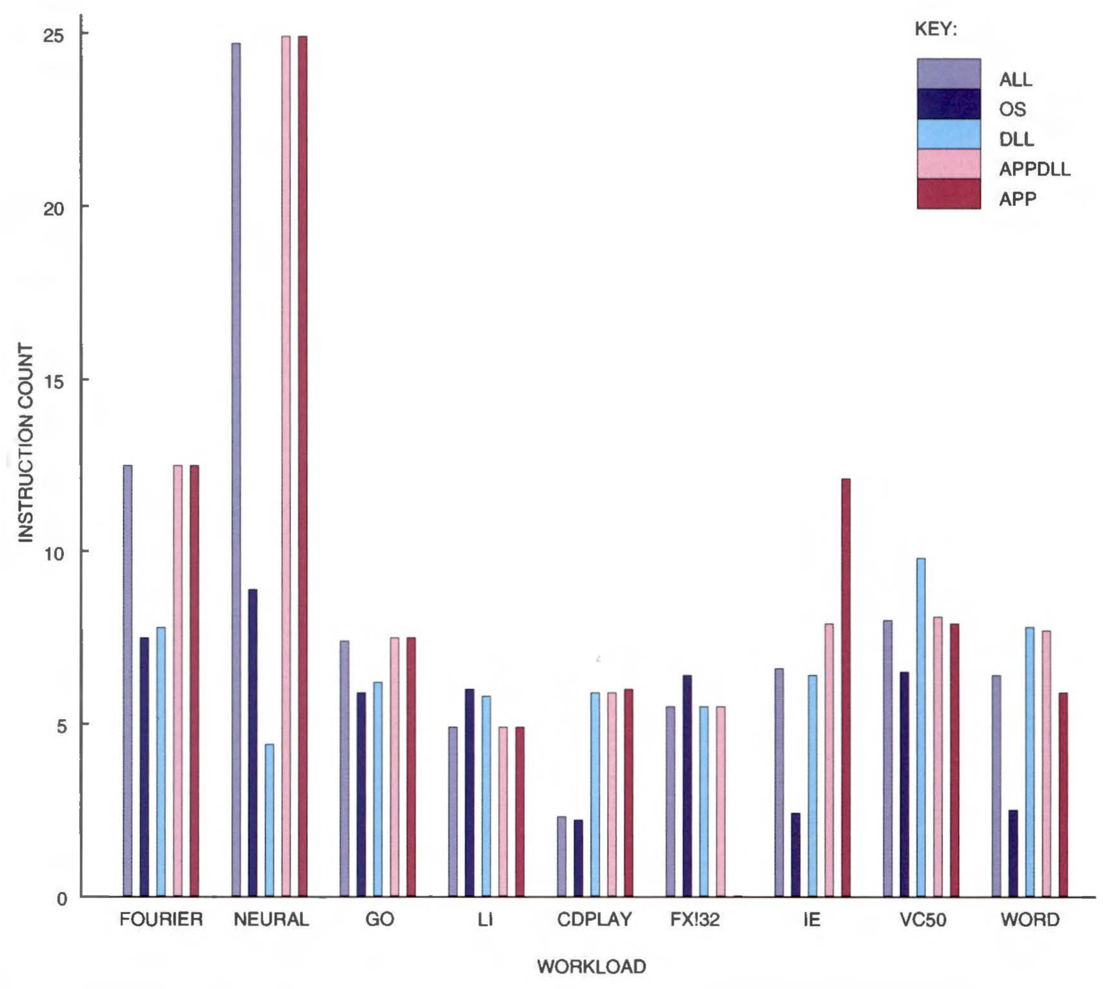


Figure 9
Average Basic Block Length

within their executable images. Therefore, including any operating system activity into a basic block length average has a minimal effect.

However, considering the large amount of operating system execution present in the cdplay trace, the overall basic block length is significantly less than the application-only length. The overall and operating system length values are almost the same. Not only does including the system activity in the trace influence the overall basic block length but the *amount* of system activity determines to what degree the length is affected.

In a similar fashion, the overall basic block length of the fx!32 trace tracks that of its DLLs. The length is directly proportional to the amount of time the workload spends in its DLL domain. The execution of the ie workload is more evenly distributed among the three domains, which affects the overall basic block length, producing a more evenly weighted average of all its domain basic block lengths (no one domain dominates).

The vc50 workload spends a significant amount of time within its own executable image, which leads to an overall average basic block length similar to the application-only value. The word workload is similar, but the DLL behavior dominates. The cdplay and ie workloads experience a 50 percent decrease in average basic block length. This decrease can be attributed to an increase in the number of branches in the presence of operating system activity. With this increase in control flow instructions, we expect increased pressure to be placed upon the branch prediction hardware.

As observed in other characteristic categories, the four benchmarks do not exhibit noticeable deviations from application-only behavior when the operating system activity is introduced. Again this explains why simulation results using benchmark traces usually track the actual performance when the benchmarks are run on the real system. In contrast, four of the five desktop applications exhibit significantly different behavior in the presence of the operating system.

Summary

In this paper we described the PatchWrx toolset. We compared it to existing tools and demonstrated the need for operating system–rich traces by showing the amount of the total execution spent in the kernel and the DLLs. In addition, we showed that existing desktop benchmarks do not exercise the kernel and the DLL sufficiently to provide meaningful indicators of desktop performance.

These results have reinforced our argument that researchers need to use traces with both application and operating system information, especially as new applications spend more time executing within the operating system. The goal is for computer architects to use operating system–rich traces of applications that dominate the desktop market.

We have recently finished modifications to the PAL to enable PatchWrx to run on the Alpha 21164 platform. We plan to study a wider range of desktop applications, including database and server applications. Future plans also include migrating the toolset to the Windows 2000 operating system.

Acknowledgments

We would like to acknowledge the help and advice of the following people: Richard Sites of Adobe Systems; Sharon Smith, Geoff Lowney, Joel Emer, Steve Thierauf, Tom Wenners, Paul Delvy, and Dan Lambalot, all from Compaq Computer Corporation; and Robert Davidson from Microsoft Research. Jason Casmira and David Kaeli have been supported by a National Science Foundation CAREER grant.

References and Notes

1. *SPEC Newsletter* (September 1995).
2. Information about the BYTEmark benchmark suite is available from *BYTE Magazine* at http://www.byte.com/bmark/bmark.htm.
3. S. Perl and R. Sites, "Studies of Windows NT Performance Using Dynamic Execution Traces," *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation* (October 1996): 169–183.
4. D. Kaeli, "Issues in Trace-Driven Simulation," *Lecture Notes in Computer Science, No. 729, Performance Evaluation of Computer and Communication Systems,* L. Donatiello and R. Nelson, eds. (Springer-Verlag, 1993): 224–244.
5. R. Uhlig and T. Mudge, "Trace-Driven Memory Simulation: A Survey," *ACM Computing Surveys,* vol. 29, no. 2 (June 1997): 128–170.
6. J. Emer and D. Clark, "A Characterization of Processor Performance in the VAX 11-780," *Proceedings of the Eleventh Symposium on Computer Architecture* (June 1994): 126–135.
7. K. Flanagan, J. Archibald, B. Nelson, and K. Grimsrud, "BACH: BYU Address Collection Hardware; The Collection of Complete Traces," *Proceedings of the Sixth International Conference on Modeling Techniques and Tools for Computer Evaluation* (1992): 51–65.
8. D. Kaeli, O. LaMaire, W. White, P. Hennet, and W. Starke, "Real-Time Trace Generation," *International Journal on Computer Simulation,* vol. 6, no. 1 (1996): 53–68.
9. D. Kaeli, L. Fong, D. Renfrew, K. Imming, and R. Booth, "Performance Analysis on a CC-NUMA Prototype," *IBM Journal of Research and Development, Special Issue on Performance Tools,* vol. 41, no. 3 (May 1997): 205–214.
10. D. Nagle, R. Uhlig, and T. Mudge, "Monster: A Tool for Analyzing the Interaction Between Operating Systems and Computer Architectures," Technical Report, CSE-TR-147-92, University of Michigan, 1992.
11. B. Chen and B. Bershad, "The Impact of Operating System Structure on Memory System Performance," *Operating Systems Review,* vol. 27, no. 5 (December 1993): 120–133.
12. J. Larus, "Abstract Execution: A Technique for Efficiently Tracing Programs," Technical Report, CS-TR-90-912, University of Wisconsin-Madison, 1990.
13. A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation, Orlando,* Fla. (June 1994): 196–205.
14. M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta, "Complete Computer System Simulation: The SimOS Approach," *IEEE Journal of Parallel and Distributed Technology,* 1998, forthcoming.
15. M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Transactions on Modeling and Simulation,* vol. 7, no. 1 (January 1997): 78–103.
16. A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming* (Kluwer Academic Publisher, 1989).
17. J. Larus and E. Schnarr, "EEL: Rewriting Executable Files to Measure Program Behavior," *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation,* La Jolla, Calif. (June 1995): 291–300.

18. D. Lee, P. Crowley, J.-L. Baer, T. Anderson, and B. Bershad, "Execution Characteristics of Desktop Applications on Windows NT," *Proceedings of the Twenty-fifth International Symposium on Computer Architecture,* Barcelona, Spain (June 1998).

19. E. Betts, D. Hunter, and S. Smith, "Moving ATOM to Windows NT for Alpha," *Digital Technical Journal,* vol. 10, no. 2, accepted for publication.

20. M. Smith, "Tracing with Pixie," Technical Report, CSL-TR-91-497, Stanford University, November 1991.

21. R. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," *Proceedings of ACM Sigmetrics* (May 1994): 128–137.

22. *Alpha AXP Architecture Handbook,* Order No. EC-QD2KA-TE (Maynard, Mass.: Digital Equipment Corporation, October 1994).

23. H. Custer, *Inside Windows NT* (Redmond, Wash.: Microsoft Press, 1993).

24. Microsoft Software Developer's Toolkit. This toolkit is available at http://msdn.microsoft.com/developer/sdk/platform.htm.

25. J. Casmira, "Operating System Rich Workload Characterization," Master's thesis, ECE-CEG-98-018, Northeastern University, May 1998.

26. R. Hookway and M. Herdeg, "DIGITAL FX!32: Combining Emulation and Binary Translation," *Digital Technical Journal,* vol. 9, no. 1 (1997): 3–12.

Biographies

Jason P. Casmira
Jason Casmira received B.S. and M.S. degrees in electrical engineering from Northeastern University in 1996 and 1998, respectively, and is pursuing a Ph.D. degree in computer science at the University of Colorado, Boulder. For the past two years, Jason was a member of the Northeastern University Computer Architecture Research Laboratory (NUCAR), where he focused on developing the current version of the PatchWrx tracing toolset. He also investigated issues related to studying operating system–rich traces. While at NUCAR, Jason was supported by a grant from the National Science Foundation. He has published seven papers and is a member of the IEEE and the Eta Kappa Nu honor society.

David P. Hunter
David Hunter is the engineering manager of Compaq Computer Corporation's Advanced and Emerging Technologies Group. Prior to that he was the manager of DIGITAL's Software Partner Engineering Advanced Development Group, where he was involved in performance investigations of databases and their interactions with the UNIX and Windows NT operating systems. He has held positions in the Alpha Migration Organization, the ISV Porting Group, and the Government Group's Technical Program Management Office. David joined DIGITAL's Laboratory Data Products Group in 1983, where he developed the VAXlab User Management System. He was the project leader of the advanced development project, ITS, an executive information system, for which he designed hardware and software components. David has two patent applications pending in the area of software engineering. He holds a degree in electrical and computer engineering from Northeastern University in Boston, Massachusetts, and a diploma in National Security and Strategic Studies from the United States Naval War College in Newport, Rhode Island.

David R. Kaeli
David Kaeli received Ph.D. (1992) and B.S. (1981) degrees in electrical engineering from Rutgers University and an M.S. degree in computer engineering from Syracuse University in 1985. He joined the electrical and computer engineering faculty at Northeastern University in 1993 after spending 12 years at IBM, the last 7 of which were at the IBM T. J. Watson Research Center in Yorktown Heights, New York. David is the director of the Northeastern University Computer Architecture Research Laboratory (NUCAR), where he investigates the performance and design of high-performance computer systems and software. His current research topics include I/O workload characterization, branch prediction studies, memory hierarchy design, object-oriented code execution performance, 3-D microelectronics, and back-end compiler design. He frequently gives tutorials on the subject of trace-driven characterization and simulation. In 1995, David received the prestigious National Science Foundation CAREER Award. His research has been supported by the Office of Naval Research, Kopin Corporation, Digital Equipment Corporation, EMC, Data General, Microsoft Research, I-Tech Corporation, IEEE DAC, and IBM Research. David is a member of the ACM, IEEE, and the Eta Kappa Nu and Sigma Xi honor societies.

Avrum E. Itzkowitz
Lois D. Foltan

Automatic Template Instantiation In DIGITAL C++

Automatic template instantiation in DIGITAL C++ version 6.0 employs a compile-time scheme that generates instantiation object files into a repository. This paper provides an overview of the C++ template facility and the template instantiation process, including manual and automatic instantiation techniques. It reviews the features of template instantiation in DIGITAL C++ and focuses on the development and implementation of automatic template instantiation in DIGITAL C++ version 6.0.

The template facility within the C++ language allows the user to provide a template for a class or function and then apply specific arguments to the template to specify a type or function. The process of applying arguments to a template, referred to as template instantiation, causes specific code to be generated to implement the functions and static data members of the instantiated template as needed by the program. Automatic template instantiation relieves the user of determining which template entities need to be instantiated and where they should be instantiated.

In this paper, we review the C++ template facility and describe approaches to implementing automatic template instantiation. We follow that with a discussion of the facilities, rationale, and experience of the DIGITAL C++ automatic template instantiation support. We then describe the design of the DIGITAL C++ version 6.0 automatic template instantiation facility and indicate areas to be explored for further improvement.

C++ Template Facility

The C++ language provides a template facility that allows the user to create a family of classes or functions that are parameterized by type.[1,2] For example, a user may provide a Stack template, which defines a stack class for its argument type. Consider the following template declaration:

```
template <class T> class Stack {
   T *top_of_stack;
public:
   void push( T arg );
   void pop( T& arg );
};
```

The act of applying the arguments to the template is referred to as template instantiation. An instantiation of a template creates a new type or function that is defined for the specified types. Stack<int> creates a class that provides a stack of the type int. Stack<user_class> creates a class that provides a stack of user_class. The types int and user_class are the arguments for the template Stack.

In general, a template needs to be instantiated when it is referenced. When a class template is instantiated, only those member functions and static data members that are referenced are also instantiated. In the Stack example, the member function Push of the class Stack<int> needs to be instantiated only if it is used. Template functions and static data members have global scope; therefore, only one instantiation of each should be in a user's application. Since source files are compiled separately and combined later at link time to produce an executable, the compiler alone is not able to ensure that one and only one instance of a specific template is efficiently generated for any given executable. That is, the compiler by itself is not able to know whether the function or variable definition for a specific template is satisfied by code generated in another object module.

The C++ Standard provides facilities for the user to specify where a template entity should be instantiated.[1] When the user explicitly specifies template instantiation, the user then becomes responsible for ensuring that there is only one instantiation of the template function or static data member per application. This responsibility can necessitate a considerable amount of work. However, the compiler and linker working together can provide effective template instantiation without specific user direction.

In the following section, we present the various approaches that can be used for template instantiation.

Template Instantiation Techniques

Template instantiation techniques can be broadly categorized as either manual or automatic. With manual instantiation, the compilation system responds to user directives to instantiate template entities. These directives can be in the source program, or they may be command-line options. With automatic instantiation, the compilation system, including the linker, decides which instantiations are required and attempts to provide them for the user's application.

Manual Instantiation

Manual template instantiation is the act of manually specifying that a template should be instantiated in the file that is being compiled. This instantiation is given global external linkage, so that references to the instantiation that are made in other files resolve to this template instantiation. Manual template instantiation includes explicit instantiation requests and pragmas as well as command-line options.

Explicit Instantiation Requests and Pragmas The compilation system instantiates those template entities that the user specifies for instantiation. The specification can be made using the C++ explicit template instantiation syntax or may be made using implementation-defined directives or pragmas. Since instantiations are given global external linkage, the user must ensure that the specified template instantiations appear only once throughout all the modules that compose the program. When only this mode of instantiation is used, the user also must ensure that all required template instantiations are specified to avoid unresolved symbols at link time.

Command-line Instantiation Command-line options can be used to specify template instantiation. They are similar in operation to the explicit instantiation requests, except they indicate groups of templates that should be instantiated, rather than naming specific templates to be instantiated. The command-line options include

- Instantiate All Templates. A command-line option can direct the compiler to instantiate all template entities whose definitions are known during compilation and whose argument types are specified. This has the advantage of specifying many template instantiations at once. The user must still ensure that no template instantiation happens more than once in the program and that all required instantiations are satisfied. Due to these requirements, the user cannot usually specify this option on more than one source-file compilation in the program. This option can also cause the instantiation of templates that are not used by the program.
- Instantiate Used Templates. A command-line option can be used to direct the compiler to instantiate only those template entities that are used by the source code and whose definitions are known at compilation. As in the previous technique, the user must ensure that no template instantiation happens more than once in the program and that all required instantiations are satisfied. Due to these requirements, the user cannot usually specify this option on more than one source-file compilation in the program.
- Instantiate Used Templates Locally. This command-line option works like the instantiate used templates option, except that it defines each template instantiation locally in the current compilation. This option has the advantage of providing complete template instantiation coverage for the program, as long as the definitions of the used templates are available in each module. Since all template instantiations are given local scope, there is no potential problem with multiply defined instantiations when the program is linked. The major problem with this technique is that the user's application can be unnecessarily large, since the same template instantiations could appear within multiple object files used to link the application. This technique will fail if the instantiations must have global scope such as a class's static data members.

Figure 1 shows an example of a template function, template_func, that contains a locally defined static variable. As shown in the figure, the object files of both A and B contain local copies of template_func instantiated with int. Each instance of template_func<int> defines its own version of static variable *x*. In this case, directing the compiler to instantiate used templates locally yields a different result than instantiating all or used templates globally.

If we give the static data members global scope and ensure that they are properly defined and initialized by executable code rather than by static initialization, we can solve the static data members problem. The application, however, remains unnecessarily large, because multiple copies of the instantiated templates can be present in the executable.

Automatic Instantiation

Automatic template instantiation relieves the user of the burden of determining which templates must be instantiated and where in the application those instantiations should take place. Automatic template instantiation can be divided into two categories: compile-time instantiation, whereby the decision about what should be instantiated is made at compile time, and link-time instantiation, whereby decisions about template instantiation are made when the user's application is linked. In both cases, specific link-time support is needed to select the required instantiations for the executable.

Compile-time Instantiation Two major techniques can be used to perform automatic template instantiation at compile time. The choice between the two depends upon the facilities available in the linker. Microsoft Visual C++ instantiates templates at compile time using a strategy similar to the instantiate used templates command-line option described previously.[3] Each instantiation is placed in the communal data section (COMDAT) of the current compilation's object file. Each object file contains a copy of every template instantiation needed by that compilation unit. COMDATs are sections that have an attribute that tells the linker to accept, without issuing a warning, multiple definitions of a symbol defined in the section.[4] If more than one object file defines that symbol, only the section from one object file is linked into the image and the rest are discarded, along with all symbols in the symbol table defined in the discarded section contribution. At link time, the linker resolves an instantiation reference by choosing one of the instantiations defined in an individual object file's COMDAT. The resulting user's application executable has a single copy of each requested instantiation.

When such linker support is not available, another mechanism must be used to control compile-time instantiation. One such approach is to use a repository to contain the generated instantiations. The compiler creates the instantiations in the repository instead of the current compilation's object file. At link time, the linker includes any requested instantiations from the repository. As a performance improvement, the compiler can also decide whether an instantiation needs to be generated from the state of the repository. If the requested instantiation is in the repository and can be determined to be up to date, the compiler does not need to regenerate the instantiation.

Link-time Instantiation The decision to instantiate can be left until link time. The linker can find the instantiations that are needed and direct the compiler to generate those instantiations. McCluskey describes one link-time instantiation scheme.[5,6] The compiler logs every class, union, struct, or enum in a name-mapping file in a repository. Every declared template is also logged in the name-

```
//template.hxx
#include <iostream.h>
template <class T> void template_func (T p)
{
   static T x = 0;
   cout << x + p;
   x++;
}

//A.cxx                        //B.cxx
#include "template.hxx"        #include "template.hxx"
extern void b_func();          void b_func(void)
int main()                     {
{                                 //...
   template_func(10);             template_func(20);
   b_func();                      //...
   return 0;                   }
}
```

Figure 1
Template Function Containing a Locally Defined Static Variable

mapping file. At link time, a prelinker determines which template instantiations are required. The prelinker builds temporary instantiation source files in the repository to satisfy the referenced instantiations, compiles them, and adds the resulting object files to the linker input. Consider the example in Figure 2.

During the compilation of main.cxx, a name-mapping file is built in the repository and the location of the user-defined class C and the function template, perform_some_function, are recorded. From the information stored in the name-mapping file, an instantiation source file is then created in the repository. Figure 3 shows the contents of the instantiation source file created to satisfy perform_some_function<C>.

The prelinker then compiles the instantiation source file by invoking the compiler in a special directed mode, which directs the compiler to generate code only for specific template instantiations that are listed on the command line. The compiler then generates the definition of perform_some_function<C> in the resulting object file. The resulting object now satisfies the instantiation request and is included as part of the application's final link. To build the instantiation source files easily, the implementation of this scheme generally requires that template declarations, template definitions, and any argument types used to instantiate a class or function template must appear in separate, related header files.

The Edison Design Group has developed another approach to link-time instantiation.[7] In this approach, the compiler records where template instantiations are used and where they can be instantiated. At link time, a prelinker assigns template instantiations by recording the assignments in a specially generated file that corresponds to the particular source file that can successfully instantiate the user's request. Compiling and prelinking the program used in Figure 2 generates an instantiation assignment file for main.cxx. This file contains information concerning the command-line options specified, the user's current working directory, and a list of instantiations that should be instantiated. Main.cxx now owns the responsibility of instantiating perform_some_function<C>. The prelinker recompiles the source files, such as main.cxx, that have changes in their template instantiation assignments. The process is repeated until there are no changes made to the instantiation assignments. Then the final link can be completed.

This approach has the advantage of requiring no special file structure to support automatic template instantiation. It is generally faster and simpler than McCluskey's approach, because fewer files are compiled in the generation of the needed instantiations and the instantiations are generated in the context of the user's source code. In addition, the assignment of instantiations to source files can be preserved between recompilations of the source code, so that unless the structure of the application changes, the needed instantiations will be available without additional recompilation.

```
/* perform_some_function(C&) */
#include "template.hxx"
#include "template.cxx"
#include "C_class.h"
```

Figure 3
Example of an Instantiation Source File

```
//C_class.hxx
class C {
public:
   //...
        };

//template.hxx
template <class T> void perform_some_function(T &param);

//template.cxx
template <class T> void perform_some_function(T &param) { }

//main.cxx
#include "C_class.hxx"
#include "template.hxx"

int main()
{
    C c;
    perform_some_function(c);
    return 0;
}
```

Figure 2
Example of a Link-time Instantiation Scheme (McCluskey)

Comparison of Manual and Automatic Instantiation Techniques

The manual instantiation techniques require planning on the part of the user to ensure that needed instantiations are present, that no extraneous instantiations are generated, and that each needed instantiation appears exactly once within the application. With manual instantiation, the user has the advantage of gaining explicit control over all template instantiations. Although the strategy of instantiating used templates locally requires less planning, it does so at the cost of object file size and the restricted use of templates when static data members are present or when static data is defined locally within a function template instantiation.

Automatic template instantiation provides template instantiation with no explicit action on the part of the user. Compile-time instantiation requires either specific linker support to select a single template instantiation from potentially many candidates, or support by the compiler to generate instantiations in separate object files while compiling the user's source code. Relying on linker support allows the compiler to efficiently generate instantiations at the cost of larger object files; however, the user loses control over which instantiation is used in the executable file. Although the use of separate instantiation object files usually takes more time at compilation than the linker-support method, it results in more compact object files and can provide the user with more control over which instantiation is used in the executable file.

Link-time instantiation provides template instantiation that is tailored to the needs of the executable file. The primary cost is link-time performance, since generation of instantiations occurs at link time. Another disadvantage of link-time instantiation can be observed when building object-code libraries. Either the library must contain all the instantiations that it requires, or the user who wants to link with the library must have access to all the machinery to create instantiations. Creating a library's instantiations involves extra steps during library construction. All the object files to be included in the library must be prelinked, so that the needed instantiations are generated. If instantiations are included in the individual object files in the library, as in the Edison Design Group approach, unintended modules may be linked from the library to provide the needed instantiations. Consider the following scenario, in which object files A and B are included in the library. Both files require the instantiation of perform_some_function<int>. When these files are prelinked, the instantiation of perform_some_function<int> is assigned to one of the files, say A. If an application that is being linked against the library requires that the object file B be linked into the executable, then the object file A is also linked. Here the instantiation needed by B was instantiated in A even though the executable never referenced anything explicitly defined in file A. This can yield an unnecessarily large executable.

In the next section, we review the template instantiation support in earlier versions of DIGITAL C++ and then discuss the rationale and design of the automatic template instantiation facility in version 6.0 of DIGITAL C++.

DIGITAL C++ Template Instantiation Experience

As the use of C++ templates has grown, DIGITAL C++ has been enhanced to support the need for improved instantiation techniques. The initial release of DIGITAL C++ occurred before the C++ standardization process had matured, so that the language supported was based on *The Annotated C++ Reference Manual,* referred to as the ARM.[8] The ARM defined template functionality, but it did not provide guidance for either manual or automatic template instantiation. Thus it was necessary to provide a DIGITAL C++-specific mechanism for template instantiation.

DIGITAL C++ Manual Template Instantiation

The #pragma define_template directive and the instantiate all command-line option, -define_templates, have been supported since the initial release of DIGITAL C++.

In Figure 4, the define_template pragma directs the compiler to instantiate class template, C, with type int. When the compiler detects the use of the pragma, it creates an internal C<int> type node and traverses the list of static data members and member functions defined within the class. If the definitions of these members are present at the point the pragma is specified, the compiler materializes each with type int.

As the C++ language developed and template usage increased, users found manual template instantiation to be very labor intensive and requested an automated method.

DIGITAL C++ Version 5.3 Automatic Template Instantiation

Automatic template instantiation capability became a serious issue during the planning stages of DIGITAL C++ version 5.3. The use of templates was increasing rapidly, and many new third-party libraries, such as Rogue Wave Software's Tools.h++, contained a significant use of templates. Due to this growing need, the requirements were straightforward. The support had to be easy to use, have a short design phase, be quickly implementable on both the DIGITAL UNIX and the OpenVMS platforms, and provide reasonable performance. Because McCluskey's approach had been used in several implementations, it presented itself as our best option.

```
template <class T> class C {
public:
   void mem_func1(T p);
   void mem_func2(T p);
   ...
};

template <class T> void C<T>::mem_func1(T p) { //...}
template <class T> void C<T>::mem_func2(T p) { //...}

#pragma define_template C<int>
```

Figure 4
The define_template Pragma

DIGITAL made two major changes to McCluskey's approach to take advantage of the DIGITAL C++ compiler design. First, we allowed instantiation source files to be created at compile time instead of link time. This eliminated the need for McCluskey's name-mapping file and simplified the prelinking process considerably. Since the needed source files existed in the repository, there was no need to deconstruct the required template instantiations to determine their arguments and types.

The second change addressed the transitive closure problem. Figure 5 shows an example of the class template Buffer being instantiated with the user-defined type C. After compilation of app.cxx with the McCluskey approach, the name-mapping file contained definition locations of class B and class C. However, it did not contain any indication that class C had a data member that relied on the definition of class B. From the information in the name-mapping file, the prelinker then created an instantiation source file that included only C_class.hxx, Buffer.hxx, and Buffer.cxx. When this instantiation source file was compiled, an error resulted complaining that B is an undefined type whose size is unknown.

We solved this problem in DIGITAL C++ version 5.3 by including all the top-level header files included by the current compilation unit in any instantiation source files created. This ensured that B_class.hxx would be included in the generated instantiation file.

```
//B_class.hxx
class B { //... };
```

```
//C_class.hxx
class C {
   B data_mem;
public:
   //...
};
```

```
//Buffer.hxx
template <class T> class Buffer {

   T *buffer;
   int num_of_items;
public:
   void add_item(T *);
   //...
};
```

```
//Buffer.cxx
template <class T>
         void Buffer<T>::add_item(T *p) { }
```

```
//app.cxx
#include "B_class.hxx"
#include "C_class.hxx"
#include "Buffer.hxx"

void f(void)
{
   C c;
   Buffer<C> c_buffer;
   c_buffer.add_item(&c);
}
```

Figure 5
Instantiation of the Class Template Buffer

Despite the fact that this type of automatic link-time instantiation scheme was being widely used in the industry, the results of using a modified McCluskey approach were mixed. Stroustrup has described the general problems with McCluskey's approach.[9] We found that our implementation suffered particularly from poor link-time performance and so did not satisfy our users' needs.

DIGITAL C++ Version 6.0 Automatic Template Instantiation

DIGITAL C++ version 6.0 is a complete reimplementation of DIGITAL C++, with emphasis on ANSI C++ conformance. It is implemented using a completely new code base, which includes the industry-standard C++ front end from the Edison Design Group and a standard class library from Rogue Wave.

From our experience with template instantiation in DIGITAL C++ versions 5.3 through 5.6, we concluded that the most important issue that should be addressed in the design and implementation of the automatic template instantiation facility was the compile- and link-time performance. The primary goal was to have the performance of automatic template instantiation substantially exceed the performance of version 5.6. Another important goal was to remove the restriction of template declaration and definition placement in header files. In addition, the automatic template instantiation facility in version 6.0 had to be culturally compatible with the previous implementation. The user had to be able to move sources and objects to different directories, easily build archived and shared libraries, share instantiations between various applications, and have error diagnostics reported at the earliest possible moment in the instantiation process.

Design and Implementation We decided to use a compile-time instantiation model as the basis for our implementation. Since we were using the Edison Design Group's front end, we seriously considered using their link-time model. However, the compile-time model seemed advantageous for several reasons. First, there are significant complications (as described in the section Comparison of Manual and Automatic Instantiation Techniques) when trying to build libraries with a compiler that uses the Edison Design Group link-time model. In addition, the link-time model requires recompilations that limit performance in many typical cases of template use. We recognized that the link-time model could provide better performance in some cases, but these would be in the minority. Finally, the implementation of the link-time model would require substantially more implementation effort on the OpenVMS platform. The version of the Edison Design Group front end being used to build DIGITAL C++ version 6.0 required tools to scan a user's object files for information concerning which modules could instantiate requested templates. Similar functionality would need to be implemented for the OpenVMS platform.

We preserved the concept of the template repository as a directory that contains the individual template instantiation object files. The repository stores one object file for each template function, member function, static data member, and virtual table that is generated by automatic template instantiation. The file name of the instantiation object file is derived from the name of the instantiation's external name. At compile time, the front end generates intermediate code for all templates that are needed in the compilation unit and can be instantiated. A tree walk is performed over the intermediate code to find all entities that are needed by each generated template instantiation. The code generator is called to generate code for the user-specified object file and is then called repeatedly for each template instantiation to generate the instantiation object files in the repository.

The compiler generally considers an instantiation to be needed when it is referenced from a context that is itself needed, such as in a function with global visibility or by the initialization of a variable that is needed. Virtual member functions are needed when a constructor for the class is needed. Thus, all virtual function definitions should be visible in a compilation unit that requires a constructor for the class. Each instantiation that is generated with automatic instantiation is marked as potentially being in its own object file in the repository.

The intermediate representation of each generated instantiation is walked to determine what other entities it references. At this point, the instantiation is a candidate to be generated in its own object file, but it can sometimes be generated as part of the user-specified object file. If the instantiation references an entity that is local to the compilation unit, such as a static function, and that local entity is nonconstant and statically initialized, the instantiation is merged into the user-specified object file rather than generated in its own object file. As an alternative, we could have chosen to change the local entity into a global entity with a unique name and generate the instantiation in its own object file. We chose not to do this in order to make it easier to share a repository between applications. With this alternative, the instantiation in the repository requires the object file containing the local entity's definition, which may be in another application. Note that any application that contains more than one definition of the same instantiation that references a nonconstant local entity is a nonstandard-conforming application. This is a violation of the one definition rule.[10] Consider the following code fragment:

```
static int j;
template <class T> int func (T arg) { return j; }
int var = func( 2.5 );
```

The reference to the static variable *j* in the template function, func, prevents the template from being generated into its own object file in the repository.

When the individual instantiations are walked, we mark each global entity that is defined in the compilation unit so that the definition is replaced by an external reference when the instantiation object file is generated. Consider the following code fragment:

```
void print_count(const char * s, int ivar)
{
   cout<< s <<":" << ivar;
}

template <class T> void func (T arg)
{
   static int count = 0;
   print_count("count", count++);
}
```

The function, print_count, is defined in the source file and generated as a defined function in the user-specified object file. The template function, func, references the function, print_count. When the code for func is generated in its own object file, the reference to print_count must be changed from a reference to a defined function to a reference to an external function.

By default, each needed instantiation is generated by every compilation that requires the instantiation. This is the safe default because it ensures that instantiations in the repository are up to date. However, there will probably be some compilation overhead from regenerating instantiations that may already be up to date. We believed that the overhead of regenerating instantiations would typically be relatively small. For applications with a high overhead of instantiation, such as a large number of source files using the same large number of template instantiations, we provided a compilation option to control the generation of template instantiations to improve compile-time performance.

The generation of instantiation object files only when they are actually required is a difficult problem. Fine-grain dependency information would have to be kept for each instantiation object file. Such dependency information would need to reflect those files that are required to successfully generate the instantiation and record which command-line options the user specified to the compiler. We suspected that the overhead involved with gathering and checking the information might be an appreciable percentage of the time it would take to do the instantiation, and thus it would not give us the performance improvement that we wanted.

Instead, we decided to provide an option that allows the user to decide when instantiations are generated. We refer to this as the template time-stamp option, -ttimestamp. When using the time-stamp option, the compiler looks in the repository for a file named TIMESTAMP. If the file is not found, it is created. The modification time of this file is referred to as the time stamp. When generating an instantiation, the compiler looks in the repository to see if the instantiation object file exists. If it does not exist, it is generated. If the file already exists, its modification time is compared to the time stamp. If the modification time is later than the time stamp, the instantiation is assumed to be up to date and is not regenerated. Otherwise, the instantiation is generated. The user can control the generation of instantiation object files by changing the modification time of the TIMESTAMP file.

The time-stamp option would typically be used in a makefile or a shell script that compiles and builds an entire application. Before invoking make or the shell script, the user would make certain that no TIMESTAMP file resided in the repository. This would ensure that each needed instantiation would be generated exactly once during all the compilations done by the build procedure.

Much of the C++ linker support in version 5.6 was reused with only minor modifications for version 6.0. The compiler is presented with a single repository into which the instantiation object files are written. Multiple repositories can be specified at link time, and each can be searched for instantiations that are needed by the executable file. The linker is used in a trial link mode to generate a list of all the unresolved external references. This list is then used to search the repositories to find the needed instantiation files, and the process is repeated until no more instantiations are needed or can be satisfied from the repository. The link then proceeds as any normal link, adding the list of instantiation object files to the list of object files and libraries as specified by the user.

If a vendor is creating a library rather than an executable file, the instantiations needed by the modules in the library can be provided in either of two ways: (1) The library vendor can put the needed instantiations in the library by adding the files in the repository to the library file. (2) The library vendor can provide the repository with the library and require that library users link with the repository as well. Note that instantiations placed in the library are fixed when the library is created. Since the library is included in the trial link of an application, any instantiation in the library takes precedence over the same named instantiation in a repository.

Results In a number of tests, DIGITAL C++ version 6.0 showed improved performance over version 5.6. We tested a variety of user code samples that use templates to varying degrees and found that build times for version 6.0 decreased substantially compared to the version 5.6 compiler. Examples of two typical C++ applications used in our tests are the publicly available EON ray-tracing benchmark and a subset of tests from our Standard Template Library (STL) test suite. For the EON benchmark, the build time for version 6.0 was reduced to 28 percent of the build time for version 5.6. For the STL tests, the build time for version 6.0 was reduced to 19 percent of the build time for version 5.6. The number of files in the repository also decreased significantly because version 6.0 generates only instantiation object files instead of the instantiation source, command, dependency, and object files of version 5.6. For EON, the version 6.0 repository contained 88 files compared to 260 files in version 5.6.

Using the time-stamp option, build time for the EON benchmark was reduced by only 5 percent compared to the default instantiation strategy. The real benefit of the time-stamp option comes with applications that use the same template instantiations in many compilation units. For example, in one user's test case, build times dropped from roughly 18 hours with the default instantiation to 3 hours when using the time-stamp option.

In the next section, we conclude our paper with a discussion of further work that can improve the performance and usability of automatic template instantiation.

Future Research

We continue to investigate approaches and techniques to improve the usability and performance of the automatic template instantiation facility. Optimal usability and performance would seem to require a development environment completely integrated for C++. This environment would keep track of all entity definitions and usage and would be able to limit all instantiation generation to the minimum needed. This approach would require a great deal of development work and might be difficult to integrate with existing customer development methodologies. Therefore, we focus on more modest techniques that approximate the optimal case.

We are exploring ways to improve both performance and usability in the management of dependency information. We continue to look at approaches for using dependencies that can be reliable, automatic, and fast. We also continue to investigate ways to gather and check fine-grained dependency information for the instantiation object files, though performance is a concern. One approximation to the fine-grain dependency information that we are investigating is a larger grain dependency scheme. This technique creates a time stamp from the latest creation time of any source file included during compilation of a given module. Any instantiation object file in the repository whose modification time is later than this time stamp would not be regenerated. This approach is more automatic and can potentially yield better performance than our current time-stamp option, but it would not be sensitive to changes on the command line or changes to the structure of the files used to generate the instantiation. For example, if the user specified an include directory of old_include on the initial compilation and later specified an include directory of new_include, this approach would not recognize that different files were being included.

Another approach to improving application build performance is to support a build facility that can make use of template information in determining dependency. Currently, each user-specified object file is dependent on all the included files necessary to create instantiation object files for template requests. When a change is made to a template definition, all the sources that reference the template need to be recompiled. A build facility designed to be sensitive to template instantiation could detect that a change in the template definition was limited to the instantiation object file. It could then instruct the compiler to suppress the regeneration of object files for source files that are only being recompiled due to the change in the template instantiation. Such a facility could also suppress the recompilation of any source file that would only reproduce the changes to instantiations that were already regenerated.

Because we recognize that link-time instantiation can perform better in some cases than the compile-time approach, we are investigating the link-time instantiation model as a user option.

Finally, we continue to look at ways to reduce the cost of generating each instantiation. For example, by default the compiler compresses the generated object files. Although most instantiation object files are small, many of them are potentially generated in a single compilation. As a result, the time to compress all the instantiation object files can be significant. Improvements such as not compressing small object files and/or improving the algorithm of the object file compression implementation itself could yield significant performance improvement. In addition to improvements that would reduce the overhead of generating instantiations, we are also researching ways to reduce the number of instantiation object files. For example, we might combine all the virtual functions of a class into a single instantiation object file in the repository.

Summary

As with most engineering problems, no single approach to the automatic instantiation of templates is optimal for all potential uses of templates. Based on our experience with providing template support in DIGITAL C++, we chose to implement a compile-time automatic template instantiation scheme for version 6.0 that generates instantiation object files into a repository. This choice allows users to better control when template instantiation occurs. In addition, it provides a substantial improvement in performance of template instantiation over version 5.6 and reduces the restrictions on the location of template declarations and definitions. We continue to investigate the template-instantiation implementation to further improve compile- and link-time performance and ease of use.

Acknowledgment

The authors wish to acknowledge Bevin Brett, who contributed substantially to the design and implementation of the needed walk and instantiation object file generation for DIGITAL C++ version 6.0, and Hemant Rotithor, who provided the performance measurements for DIGITAL C++ version 6.0 versus version 5.6. The authors also wish to acknowledge Charlie Mitchell, Coleen Phillimore, Rich Phillips, and Harold Seigel for their contributions to the design and implementation of the DIGITAL C++ automatic template instantiation.

References

1. ISO/IEC Standard 14882, Programming Language C++, 1998.
2. B. Stroustrup, *The C++ Programming Language,* Third Edition (Reading, Mass.: Addison-Wesley, 1997).
3. Microsoft Visual C++ 5.0, On-line Help, "Templates, C++."
4. Microsoft Corporation, "Microsoft Portable Executable and Common Object File Format Specification," Revision 5.0, Section 5.5.6, *Microsoft Developer's Network* (October 1997).
5. G. McCluskey, "An Environment for Template Instantiation," *The C++ Report,* vol. 4, no. 2 (1992).
6. G. McCluskey and R. Murray, "Template Instantiation for C++," *Sigplan Notices,* vol. 27, no. 12 (1992): 47–56.
7. Edison Design Group, "Template Instantiation in the EDG C++ Front End," Note to the ANSI C++ Committee, X3J16/95-0163, WG21/N0763.
8. M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual* (Reading, Mass.: Addison-Wesley, 1990).
9. B. Stroustrup, *The Design and Evolution of C++* (Reading, Mass.: Addison-Wesley, 1994): 366.
10. B. Stroustrup, *The C++ Programming Language,* Third Edition (Reading, Mass.: Addison-Wesley, 1997): 203–205.

Biographies

Avrum E. Itzkowitz
Avrum Itzkowitz was a contractor/consultant at DIGITAL from September 1995 through December 1997. During that time, he worked as part of the DIGITAL C++ development team, designing and implementing much of the support for the automatic template instantiation facility in DIGITAL C++ version 6.0. Avrum also designed and implemented template instantiation tests. He is currently a senior software architect engineer at GTE Internetworking. He holds a B.S. (1972) in electrical engineering from Northwestern University and M.S. (1976) and Ph.D. (1979) degrees in computer science from the University of Illinois. Avrum is a member of the ACM, the IEEE-Computer Society, and SIGPLAN.

Lois D. Foltan
Lois Foltan is a principal software engineer at Compaq. Her areas of expertise include support for C++ automatic template instantiation and the DIGITAL C++ object model. She was a member of the DEC C/C++ compiler team for eight years. During that time, she contributed to the first GEM-based DEC C and DEC C++ compilers. Recently, she joined the Digital Java team. Lois received a B.S. in computer science from the University of Vermont in 1988.

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

Measurement and Analysis of C and C++ Performance

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 literature[2-4] 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 precompiled 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 implementations 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 NULLSTONE[6] 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 benchmark[8]).

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 compatibility 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 functions). 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 tool[9–11] 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
collevol	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 standard[15] 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. Kerninghan 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 b136    |
|                  |                             | no restrict                 |
| 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

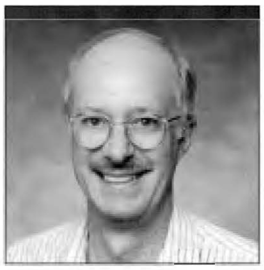
```
===============================================================================
|                  NULLSTONE SUMMARY PERFORMANCE REGRESSION REPORT            |
|                              Nullstone Release 3.9b2                        |
+-----------------------------------------------------------------------------+
| Threshold: Nullstone Ratio Decreased by at least 10%                        |
+-----------------+---------------------------+-------------------------------+
|                 |     Baseline Compiler     |     Comparison Compiler      |
+-----------------+---------------------------+-------------------------------+
| Compiler        | GCC 2.7.2                 | DEC Alpha C 5.7-123 bl36     |
|                 |                           | no restrict                  |
| Architecture    | DEC Alpha                 | DEC Alpha                    |
| Model           | 3000/300                  | 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.

August G. Reinig

Alias Analysis in the DEC C and DIGITAL C++ Compilers

During alias analysis, the DEC C and DIGITAL C++ compilers use source-level type information to improve the quality of code generated. Without the use of type information, the compilers would have to assume that any assignment through a pointer expression could modify any pointer-aliased object. In contrast, through the use of type information, the compilers can assume that such an assignment can modify only those objects whose type matches that referenced by the pointer.

When two or more address expressions reference the same memory location, these address expressions are aliases for each other. A compiler performs alias analysis to detect which address expressions do not reference the same memory locations. Good alias analysis is essential to the generation of efficient code. Code motion out of loops, common subexpression elimination, allocation of variables to registers, and detection of uninitialized variables all depend upon the compiler knowing which objects a load or a store operation could reference.

Address expressions may be symbol expressions or pointer expressions. In the C and C++ languages, a compiler always knows what object a symbol expression references. The same is not true with pointer expressions. Determining which objects a pointer expression may reference is an ongoing topic of research.

Most of the research in this area focuses on the use of techniques that track which object a pointer expression might point to.[1,2] When these techniques cannot make this determination, they assume that the pointer expression points to any object whose address has been taken. These techniques generally ignore the type information available to the source program. The best techniques perform interprocedural analysis to improve their accuracy. Although effective, the cost of analyzing a complete program can make this analysis impractical.

In contrast, the DEC C and DIGITAL C++ compilers use high-level type information as they perform alias analysis on a routine-by-routine basis. Limiting alias analysis to within a routine reduces its cost, albeit at the cost of reducing its effectiveness.

The use of this type information results in slight improvements in the performance of some standard-conforming C and C++ programs. These improvements come at little expense in terms of compilation time. There is, however, a risk that the use of this type information on nonstandard-conforming C or C++ programs may result in the compiler producing code that exhibits unexpected behavior.

The C and C++ Type Systems

Research available on the use of type information during alias analysis involves languages other than C and C++.[3] Traditionally, C is a weakly typed language. A pointer that references one type may actually point to an object of a different type. For this reason, most alias-analysis techniques ignore type information when analyzing programs written in C.

The ISO Standard for C defines a much stronger typing system.[4] In ISO Standard C, a pointer expression can access an object only if the type referenced by the pointer meets the following criteria:

- It is compatible with the type of the object, ignoring type qualifiers and signedness.
- It is compatible with the type of a member of an aggregate or union or submembers thereof, ignoring type qualifiers and signedness.
- It is the char type.

Thus, in Figure 1, the pointer p can point to A, B, C, or S (through S.sub.m) but not to T or F. The pointer q, being a pointer to char, can refer to any of A, B, C, S, T, or F.

The proposed ISO Standard for C++ defines a similar typing system for C++.[5] The strength of the Standard C and C++ type systems allows the DEC C and DIGITAL C++ compilers to use type information during alias analysis.

Many existing C applications do not conform to the Standard C typing rules. They use cast expressions to circumvent the Standard C type system. To support these applications, the DEC C compiler has a mode whereby it ignores type information during alias analysis. The DIGITAL C++ compiler also has such a mode. This mode exists to support those C++ programmers who circumvent the C++ type system.

```
int A;
signed int const B;
unsigned int volatile C;
struct {
   struct {
      int m;
   } sub;
} S;
struct {
   short z;
} T;
float F;

int *p;
char *q;
```

Figure 1
Code Fragment Associated with the Explanation of the Standard C Aliasing Rules

The Side-effects Package

The DEC C and DIGITAL C++ compilers are GEM compilers.[6] The GEM compiler system includes a highly optimizing back end. This back end uses the GEM data access model to determine which objects a load or a store may access. GEM compiler front ends augment the GEM data access model with a side-effects package, i.e., an alias-analysis package. The side-effects package provides the GEM optimizer additional information about loads and stores using language-specific information otherwise unavailable to the GEM optimizer.

The DEC C and DIGITAL C++ compilers share a common side-effects package. The DEC C and C++ side-effects package

- Determines which symbols, types, and parts thereof a routine references
- Determines the possible side effects of these references
- Answers queries from the GEM optimizer regarding the effects and dependencies of memory accesses

Preserving Memory Reference Information

The DEC C and DIGITAL C++ front ends perform lexical analysis and parsing of the source program, generating a GEM intermediate language (GEM IL) graph representation of the source program.[6] A *tuple* is a node in the GEM IL and represents an operation in the source program.

As the DEC C and DIGITAL C++ front ends generate GEM IL, they annotate each fetch (read) and store (write) tuple with information describing the object being read or written. The front ends annotate fetches and stores of symbols with information about the symbol. They annotate fetches and stores through pointers with information about the type the pointer references. The annotation information includes information describing exactly which bytes of the symbol or type the tuple accesses. This allows the side-effects package to differentiate between access to two different members of a structure.

Arrays Neither the DEC C nor the DIGITAL C++ front end differentiates between accesses to different elements of an array. Both assume that all array accesses are to the first element of the array. The GEM optimizer does extensive analysis of array references.[7] Being flow insensitive, the DEC C and C++ side-effects package can, at best, differentiate between two array references that both use constant indices. The GEM optimizer can do much more.

What the GEM optimizer cannot do, however, is determine that an assignment through a pointer to an int does not change any value in an array of doubles. This is the purpose of the DEC C and C++ side-effects package. Mapping all array accesses to access the first element of an array does not hinder this purpose and simplifies alias analysis of arrays.

Tuple Annotation Example For the program fragment in Figure 2, the DEC C and DIGITAL C++ front ends generate the annotated tuples displayed in Table 1.

Intraprocedural Effects Analysis

The GEM optimizer makes several optimization passes over a routine. During each optimization pass, the DEC C and C++ side-effects package provides alias analysis information to the GEM optimizer by means of the following procedures:

- Examining each tuple within the routine that references (reads or writes) memory, allocating effects classes that represent the memory that the tuple references
- Performing type-based alias analysis
- Responding to alias-analysis queries from the GEM optimizer

To determine the possible side effects of a memory access, the side-effects package partitions memory into effects classes. An effects class represents all or part of an object. To minimize the number of effects classes under consideration, the side-effects package creates effects classes for only those object regions referenced within the current routine.

Having created effects classes for each referenced object region within the current routine, the side-effects package then associates a signature with each effects class. The signature for an effects class records the possible side effects of referencing the effects class. The side-effects package uses this signature to respond to queries from the GEM optimizer about the effects and dependencies of tuples and symbols within the current routine.

Allocating Effects Classes There are two kinds of effects classes. The first kind represents a region of an individual object. The second kind represents a region of all allocated objects of a particular type. Allocated objects are those created by the `malloc()` function and its relatives or the C++ `new` operator.

As it processes the tuples within a routine, the side-effects package examines the memory reference information associated with the tuple. The side-effects package creates an effects class for each different set of memory reference information it encounters. Two sets of memory reference information are different if they contain different start- or end-offset information or different symbol information.

Two sets of memory reference information that contain different type information are different only if the two types are not effects equivalent. Two types are effects equivalent if they differ only in their signedness or their type qualifiers. The signed int type and the volatile unsigned int type are effects equivalent. An assignment through a pointer to a signed int may change the value of a volatile unsigned int.

Typically, an effects class represents a complete object or an individual member of a structure. An effects class may represent a subregion of the region represented by another effects class. This occurs whenever code references a whole structure as well as individual members of the structure. In the case of unions,

```
struct S {
   int x;
   int y;
} v1, v2;
int i;
double d[3];
struct S *p;

p->x = 3;
v1.y = 3;
v2 = v1;
d[i] = d[0];
```

Figure 2
Code Fragment Associated with Tuple Annotation Example

Table 1
Tuple Annotations

C/C++ Source Expression	Tuple	Annotation Symbol	Annotation Type	Start Byte	End Byte
	Fetch p	p	struct S *	0	7
p->x = 3;	Store p->x	none	struct S	0	3
v1.y = 3	Store v1.y	v1	struct S	4	7
	Fetch v1	v1	struct S	0	7
v2 = v1	Store v2	v2	struct S	0	7
	Fetch d[0]	d	double	0	7
d[i] = d[0]	Fetch i	i	int	0	3
	Store d[i]	d	double	0	7

if two members occupy exactly the same memory locations, a single effects class represents both members.

For the program fragment in Figure 3, the side-effects package creates the effects classes displayed in Table 2.

There is only one effects class for *uip and *ip since uip and ip may point to the same object. There are no effects classes for bytes 0 through 3 of s and struct S as there are no references to s.x or sp->x. By allocating effects classes for only those object regions referenced within the routine, the side-effects package greatly reduces both the number of effects classes and the time required to perform alias analysis.

In the traditional C type system, a pointer expression may point to anything, regardless of type. To represent this, the side-effects package creates exactly one effects class to represent allocated objects. It ignores the type and the start- and end-offset information.

```
struct S {
   int x;
   struct T {
       int y;
       float z;
   } t;
} s;
struct S *sp;
signed int *ip;
unsigned int *uip;
float *fp;

*uip = *ip;
*fp = 2;
sp->t = s.t;
sp->t.y = 2;
s = *sp;
```

Figure 3
Code Fragment Associated with Allocating Effects Classes

Using the traditional C type system, for the program fragment shown in Figure 3, the side-effects package creates the effects classes displayed in Table 3. Here, effects class 7 replaces effects classes 7 through 11 in Table 2. All the differentiation by types disappears.

Effects-class Signatures Having created the effects classes, the side-effects package associates a signature with each effects class. In addition, it associates an effects-class signature with each tuple within the routine and each symbol referenced within the routine.

An effects-class signature records the possible side effects of referencing an effects class. A reference to one effects class may reference another effects class. The effects class for a load through a pointer to an int indicates that the load references an allocated int object. The pointer to an int may actually reference a pointer-aliased int symbol or an int member of a structure or union.

An effects-class signature is a subset of all the effects classes that might be referenced by a tuple. There is only one requirement for an effects-class signature: If two tuples may refer to the same part of memory, the intersection of their respective effects-class signatures must be non-null. If two tuples cannot refer to the same part of memory, it is desirable that the intersection of their effects-class signatures is null. An empty intersection leads to more optimization opportunities.

The most obvious rule for building an effects-class signature is to include in it all the effects classes that might be touched by a reference to the effects class. This leads to suboptimal code in cases such as that shown in Figure 4.

There are three effects classes for this code, s<0,3>, s<4,7>, and s<0,7>, generated by references to s.x, s.y, and s, respectively. If the effects-class signature for s<0,3> includes both s<0,3> and s<0,7> and the effects-class signature for s<4,7> includes both s<4,7> and s<0,7>, then the intersection of these two effects-

Table 2
Effects Classes Using the Standard C Type Rules

Effects Class	Type or Symbol	Start Offset	End Offset	Source Generating Effects Class
1	s	0	11	s
2	s	4	11	s.t
3	sp	0	7	sp
4	fp	0	7	fp
5	ip	0	7	ip
6	uip	0	7	uip
7	struct S	0	11	*sp
8	struct S	4	11	sp->t
9	struct S	4	7	sp->t.y
10	float	0	3	*fp
11	int	0	3	*uip and *ip

Table 3
Effects Classes Using the Traditional C Type Rules

Effects Class	Type or Symbol	Start Offset	End Offset	Source Generating Effects Class
1	s	0	11	s
2	s	4	11	s.t
3	sp	0	7	sp
4	fp	0	7	fp
5	ip	0	7	ip
6	uip	0	7	uip
7	char	0	1	*sp, sp->t, *uip, sp->t.y, *fp, *ip

class signatures is non-null. This falsely indicates that s.x and s.y may refer to the same memory location. This forces GEM to generate code that stores s.y after storing to s.x.

The DEC C and C++ side-effects package uses more effective rules for building effects-class signatures. These rules offer more optimization opportunities while preserving necessary dependency information.

Effects-class Signatures for Symbols If an effects class represents a region A of a symbol, its signature includes itself. Its signature also includes all effects classes representing regions of the symbol wholly contained within A. Finally, it includes any effects class representing a region of the symbol that partially overlaps A. It does not include effects classes representing regions of the symbol that do not overlap A or that wholly contain A.

Table 4 gives the symbol effects-class signatures for the three effects classes under discussion.

The inclusion of subregions in an effects-class signature means that references to symbols interfere with references to members therein and vice versa. Excluding super-regions in an effects-class signature means that

```
struct S {
   int x;
   int y;
} s;
s.x = ...;
s.y = ...;
return s;
```

Figure 4
Example of Problematic Code for the Naïve Rule for Building Effects-class Signatures

Table 4
Symbol Effects-class Signatures

Effects Class	Effects-class Signature
s<0,3>	s<0,3>
s<4,7>	s<4,7>
s<0,7>	<0,3>, s<4,7>, s<0,7>

references to two separate members of a symbol do not interfere with each other. In Table 4, the effects-class signatures for s<0,3> and s<4,7> do not interfere with each other. Both signatures interfere with the effects-class signature for s<0,7>.

The inclusion of effects classes representing partially overlapping regions of a symbol allows for the correct representation of the side effects of referencing submembers of complex unions.

Effects-class Signatures for Types If an effects class represents a region of a type, the contents of its signature depends upon the type. If the type is the char type, the effects-class signature contains all the effects classes representing regions of other types or pointer-aliased symbols. This reflects the C and C++ type rules, which state that a pointer to a char can point to anything.

If the type is some type T other than char, the effects-class signature contains effects classes representing:

- Those regions of T that overlap the region of T the effects class represents, using the same overlap rules as for symbols
- Any region of a pointer-aliased symbol whose type is compatible to T, ignoring type qualifiers and signedness
- A region of a pointer-aliased aggregate or union symbol that contains a member or submember whose type is compatible to T, ignoring type qualifiers and signedness
- A region of an aggregate or union type that contains a member or submember whose type is compatible to T, ignoring type qualifiers and signedness

Table 5 gives the signatures for the effects classes in Table 2, assuming that the symbol s is pointer aliased.

Including the effects classes of symbols in the effects-class signatures of types records the interference of references through pointers with references to pointer-aliased symbols. In Figure 3, the pointer uip points to an unsigned int. The member s.t.y has type int. Thus, uip may point to s.t.y. The member s.t contains s.t.y. Thus, the signature for the effects-class int<0,3> con-

Table 5
Type Effects-class Signatures

Number	Effects Class	Effects-class Signature
1	s<0,11>	1, 2
2	s<4,11>	2
3	sp<0,7>	3
4	fp<0,7>	4
5	ip<0,7>	5
6	uip<0,7>	6
7	struct S<0,11>	1, 2, 7, 8, 9
8	struct S<4,11>	1, 2, 8, 9
9	struct S<4,7>	1, 2, 9
10	float<0,3>	1, 2, 7, 8, 10
11	int<0,3>	1, 2, 7, 8, 9, 11

tains the effects-class s<4,11>. This means that the load of s.t depends upon the store through uip.

Including the effects classes of types in the signatures of the effects classes of other types records the interference of references through a pointer with references through pointers to other types. In Figure 3, the pointer fp points to a float object. The member sp->t.z has type float. Thus, fp may point to sp->t.z. The member sp->t contains sp->t.z. Thus, the signature for the effects-class float<0,3> contains the effects-class struct S<4,11>. This reflects the fact that the store to sp->t.y depends upon the store through fp, i.e., it must occur after the store through fp.

Even though the signature for the effects-class float<0,3> contains the effects-class struct S<4,11> (sp->t), it does not contain the effects-class struct S<4,7> (sp->t.y). There is no float member of struct S whose position within struct S overlaps bytes 4 through 7 of struct S. There is a float member of struct S, namely z, whose position within struct S overlaps bytes 4 through 11 of struct S. The signature for the effects-class float<0,3> would not contain the effects-class s<0,3> if it existed. There is no float member of s whose position overlaps bytes 0 through 3 of s.

Additional Effects-class Signatures The side-effects package creates a special effects-class signature representing the side effects of a call. A called procedure may reference the following:

- Any pointer-aliased symbol (by means of a reference through a pointer)
- Any allocated object (by means of a reference through a pointer)
- Any nonlocal symbol (by means of direct access)
- Any local static symbol (by means of recursion)

The effects signature for a call includes all the effects classes representing these objects.

Responding to Optimizer Queries During optimization, the optimizer makes two types of queries to the side-effects analysis routines: dominator-based queries and nondominator-based queries.

When doing nondominator-based optimizations, the optimizer uses a bit vector to represent those objects a write may change (its effects). A similar bit vector represents those objects whose value a read may fetch (its dependencies). Each bit in the bit vector represents an effects class. If a tuple's effects-class signature contains an effects class, that effects class's bit is set in the tuple's bit vector. The optimizer uses the union of the bit vectors associated with a set of tuples to represent the combined effects or dependencies of those tuples.

Dominator-based queries involve finding the nearest dominating tuple that might write to the same memory location as the tuple in question. Tuple A dominates tuple B if every path from the start of the routine to B goes through A.[8] If both tuples A and C dominate B, tuple A is the nearer dominator if C dominates A.

When doing dominator-based optimizations, the side-effects package represents the tuples in the current dominator chain as a stack, adding and removing tuples from the stack as GEM moves from one path in the routine's dominator tree to another. Searching a single stack for the nearest dominating tuple that might write the same memory as the tuple in question references could lead to $O(N^2)$ performance, where N is the number of tuples in the dominator chain. This worst-case behavior occurs when none of the tuples in a dominator chain affects any subsequent tuple in the chain. Each time the side-effects package searches the stack, it examines all the tuples in the stack.

To avoid this, the DEC C and C++ side-effects package creates a stack for each effects class. When pushing a tuple, the side-effects package pushes the tuple on each stack associated with an effects class in the tuple's effects-class signature. When the GEM optimizer tells the side-effects package to find the nearest dominating write for a tuple, the side-effects package need only choose the nearest of those tuples that are on the top of the stacks associated with the tuple's effects-class signature. It need only look at the top of each stack, because a tuple would not be in the stack unless it might affect objects in the effects class associated with the stack.

The multistack worst-case behavior is $O(NC)$. There are C separate stacks, one for each effects class. The effects-class signature for each effects class may contain all the other effects classes. This would mean that each of the N tuples in the dominator chain would appear in each of the stacks.

Although the worst-case behavior for the multistack case is no better than the single-stack case (C may be equal to N), in practice there are often more tuples within a routine than effects classes. Furthermore, effects-class signatures often contain a small number of effects classes. A small number of effects classes in an effects-class signature means that there are a small number of stacks to consider. Choosing the nearest dominator from among the top tuples on these stacks requires examining only a small number of tuples.

Cost of Using Type Information

When compiling all of the SPECint95 test suite[9] using high optimization, alias analysis accounts for approximately 5 percent of the compilation time. The use of Standard C type rules during alias analysis increases compilation time by less than 0.2 percent (time measured in number of cycles consumed by the compiler as reported by Digital Continuous Profiling Infrastructure [DCPI][10]). The increase in compilation time varies from program to program but never exceeds 0.5 percent. Handling the extra effects classes generated by using Standard C type aliasing information accounted for most of the increase.

Potentially, the cost of including type-aliasing information could be huge. Calculating which effects classes a reference through a char * pointer could touch is straightforward as shown by the algorithm in Figure 5.

A much more complicated process is required to calculate which effects classes could be touched by a reference through a pointer to a type other than char. The algorithm in Figure 6 performs this process.

Fortunately, the innermost section of this loop is rarely executed. The innermost section executes only if a routine references a structure either through a pointer or a pointer-aliased symbol, that structure contains a substructure, and the routine references the substructure through a pointer.

Effectiveness

The benchmark programs from the SPECint95 suite offer some convenient test cases for measuring the effectiveness of type-based alias analysis. The sources are readily available and portable. The programs conform to alias rules established by the American National Standards Institute (ANSI) and are compute intensive. Unfortunately, they do not contain floating-point calculations. This reduces the number of different types used in the programs. Type-based alias analysis works best when there are many different types in use.

Three of the SPECint95 programs show no improvement when compiled using the Standard C typing rules as opposed to using the traditional C typing rules. These programs, namely compress, go, and li, do not use many different types and pointers to them. When all the pointers in a program are pointers to ints (go), there is only one effects class for all pointer accesses. Because the compiler has no way to differentiate among the objects touched by a dereference of a pointer expression, it generates identical code for these programs, regardless of the type rules used. The generated code for li differs only slightly and only for infrequently executed routines.

Changes in generated code for the remaining five benchmarks are more prevalent. Two benchmarks, ijpeg and perl, show a small reduction in the number of loads executed but no meaningful reduction in the total number of instructions executed. The other three SPECint95 benchmarks show varying degrees of reduction in both the number of loads executed (see Table 6) and the total number of instructions executed (see Table 7).

```
foreach pointer aliased symbol
   foreach effects class representing a region of the symbol
       add that effects class to the effects class signature for char
```

Figure 5
Calculation of the Effects-class Signature of the Type char *

```
foreach pointer aliased symbol or type referenced through a pointer
   foreach member therein
      if the member's type is referenced through pointer
         foreach effects class representing a region of the member's type
            foreach effects class representing a region of the symbol or type
                           referenced through a pointer
               if the two effects class regions overlap
                  add the symbol's or pointer's effects class to the effects
                           class signature associated with the effect class
                           representing the member's type
```

Figure 6
Calculation of the Effects-class Signature for Types Other Than char

Table 6
Number of Loads Executed by the Select SPECint95 Benchmarks

SPEC Benchmark	Millions of Loads Using Type Information	Millions of Loads without Type Information	Percent Reduction
gcc	10,268	10,365	0.9
ijpeg	16,853	16,888	0.2
m88ksim	13,889	14,157	1.9
perl	11,260	11,296	0.3
vortex	18,994	19,207	1.1

Table 7
Number of Instructions Executed by the Select SPECint95 Benchmarks

SPEC Benchmark	Millions of Instructions Using Type Information	Millions of Instructions without Type Information	Percent Reduction
gcc	42,830	42,935	0.2
ijpeg	82,844	82,834	0.0
m88ksim	72,490	73,155	0.9
perl	45,219	45,252	0.1
vortex	80,093	80,607	0.6

The load and instruction counts are those reported by using Atom's pixie tool on the SPECint95 binaries to generate pixstat data.[11,12] The compiler used was a development C compiler. All compilations used the following switches: `-fast`, `-O4`, `-arch ev56`, and `-inline speed`. The compilations using the Standard C type system used the `-ansi_alias` switch. The compilations using the traditional C type system used the `-noansi_alias` switch. The benchmark binaries were run using the reference data set.

DCPI[10] measurements of the reduction in the number of cycles consumed by these SPECint95 benchmarks showed no consistent reductions. Run-to-run variability in the data collected swamped any cycle-time reductions that might have occurred. Similarly, measurements of gains in SPECint95[9] results due to the use of type information during alias analysis showed no significant changes.

Changes in Generated Code

The code-generation changes one sees in the SPECint95 benchmarks are exactly what one would expect.

The use of type information during alias analysis reduces the number of redundant loads. An example of this occurs in ijpeg, which contains the code sequence:

```
main->rowgroup_ctr
   = (JDIMENSION)(cinfo->min_DCT_scaled_size + 1);
main->rowgroups_avail
   = (JDIMENSION)(cinfo->min_DCT_scaled_size + 2);
```

in process_data_context. Using the traditional C type system, the compiler must assume that main->row group_ctr is an alias for cinfo->min_DCT_scaled_size. Thus, it must generate code that loads cinfo->min_DCT_scaled_size twice. The Standard C type system allows the compiler to generate only one load of cinfo->min_DCT_scaled_size.

Several of the benchmarks contain code similar to the following from conversion_recipe in gcc:

```
curr.next->list->opcode = -1;
curr.next->list->to = from;
curr.next->list->cost = 0;
curr.next->list->prev = 0;
```

Using traditional C type rules, the compiler must generate four loads of curr.next->list. The compiler must assume that the pointer curr.next->list may point to itself, making curr.next->list->member an alias for curr.next->list. The Standard C type rules allow the compiler to assume that curr.next->list does not point to itself. This allows the compiler to generate code that reuses the result of the first load of curr.next->list, eliminating three redundant loads.

In another example in gcc, the use of Standard C type rules allows the compiler to move a load outside a loop. The following loop occurs in fixup_gotos:

```
for (; lists; lists = TREE_CHAIN (lists))
  if (TREE_CHAIN (lists)
        == thisblock->data.block.outer_cleanups)
     TREE_ADDRESSABLE (lists) = 1
```

Standard C type rules tell the compiler that the store generated by TREE_ADDRESSABLE (lists) = 1 cannot modify thisblock->data.block.outer_cleanups. This allows the compiler to generate code that fetches thisblock->data.block.outer_cleanups once before entering the loop. Using traditional C type rules, the compiler must generate code that fetches thisblock->data.block.outer_cleanups each time it traverses the loop.

Not only can type information reduce the number of redundant loads, it can reduce the number of redundant stores. In m88ksim, there are many routines similar to the following:

```
int ffirst(struct instruction *cmd, union opcode *ptr) {
  ...
  ptr->gen.opc1 = 0x3d;
  ptr->gen.dest = operands.value[0];
  ptr->gen.opc2 = cmd->opc.rrr;
  ptr->gen.src2 = operands.value[1];
  return(0);
}
```

where opc1, dest, opc2, and src2 are bit fields sharing the same 32 bits (longword). Using traditional C typing rules, ptr->gen and cmd->opc may be aliases for each other. Thus to implement the above routine, the compiler must generate code that performs the following actions:

- Load ptr->gen
- Update bit fields ptr->gen.opc1 and ptr->gen.dest
- Store ptr->gen
- Load cmd->opc.rrr
- Update bit fields ptr->gen.opc2 and ptr->gen.src2
- Store ptr->gen

Using Standard C typing rules, the compiler does not have to generate the first store of ptr->gen. The assignments to ptr->gen.opc1 and ptr->gen.dest cannot change cmd->opc.rrr. In this case, alias analysis that is not type based would have a difficult time detecting that ptr->gen and cmd->opc do not alias each other. M88ksim never calls ffirst directly. It calls it by means of an array-indexed function pointer.

A Note of Caution

Many C programs do not adhere to the Standard C aliasing rules. Through the use of explicit casting and implicit casting, they access objects of one type by means of pointers to other types. More aggressive optimization by GEM combined with more detailed alias-analysis information from the DEC C and C++ side-effects package increasingly results in these programs exhibiting unexpected behavior when the compiler uses Standard C aliasing rules.

Passing a pointer to one type to a routine that expects a pointer to another type works as expected, until the GEM optimizer inlines the called procedure. If the procedure is not inlined, the DEC C and C++ side-effects package must assume that the call conflicts with all pointer accesses before and after the call. Once GEM inlines the routine, the side-effects package is free to assume that references using the inlined pointer do not conflict with references using the pointer at the call site. The two pointers point to two different types.

A recent example of this problem occurred in the gcc program in the SPECint95 benchmark suite. All programs in this suite are supposed to conform to the Standard C type-aliasing rules. Because of an improvement to the GEM optimizer, this benchmark started to give unexpected results. In rtx_alloc, gcc clears a structure by treating it as an array of ints, assigning zero to each element of the array. Subsequent to zeroing this structure, gcc assigns a value to one of the fields in the structure. Through a series of valid optimizations (given the incorrect type information), the resulting code did not clear all the fields in the structure. This left uninitialized data in the structure, resulting in gcc behaving in an unexpected manner.

To avoid potential problems, the DEC C compiler, by default, does not use the Standard C type rules when performing alias analysis. The user of the compiler has to explicitly assert that the program does follow the Standard C type rules through the use of a command-line switch.

The DIGITAL C++ compiler does assume that the C++ program it is compiling adheres to the Standard C++ type rules. A user of the DIGITAL C++ compiler can use a command-line switch to inform the compiler that it should use traditional C type rules when performing alias analysis.

Summary

Using Standard C type information during alias analysis does improve the generated code for some C and C++ programs. The compilation cost of using type information is small. Except for rare cases, performance gains resulting from these code improvements are small. Any programs compiled using type information during alias analysis must strictly adhere to the Standard C and C++ aliasing rules. If not, the optimizer may generate code that produces unexpected results.

Acknowledgments

The author would like to thank Dave Blickstein, Mark Davis, Neil Faiman, Steve Hobbs, and Bill Noyce of the GEM team for their advice and reviews of this work. Dave Blickstein and Neil Faiman also did work in the GEM optimizer to ensure that the DEC C and C++ side-effects package had all the information it needed to do alias analysis correctly and to ensure that the GEM optimizer effectively used the information the side-effects package provided. Thanks also to John Henning of the CSD Performance Group and Jeannie Lieb of the GEM team for their help using the SPECint95 benchmark suite. A final word of thanks goes to Bob Morgan for suggesting that I write this paper and to my management for supporting my doing so.

References and Notes

1. R. Wilson and M. Lam, "Efficient Context-Sensitive Pointer Analysis for C Programs," *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation,* La Jolla, Calif. (June 1995): 1–12.
2. D. Coutant, "Retargetable High-Level Alias Analysis," *Proceedings of the 13th Annual Symposium on Principles of Programming Languages,* St. Petersburg Beach, Fla. (January 1986): 110–118.
3. A. Diwan et al., "Type-Based Alias Analysis," *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation,* Montreal, Canada (June 1998): 106–117.
4. Joint Technical Committee ISO/IEC JTC 1, "The C Programming Language," *International Standard ISO/IEC 9899:1990,* section 6.3 Expressions.
5. "Working Paper for Draft Proposed International Standard for Information Systems—Programming Language C++," WG21/N1146, November 1997, section 3.10.
6. D. Blickstein et al., "The GEM Optimizing Compiler System," *Digital Technical Journal,* vol. 4, no. 4 (Special Issue, 1992): 121–136.
7. R. Crowell et al., "The GEM Loop Transformer," *Digital Technical Journal,* vol. 10, no. 2, accepted for publication.
8. A. Aho, R. Sethi, and J. Ullman, *Compilers Principles, Techniques, and Tools* (Reading, Mass: Addison-Wesley, 1986): 104.
9. Information about the SPEC benchmarks is available from the Standard Performance Evaluation Corporation at http://www.specbench.org/.
10. J. Anderson et al., "Continuous Profiling: Where Have All the Cycles Gone?" *Proceedings of the Sixteenth ACM Symposium on Operating System Principles,* Sait-Malo, France (October 1997): 15–26.
11. A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation,* Orlando, Fla. (June 1994): 196–205.
12. *UMIPS-V Reference Manual (pixie and pixstats)* (Sunnyvale, Calif.: MIPS Computer Systems, 1990).

Biography

August G. Reinig
August Reinig is a principal software engineer, currently working on debugger support in the DIGITAL C++ compiler. In addition to his work on the DEC C and C++ side-effects package, August implemented a Java-based distributed test system for the DEC C and DIGITAL C++ compilers and a parallel build system for the DEC C and DIGITAL C++ compilers. The distributed test system simultaneously runs multiple tests on different machines and is fault tolerant. Before joining the DEC C and C++ team, he contributed to an advanced development incremental compiler project, which led to two patents, "Method and Apparatus for Software Testing Using a Testing Technique to Test Compilers" and "Method and Apparatus for Testing Software." He earned a B.S. in mathematics (magna cum laude) from Dartmouth College in 1980 and an M.S. in computer science from Harvard University in 1997. He is a member of Phi Beta Kappa.

Compiler Optimization for Superscalar Systems: Global Instruction Scheduling without Copies

Philip H. Sweany
Steven M. Carr
Brett L. Huber

The performance of instruction-level parallel systems can be improved by compiler programs that order machine operations to increase system parallelism and reduce execution time. The optimization, called instruction scheduling, is typically classified as local scheduling if only basic-block context is considered, or as global scheduling if a larger context is used. Global scheduling is generally thought to give better results. One global method, dominator-path scheduling, schedules paths in a function's dominator tree. Unlike many other global scheduling methods, dominator-path scheduling does not require copying of operations to preserve program semantics, making this method attractive for superscalar architectures that provide a limited amount of instruction-level parallelism. In a small test suite for the Alpha 21164 superscalar architecture, dominator-path scheduling produced schedules requiring 7.3 percent less execution time than those produced by local scheduling alone.

Many of today's computer applications require computation power not easily achieved by computer architectures that provide little or no parallelism. A promising alternative is the parallel architecture, more specifically, the instruction-level parallel (ILP) architecture, which increases computation during each machine cycle. ILP computers allow parallel computation of the lowest level machine operations within a single instruction cycle, including such operations as memory loads and stores, integer additions, and floating-point multiplications. ILP architectures, like conventional architectures, contain multiple functional units and pipelined functional units; but, they have a single program counter and operate on a single instruction stream. Compaq Computer Corporation's AlphaServer system, based on the Alpha 21164 microprocessor, is an example of an ILP machine.

To effectively use parallel hardware and obtain performance advantages, compiler programs must identify the appropriate level of parallelism. For ILP architectures, the compiler must order the single instruction stream such that multiple, low-level operations execute simultaneously whenever possible. This ordering by the compiler of machine operations to effectively use an ILP architecture's increased parallelism is called *instruction scheduling*. It is an optimization not usually found in compilers for non-ILP architectures.

Instruction scheduling is classified as *local* if it considers code only within a basic block and *global* if it schedules code across multiple basic blocks. A disadvantage to local instruction scheduling is its inability to consider context from surrounding blocks. While local scheduling can find parallelism within a basic block, it can do nothing to exploit parallelism between basic blocks. Generally, global scheduling is preferred because it can take advantage of added program parallelism available when the compiler is allowed to move code across basic block boundaries. Tjaden and Flynn,[1] for example, found parallelism within a basic block quite limited. Using a test suite of scientific programs, they measured an average parallelism of 1.8 within basic blocks. In similar experiments on scientific programs in which the compiler moved code across basic block boundaries, Nicolau and Fisher[2] found parallelism that ranged from 4 to a virtually unlimited number, with an average of 90 for the entire test suite.

Trace scheduling[3,4] is a global scheduling technique that attempts to optimize frequently executed paths of a program, possibly at the expense of less frequently executed paths. Trace scheduling exploits parallelism within sequential code by allowing massive migration of operations across basic block boundaries during scheduling. By addressing this larger scheduling context (many basic blocks), trace scheduling can produce better schedules than techniques that address the smaller context of a single block. To ensure the program semantics are not changed by interblock motion, trace scheduling inserts copies of operations that move across block boundaries. Such copies, necessary to ensure program semantics, are called *compensation copies.*

The research described here is driven by a desire to develop a global instruction scheduling technique that, like trace scheduling, allows operations to cross block boundaries to find good schedules and that, unlike trace scheduling, does not require insertion of compensation copies. Like trace scheduling, DPS first defines a multiblock context for scheduling and then uses a local instruction scheduler to treat the larger context like a single basic block. Such a technique provides effective schedules and avoids the performance cost of executing compensation copies. The global scheduling technique described here is based on the dominator relation* among the basic blocks of a function and is called dominator-path scheduling (DPS).

Local Instruction Scheduling

Since DPS relies on a local instruction scheduler, we begin with a brief discussion of the local scheduling problem. As the name implies, local instruction scheduling attempts to maximize parallelism within each basic block of a function's control flow graph. In general, this optimization problem is NP-complete.[5] However, in practice, heuristics achieve good results. (Landskov et al.[6] give a good survey of early instruction scheduling algorithms. Allan et al.[7] describe how one might build a retargetable local instruction scheduler.)

List scheduling[8] is a general method often used for local instruction scheduling. Briefly, list scheduling typically requires two phases. The first phase builds a directed acyclic graph (DAG), called the data dependence DAG (DDD), for each basic block in the function. DDD nodes represent operations to be scheduled. The DDD's directed edges indicate that a node X preceding a node Y constrains X to occur no later than Y. These DDD edges are based on the formalism of data dependence analysis. There are three basic types of data dependence, as described by Padua et al.[9]

- Flow dependence, also called true dependence or data dependence. A DDD node M_2 is flow dependent on DDD node M_1 if M_1 executes before M_2 and M_1 writes to some memory location read by M_2.
- Antidependence, also called false dependence. A DDD node M_2 is antidependent on DDD node M_1 if M_1 executes before M_2 and M_2 writes to a memory location read by M_1, thereby destroying the value needed by M_1.
- Output dependence. A DDD node M_2 is output dependent on DDD node M_1 if M_1 executes before M_2 and M_2 and M_1 both write to the same location.

To facilitate determination and manipulation of data dependence, the compiler maintains, for each DDD node, a set of all memory locations *used* (read) and all memory locations *defined* (written) by that particular DDD node.

Once the DDD is constructed, the second phase begins when list scheduling orders the graph's nodes into the shortest sequence of instructions, subject to (1) the constraints in the graph, and (2) the resource limitations in the machine (i.e., a machine is typically limited to holding only a single value at any time). In general list scheduling, an ordered list of tasks, called a *priority list,* is constructed. The priority list takes its name from the fact that tasks are ranked such that those with the highest priority are chosen first. In the context of local instruction scheduling, the priority list contains DDD nodes, all of whose predecessors have already been included in the schedule being constructed.

Expressions, Statements, and Operations

Within the context of this paper, we discuss algorithms for code motion. Before going further, we need to ensure common understanding among our readers for our use of terms such as *expressions, statements,* and *operations.* To start, we consider a computer program to be a list of operations, each of which (possibly) computes a right-hand side (rhs) value and assigns the rhs value to a memory location represented by a left-hand side (lhs) variable. This can be expressed as

$$A \leftarrow E$$

where A represents a single memory location and E represents an expression with one or more operators and an appropriate number of operands. During different phases of a compiler, operations might be represented as

- Source code, a high-level language such as C
- Intermediate statements, a linear form of three-address code such as quads or *n*-tuples[10]

*A basic block, D, dominates another block, B, if every path from the root of the control-flow graph for a function to B must pass through D.

- DDD nodes, nodes in a DDD, ready to be scheduled by the instruction scheduler

Important to note about operations, whether represented as intermediate statements, source code, or DDD nodes, is that operations include both a set of definitions and a set of uses.

Expressions, in contrast, represent the rhs of an operation and, as such, include uses but not definitions. Throughout this paper, we use the terms *statement, intermediate statement, operation,* and *DDD node* interchangeably, because they all represent an operation, with both uses and definitions, albeit generally at different stages of the compilation process. When we use the term *expression,* however, we mean an rhs with uses only and no definition.

Dominator Analysis Used in Code Motion

In order to determine which operations can move across basic block boundaries, we need to analyze the source program. Although there are some choices as to the exact analysis to perform, dominator-path scheduling is based upon a formalism first described by Reif and Tarjan.[11] We summarize Reif and Tarjan's work here and then discuss the enhancements needed to allow interblock movement of operations.

In their 1981 paper, Reif and Tarjan provide a fast algorithm for determining the approximate *birthpoints* of expressions in a program's flow graph. An expression's birthpoint is the first block in the control flow graph at which the expression can be computed, and the value computed is guaranteed to be the same as in the original program. Their technique is based upon fast computation of the *idef* set for each basic block of the control flow graph. The *idef* set for a block B is that set of variables defined on a path between B's immediate dominator and B. Given that the dominator relation for the basic blocks of a function can be represented as a *dominator tree,* the immediate dominator, IDOM, of a basic block B is B's parent in the dominator tree.

Expression birthpoints are not sufficient to allow us to safely move entire operations from a block to one of its dominators because birthpoints address only the movement of expressions, not definitions. Operations in general include not only a computation of some expression but the assignment of the value computed to a program variable. Ensuring a "safe" motion for an expression requires only that no expression operand move above any *possible definition* of that operand, thus changing the program semantics. A similar requirement is necessary, but not sufficient, for the variable to which the value is being assigned. In addition to not moving A above any previous definition of A, A cannot move above any possible use of A. Otherwise, we run the risk of changing A's value for that previous use. Thus, dominator analysis computes the *iuse* set for each basic block and for the *idef* set. The *iuse* set for a block, B, is that set of variables used on some path between B's immediate dominator and B. Using the *idef* and *iuse* sets, dominator analysis computes an approximate birthpoint for each operation.

In this paper, we use the term *dominator analysis* to mean the analysis necessary to allow code motion of operations while disallowing compensation copies. Additionally, we use the term *dominator motion* for the general optimization of code motion based upon dominator analysis.

Enhancing the Reif and Tarjan Algorithm

By enhancing Reif and Tarjan's algorithm to compute *birthpoints* of operations instead of expressions, we make several issues important that previously had no effect upon Reif and Tarjan's algorithm. This section motivates and describes the information needed to allow dominator motion, including the *use, def, iuse,* and *idef* sets for each basic block. An algorithmic description of this dominator analysis information is included in the section Overview of Dominator-Path Scheduling and the Algorithm for Interblock Motion.

When we allow code motion to move intermediate statements (or just expressions) from a block to one of its dominators, we run the risk that the statement (expression) will be executed a different number of times in the dominator block than it would have been in its original location. When we move only expressions, the risk is acceptable (although it may not be efficient to move a statement into a loop) since the value needed at the original point of computation is preserved. Relative to program semantics, the number of times the same value is computed has no effect as long as the correct value is computed the *last* time. This accuracy is guaranteed by expression birthpoints.

Consider also the consequences of moving an expression from a block that is *never* executed for some particular input data. Again, it may not be efficient to compute a value never used, but the computation does not alter program semantics. When dominator motion moves entire statements, however, the issue becomes more complex. If the statement moved assigns a new value to an induction variable, as in the following example,

$$n = n + 1$$

dominator motion would change *n*'s final value if it moved the statement to a block where the execution frequency differed from that of its original block. We could alleviate this problem by prohibiting motion of any statement for which the *use* and *def* sets are not disjoint, but the possibility remains that a statement may define a variable based indirectly upon that variable's previous value. To remedy the more general problem, we disallow motion of any statement, S, whose *def* set intersects with those variables that are *used-before-defined* in the basic block in which S resides.

Suppose the optimizer moves an intermediate statement that defines a global variable from a block that may never be executed for some set of input data into a dominator block that *is* executed at least once for the same input data. Then the optimized version has defined a variable that the unoptimized function did not, possibly changing program semantics. We can be sure that such motion does not change the semantics of that function being compiled; but there is no mechanism, short of compiling the entire program as a single unit, to ensure that defining a global variable in this function will not change the value used in another function. Thus, to be conservative and ensure that it does not change program semantics, dominator motion prohibits interblock movement of any statement that defines a global variable. At first glance, it may seem that this prohibition cripples dominator motion's ability to move any intermediate statements at all; but we shall see that such is not the case.

One final addition to Reif and Tarjan information is required to take care of a subtle problem. As discussed above, dominator analysis uses the *idef* and *iuse* sets to prevent illegal code motion. The use of these sets was assumed to be sufficient to ensure the legality of code motion into a dominator block; unfortunately, this is not the case. The problem is that a definition might pass through the immediate dominator of B to *reach* a *use* in a sibling of B in the dominator tree. If there were a definition of this variable in B, but the variable was not defined on any path from the immediate dominator, there would be nothing in dominator analysis to prevent the definition from being moved into the dominator. But that would change the program's semantics. Figure 1 shows the control-flow graph for a function called findmax(), with only the statements referring to register r7. Register r7 is defined in blocks B3 and B7, and referenced in B9. This means that r7 is *live-out* of B5 and *live-in* to B8, but not *live-in* to B7; there is a definition of r7 in B3 that *reaches* B8. Because there is no definition or use between B7 and its immediate dominator B5, the *idef* and *iuse* sets of B7 are empty; thus, dominator analysis, as described above, would allow the assignment of r7 to move upward to block B5. This motion is illegal; it changes the definition in B3. Moving the operation from B7 to B5 changes the conditional assignment of r7 to an unconditional one.

To prevent this from happening, we can insert the variable into the *iuse* set of the block B, in which we wish the statement to remain. We do not, however, want to add to the *iuse* set unnecessarily. The solution is to add each variable, V, that is *live-in* to any of B's siblings in the dominator tree, but not into B, or to B's

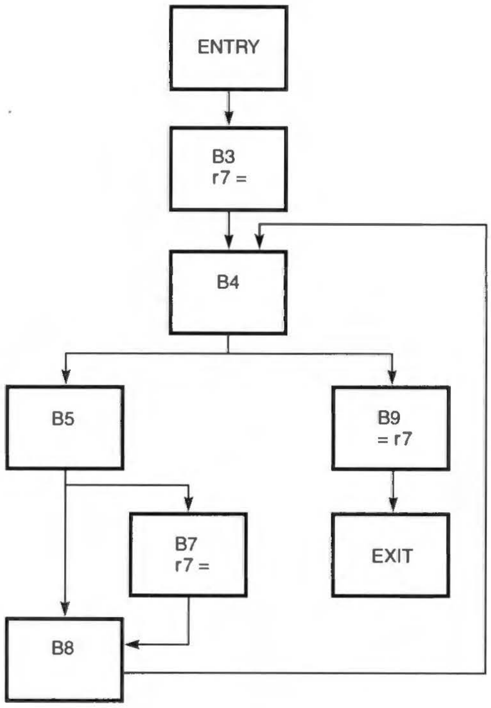


Figure 1
Control Flow Graph for the Function findmax()

iuse set. This will prevent any definition of V that might exist in B from moving up. If there is a definition of V in B, but V is *live-in* to B, there must be some use of V in B before the definition, so it could not move upward in any case.

Measurement of Dominator Motion

To measure the motion possible in C programs, Sweany[12] defined dominator motion as the movement of each intermediate statement to its birthpoint as defined by dominator analysis and by the number of dominator blocks each statement jumps during such movement. Sweany's choice of intermediate statements (as contrasted with source code, assembly language, or DDD nodes) is attributed to the lack of machine resource constraints at that level of program abstraction. He envisioned dominator motion as an upper bound on the motion available in C programs when compensation copies are included. In the test suite of 12 C programs compiled, more than 25 percent of all intermediate statements moved at least one dominator block upwards toward the root of the dominator tree. One function allowed more than 50 percent of the statements to be hoisted an average of nearly eight dominator blocks. The considerable amount of motion (without copies) available at the intermediate statement level of program abstraction provided us with the motivation to use similar analysis techniques to facilitate global instruction scheduling.

Overview of Dominator-path Scheduling and the Algorithm for Interblock Motion

Since experiments show that dominator analysis allows considerable code motion without copies, we chose to use dominator analysis as the basis for the instruction scheduling algorithm described here, namely dominator-path scheduling. As noted above, DPS is a global instruction scheduling method that does not require copies of operations that move from one basic block to another. DPS performs global instruction scheduling by treating a group of basic blocks found on a dominator tree path as a single block, scheduling the group as a whole. In this regard, it resembles trace scheduling, which schedules adjacent basic blocks as a single block. DPS's foundation is scheduling instructions while moving operations among blocks according to both the opportunities provided by and the restrictions imposed by dominator analysis.

The question arises as to how to exploit dominator analysis information to permit code motion at the instruction level during scheduling. DPS is based on the observation that we can use *idef* and *iuse* sets to allow operations to move from a block to one of its dominators during instruction scheduling. Instruction scheduling can then choose the most advantageous position for an operation that is placed in any one of several blocks. Because machine operations are incorporated in nodes of the DDD used in scheduling and, like intermediate statements, DDD nodes are represented by *def* and *use* sets, the same analysis performed on intermediate statements can also be applied to a basic block's DDD nodes.

The same motivation that drives trace scheduling—namely that scheduling one large block allows better use of machine resources than scheduling the same code as several smaller blocks—also applies to DPS. In contrast to trace scheduling, DPS does not allow motion of DDD nodes when a copy of a node is required and does not incur the code explosion due to copying that trace scheduling can potentially produce. For architectures with moderate instruction-level parallelism, DPS may produce better results than trace scheduling, because the more limited motion may be sufficient to make good use of machine resources, and unlike trace scheduling, no machine resources are devoted to executing semantic-preserving operation copies.

Much like traces,* the dominator path's blocks can be chosen by any of several methods. One method is a heuristic choice of a path based on length, nesting depth, or some other program characteristic. Another is programmer specification of the most important paths. A third is actual profiling of the running program. We visit this issue again in the section Choosing Dominator Paths. First, however, we need to discuss the algorithmic details of DPS.

Once DPS selects a dominator path to schedule, it requires a method to combine the blocks' DDDs into a single DDD for the entire dominator path. In our compiler, this task is performed by a DDD coupler,[13] which is designed for the purpose. Given the DDD coupler, DPS proceeds by repeatedly

- Choosing a dominator path to schedule
- Using the DDD coupler to combine each block's DDD on the chosen dominator path
- Scheduling the combined DDD as a single block

The dominator-path scheduling algorithm, detailed in this section, is summarized in Figures 2 and 3.

A significant aspect of the DPS process is to ensure "appropriate" interblock motion of DDD nodes and to prohibit "illegal" motion. As noted earlier, the combined DDD for a dominator path includes control flow. Therefore, when DPS schedules a group of blocks represented by a single DDD, it needs a mechanism to map correctly the scheduled instructions to the basic blocks. The mechanism is easily accomplished by the addition of two special nodes to each block's DDD. Called BlockStart and BlockEnd, these special nodes represent the basic block boundaries. Since dominator-path scheduling does not allow branches to move across block boundaries, each BlockStart and BlockEnd node is initially "tied" (with DDD arcs) to the branch statement of the block, if any. Because BlockStart and BlockEnd are nodes in the eventually combined DDD, they are scheduled like all other nodes of the combined DDD. After scheduling, all instructions between the instruction containing the BlockStart node for a block and the instruction containing the BlockEnd node for that block are considered instructions for that block. Next, DPS must ensure that the BlockStart and BlockEnd DDD nodes remain ordered (in the scheduled instructions) relative to one another and to the BlockStart and BlockEnd nodes for any other block. To do so, DPS adds *use* and *def* information to the nodes to represent a pseudoresource, BlockBoundary. Because each BlockStart node defines BlockBoundary and each BlockEnd node uses BlockBoundary, no BlockEnd node can be scheduled ahead of its associated BlockStart node (because of flow dependence.) Also, a BlockStart node cannot be scheduled before its dominator block's BlockEnd node (because of antidependence). By establishing these imaginary dependencies, DPS ensures that the DDD coupler adds arcs between all BlockStart and BlockEnd nodes.

*groups of blocks to be scheduled together in trace scheduling

```
Algorithm Dominator-Path Scheduling
Input:
        Function Control Flow Graph
        Dominator Tree
        Post-Dominator Tree

Output:
        Scheduled instructions for the function

Algorithm:
        While at least one Basic Block is unscheduled
                Heuristically choose a path $B_1, B_2, \ldots, B_n$ in the Dominator Tree that includes
                only unscheduled Basic Blocks.

                Perform dominator analysis to compute IDef and IUse sets

                        /* Build one DDD for the entire dominator path */
                CombinedDDD = $B_1$
                For $i = 2$ to $n$
                        T = InitializeTransitionDDD ($B_{i-1}$, $B_i$)
                        CombinedDDD = Couple(CombinedDDD,T)
                        CombinedDDD = Couple (CombinedDDD, $B_i$ )

                Perform list scheduling on CombinedDDD
                Mark each block of DP scheduled
                Copy scheduled instructions to the Blocks of the path (instructions between the
                BlockStart and BlockEnd nodes for a Block are "written" to that Block)
        End While
```

Figure 2
Dominator-path Scheduling Algorithm

Looking back to dominator analysis, we see that interblock motion is prohibited if the operation being moved

- Defines something that is included in either the *idef* or *iuse* set
- Uses something included in the *idef* set for the block in which the operation currently resides

To obtain the same prohibitions in the combined DDD, we add the *idef* set for a basic block, B, to the *def* set B's BlockStart node. Similarly, we add the *iuse* set for B to the *use* set of B's BlockStart node. Thus we enforce the same restriction on movement that dominator analysis imposed upon intermediate statements and ensure that any interblock motion preserves program semantics. In a similar manner, DPS includes the restrictions on movement of operations that define either global variables or induction variables. Figure 3 gives an algorithmic description of the process of "doping" the BlockStart and BlockEnd nodes to prevent disallowed code motion.

DPS is complicated by factors not relevant for dominator motion of intermediate statements. Foremost is the complexity imposed by the bidirectional motion of operations that instruction scheduling allows. In dominator motion, intermediate statements move in only one direction, i.e., toward the top of the function's control flow graph, not from a dominator block to a dominated one. This one-directional motion is reasonable when attempting to move intermediate statements because one statement's movement will likely open possibilities for more motion in the same direction by other statements. When statements move in different directions, one statement's motion might inhibit another's movement in the opposite direction. The goal of dominator motion is to move statements as far as possible in the control flow graph. In contrast, the goal of DPS is not to maximize code motion, but rather to find, for each operation, O, that location for O that will yield the shortest schedule. Thus our goal has changed from that of dominator motion. To gain the full benefit from DPS, we wish to allow operations to move past block boundaries in either direction. To permit bidirectional motion, we use the post-dominator relation, which says that a basic block, PD, is a post-dominator of a basic block B if all paths from B to the function's exit must pass through PD. Using this strategy, we similarly define *post-idef* and *post-iuse* sets. In

```
Algorithm InitializeTransitionDDD($B_1$, $B_2$)
Input:
          A Transition DDD templates, with a Dummy DDDNode
          for $B_1$'s block end and one for $B_2$'s block start
          Two basic blocks, $B_1$ and $B_2$ that we wish to couple
          Dominator Tree
          Post-Dominator Tree
          The following dataflow information
                    Def, Use, IDef, and IUse sets for $B_1$ and $B_2$
                    Used-Before-Defined set for $B_2$
                    Post-IDef, and Post-IUse sets for $B_1$ and $B_2$
                    $B_2$'s "sibling" set, defined to include any variable
                              live-in to a dominator-tree sibling of $B_2$, but not
                              live-in to $B_2$
                    A basic block DDD for each of $B_1$ and $B_2$
Output:
          An initialized Transition DDD, T
Algorithm:
          T = TransitionDDD
          /* "Fix" set for global and induction variables. */
          Add set of global variables to $B_2$'s IUse
          Add $B_2$'s Used-Before-Defined to $B_2$'s IUse
          Add $B_2$'s sibling set to $B_2$'s IUse

          If $B_2$ does not post-dominate $B_1$
                    Add $B_1$'s Use set to T's BlockEnd Def set
                    Add $B_1$'s Def set to T's BlockEnd Use set
          Else
                    Add $B_1$'s Post-IDef set to T's BlockEnd Def set
                    Add $B_1$'s Post-IUse set to T's BlockEnd Use set
          Add $B_2$'s IDef set to T's BlockStart Def set
          Add $B_2$'s IUse set to T's BlockStart Use set
          Return T
```

Figure 3
Initialize Transition DDD Algorithm

fact, it is not difficult to compute all these quantities for a function. The simplest way is to logically reverse the direction of all the control flow graph arcs and perform dominator analysis on the resulting graph. Having computed the post-dominator tree, DPS chooses dominator paths such that the dominated node is a post-dominator of its immediate predecessor in a dominator path. This choice allows operations to move "freely" in both directions. Of course, this may be too limiting on the choice of dominator paths. To allow for the possibility that nodes in a dominator path will not form a post-dominator relation, DPS needs a mechanism to limit bidirectional motion when needed. Again, we rely on the technique of adding dependencies to the combined DDD. In this case (assuming that DPS is scheduling paths in the forward dominator tree), for any basic block, B, whose successor, S, in the forward dominator path does not post-dominate B, DPS adds B's *def* set to the *use* set of the BlockEnd node associated with B. In similar fashion, we add B's *use* set to B's BlockEnd node's *def* set. This technique prevents any DDD node originally in B from moving downward in the dominator path.

Choosing Dominator Paths

DPS allows code movement along any dominator path, but there are many ways to select these paths. An investigation of the effects of dominator-path choice on the efficiency of generated schedules tells us that the choice of path is too important to be left to arbitrary selection; twice the average percent speedup* for several functions can often be achieved with a simple,

*(unoptimized_speed – optimized_speed)/unoptimized_speed

well-chosen heuristic. Some functions have a potential percent speedup almost four times the average. Thus, it is important to find a good, generally applicable heuristic to select the dominator paths.

Unfortunately, it is not practical to schedule all of the possible partitionings for large functions. If we allow a basic block to be included in only one dominator path, the formula for the number of distinct partitionings of the dominator tree is

$$\prod_{n \in N} [\text{outdeg}(n) + 1]$$

where N is the set of nodes of the dominator tree.[14] Although the number of possible paths is not prohibitive for small dominator trees, larger trees have a prohibitively large number. For example, whetstone's main(), with 49 basic blocks, has almost two trillion distinct partitionings.

To evaluate differences in dominator-path choices, we scheduled a group of small functions with DPS using every possible choice of dominator path. The target architecture for this study was a hypothetical 6-wide long-instruction-word (LIW) machine, which was simulated and in which it was assumed that all cache accesses were hits.

The results of exhaustive dominator-path testing show, as expected, that varying the choice of dominator paths significantly affects the performance of scheduling. For all functions of at least two basic blocks, DPS showed improvement over local scheduling for at least one of the possible choices of dominator paths. Table 1 shows the best, average, and worst percent speedup over local scheduling found for all functions that had a "best" speedup of over 2 percent; it also shows the speedup of the original implementation of DPS and the number of distinct dominator tree partitionings. The original implementation of DPS included a single, simple heuristic to choose dominator paths. More specifically, to choose dominator paths within a group, G, of contiguous blocks at the same nesting level, the compiler continues to choose a block, B, to "expand." Expansion of B initializes a new dominator path to include B and adds B's dominators until no more can be added. The algorithm then starts another dominator path by expanding another (as yet unexpanded) block of G. The first block of G chosen to expand is the tail block, T, in an attempt to obtain as long a dominator path as possible.

Unfortunately, not all functions are small enough to be tested by performing DPS for each possible partitioning of the dominator tree. Therefore, we defined 37 different heuristic methods of choosing dominator trees, based upon groupings of six key heuristic factors.

The maximum path lengths of the basic guidelines were adjusted to produce actual heuristics. We used the heuristic factors from which the individual heuristics were constructed; each seemed likely either to mimic the observed characteristics of the best path selection or to allow more freedom of code motion and, therefore, more flexibility in filling "gaps."

- One nesting level—Group blocks from the same nesting level of a loop. Each block is in the same strongly connected component, so the blocks tend to have similar restrictions to code motion. For a group of blocks to be a strongly connected component, there must be some path in the control flow graph from each node in the component to all the other nodes in the component. Since the function will probably repeat the loop, it seems likely that the scheduler will be able to overlap blocks in it.

Table 1
Percent of Function Speedup Improvement Using DPS Path Choices over Local Scheduling

	Percent Speedup				
Function Name	**Best**	**Average**	**Worst**	**Original**	**No. Dominator Tree Partitions**
bubble	39.2	10.6	−0.1	11.7	72
readm	32.5	9.3	−0.2	32.5	48
solve	27.8	9.9	−0.2	27.8	96
queens	25.4	8.3	−0.4	−0.4	96
swaprow	23.1	5.8	−3.7	19.5	24
print(g)	22.0	9.1	−0.2	22.0	8
findmax	21.3	6.2	−0.3	8.7	18
copycol	18.5	5.6	−5.0	19.9	8
elim	14.3	2.3	−3.8	10.2	576
mult	13.7	2.1	−3.8	10.3	96
subst	12.9	2.4	−4.9	4.9	96
print(8)	12.5	6.2	0.0	12.5	8

- Longest path—Schedule the longest available path. This heuristic class allows the maximum distance for code motion.
- Postdominator—Follow the postdominator relation in the dominator tree. When a dominator block, P, is succeeded by a non-postdominator block, S, our compiler adds P's *def* set to the *use* set of P's BlockEnd node and the *use* set to the *def* set to prevent any code motion from P to S. If P is instead succeeded by its postdominator block, no such modification is necessary, and code would be allowed to move in both directions. Intuitively, the postdominator relation is the exact inverse of the dominator relation, so code can move down, into a postdominator, as it moves up into a dominator. Further, the simple act of adding nodes to the DDD will complicate list scheduling, making it harder for the scheduler to generate the most efficient schedule.
- Non-postdominator—Follow a non-postdominator in the dominator tree. This heuristic class generally means adding loop body blocks to the path. Notice that this seems at odds with the previous heuristic class. The previous class was suggested by intuition about the scheduler, and this one by observation of path behavior.
- *idef* size—Group by *idef* set size. The larger the *idef* size, the more interference there is to code motion. A small *idef* size will probably allow more code motion, so we try to add blocks with small *idef* sizes.
- Density—Group by operation density. We define the density of each basic block as the number of nodes in the DDD divided by the number of instructions required for local scheduling. A dense block already has close to its maximum number of operations; adding or removing operations will probably not improve the schedule. For this reason, we want to avoid scheduling dense blocks together. Two methods are tried: scheduling dense blocks with sparse blocks and putting sparse blocks together.

The heuristic factors were used to make individual heuristics by changing the limit on the possible number of blocks in a path. It was reasonable to set limits for four factors: postdominator, non-postdominator, *idef* size, and density. We tried path length limits in blocks of 2, 3, 4, 5, and unlimited, making a total of five heuristics from each heuristic factor.

Running DPS using each of the heuristic methods and comparing the efficiency of the resulting code leads to several conclusions about effective heuristics for choosing DPS's dominator paths. For some heuristics, we can achieve the best schedules for DPS by using paths that rarely exceed three blocks. For any particular class of heuristics, we can achieve the best schedule with paths limited to five blocks or fewer. Consequently, path lengths can be limited without lowering the efficiency of generated code, and longer paths, which increase scheduling time, can be avoided.

Since no one heuristic performed well for all functions, we advise using a combination of heuristics, i.e., schedule by using each of three heuristics and taking the best schedule. The "combined" heuristic includes the following:

- Instruction density, limit to five blocks
- One nesting level on path, limit to five blocks
- Non-postdominator, unlimited length

Frequency-based List Scheduling

Like some other global schedulers, DPS uses a local scheduling algorithm (list scheduling) on a global context, namely the meta-blocks built by DPS. This algorithm raises the possibility of moving code from less frequently executed blocks to more frequently executed blocks. At first glance, this practice seems to be a bad idea.

In theory, to best schedule any meta-block, an instruction scheduler must account for the differing cost of the instructions within the meta-block. If a single meta-block includes multiple nesting levels, the scheduler must recognize that instructions added to blocks with higher nesting levels are more costly than those added to blocks with lower nesting levels. Even within a loop, there exists the potential for considerable variation in the execution frequencies of different blocks in the meta-block due to control flow. Of course variable execution frequency is not an issue in traditional local scheduling because, within the context of a single basic block, each DDD node is executed the same number of times, namely, once each time execution enters the block.

To address the issue of differing execution frequencies within meta-blocks scheduled as a single block by DPS, we investigated frequency-based list scheduling (FBLS),[15] an extension of list scheduling that provides an answer to this difficulty by considering that execution frequencies differ within sections of the meta-blocks. FBLS uses a greedy method to place DDD nodes in the lowest-cost instruction possible. FBLS amends the basic list-scheduling algorithm by revising only the DDD node placement policy in an attempt to reduce the run-time cycles required to execute a meta-block.

Unfortunately, although FBLS makes intuitive sense, we found that DPS produced worse schedules with FBLS than it produced with a naive local scheduling algorithm that ignored frequency differences within DPS's meta-blocks. Therefore, the current implementation of DPS ignores the execution frequency differences between basic blocks, both in choosing dominator paths to schedule and in scheduling those dominator-path meta-blocks.

Evaluation of Dominator-path Scheduling

To measure the potential of DPS to generate more efficient schedules than local scheduling for commercial superscalar architectures, we ran a small test suite of C programs on an Alpha 21164 server. The Alpha server is a superscalar architecture capable of issuing two integer and two floating-point instructions each cycle. Our compiler estimates the effectiveness of a schedule by modeling the 21164 as an LIW architecture with all operation latencies known at compile time. Of course this model was used only within the compiler itself. Our results measured changes in 21164 execution time (measured with the UNIX "time" command) required for each program.

Our test suite of 14 C programs includes 8 programs that use integer computation only and 6 programs that include floating-point computation. We separated those groups because we see dramatic differences in DPS's performance when viewing integer and floating-point programs. To choose dominator paths, we used the combined heuristic recommended by Huber.[14]

Table 2 summarizes the results of tests we conducted to compare the execution times of programs using DPS scheduling with those using local scheduling only. The table lists the programs used in the test suite and the percent improvement in execution times for DPS-scheduled programs. The execution time measurements were made on an Alpha 21164 server running at 250 megahertz with data cache sizes of 8 kilobytes, 96 kilobytes, and 4 megabytes.

Table 2
Percent DPS Scheduling Improvements over Local Scheduling of Programs

Program	Percent Execution Time Improvement
8- Queens	7.3
SymbolTable	7.3
BubbleSort	5.0
Nsieve	6.1
Heapsort	6.0
Killcache	2.6
TSP	2.4
Dhrystone	0.7
C integer average	**4.7**
Dice	3.7
Whetstone	5.4
Matrix Multiply	16.2
Gauss	12.3
Finite Difference	17.6
Livermore	9.3
C floating-point average	**10.8**
Overall average	**7.3**

Looking at Table 2, we see that, in general, DPS improved the integer programs less than it improved the floating-point programs. The range of improvements for integer programs was from 0.7 percent for Dhrystone to 7.3 percent each for 8-Queens and for SymbolTable. Summing all the improvements and dividing by eight (the number of integer programs) gives an "average" of 4.7 percent improvement for the integer programs. DPS improved some of the floating-point programs even more significantly than the integer programs. The range of improvements for the six floating-point programs was from 3.7 percent for Dice (a simulation of rolling a pair of dice 10,000,000 times using a uniform random number generator) to 17.6 percent improvement for the finite difference program. The average for the six floating-point programs was 10.8 percent. This suggests, not surprisingly, that the Alpha 21164 provides more opportunities for global scheduling improvement when floating-point programs are being compiled.

Even within the six floating-point programs, however, we see a distinct bi-modal behavior in terms of execution-time improvement. Three of the programs range from 12.3 percent to 17.6 percent improvement, whereas three are below 10 percent (and two of those significantly below 10 percent). A reason for this wide range is the use of global variables. Remember that DPS forbids the motion of global variable definitions across block boundaries. This is necessary to ensure correct program semantics. It is hardly a coincidence that both Dice and Whetstone include only global floating-point variables, whereas Livermore's floating-point variables are mixed about half local and half global, and the three better performers use almost no global variables. Thus we conclude that, for floating-point programs with few global variables, we can expect improvements of roughly 12 to 15 percent in execution time. Inclusion of global variables and exclusion of floating-point values will, however, decrease DPS's ability to improve execution time for the Alpha 21164.

Related Work

As we have discussed, local instruction scheduling can find parallelism within a basic block but cannot exploit parallelism between basic blocks. Several global scheduling techniques are available, however, that extract parallelism from a program by moving operations across block boundaries and subsequently inserting compensation copies to maintain program semantics. Trace scheduling[3] was the first of these techniques to be defined. As previously mentioned, trace scheduling requires compensation copies. Other "early" global scheduling algorithms that require compenstation copies include Nicolau's *percolation scheduling*[16,17] and Gupta's *region scheduling.*[18] A recent and quite popular extension of trace scheduling is Hwu's SuperBlock scheduling.[19,20] In addition to these more general, global scheduling methods, significant results have been obtained by software pipelining, which is a technique that overlaps iterations of loops to exploit available ILP. Allan et al.[21] provide a good summary, and Rau[22] provides an excellent tutorial on how *modulo scheduling,* a popular software pipelining technique, should be implemented. Promising recent techniques have focused on defining a meta-environment, which includes both global scheduling and software pipelining. Moon and Ebcioglu[23] present an aggressive technique that combines software pipelining and global code motion (with copies) into a single framework. Novak and Nicolau[24] describe a sophisticated scheduling framework in which to place software pipelining, including alternatives to modulo scheduling. While providing a significant number of excellent global scheduling alternatives, none of these techniques provides global scheduling without the possibility of code expansion (copy code) as DPS does.

To address the issue of producing schedules without operation copies, Bernstein[25-27] defined a technique he calls *global instruction scheduling* (GPS) that allows movement of instructions beyond block boundaries based upon the program dependence graph (PDG).[28] In a test suite of four programs run on IBM's RS/6000, Bernstein's method showed improvement of roughly 7 percent over local scheduling for two of the programs, with no significant difference for the others.

Comparing DPS to Bernstein's method, we see that both allow for interblock motion without copies. Bernstein also allows for interblock movement requiring duplicates that DPS does not. Interestingly, Bernstein's later work[27] does not make use of this ability to allow motion that requires duplication of operations, suggesting that, to date, he has not found such motion advisable for the RS/6000 architecture to which his techniques have been applied. Bernstein allows operation movement in only one direction, whereas DPS allows operations to move from a dominator block to a postdominator. This added flexibility is an advantage to DPS. Of possibly greater significance, DPS uses the local instruction scheduler to place operations. Bernstein uses a separate set of heuristics to move operations in the PDG and then uses a subsequent local scheduling pass to order operations within each block. Fisher[3] argues that incorporating movement of operations with the scheduling phase itself provides better scheduling than dividing the interblock motion and scheduling phases. Based on that criterion alone, DPS has some advantages over Bernestein's method.

Conclusions

It is commonly accepted that to exploit the performance benefits of ILP, global instruction scheduling is required. Several varieties of global instruction scheduling exist, most requiring compensation copies to ensure proper program semantics when operations cross block boundaries during instruction scheduling. Although such global scheduling with compensation copies may be an effective strategy for architectures with large degrees of ILP, another approach seems reasonable for more limited architectures, such as currently available superscalar computers.

This paper outlines DPS, a global instruction scheduling technique that does not require compensation copies. Based on the fact that more than 25 percent of intermediate statements can be moved upward at least one dominator block in the control flow graph without changing program semantics, DPS schedules paths in a function's dominator tree as meta-blocks, making use of an extended local instruction scheduler to schedule dominator paths.

Experimental evidence shows that DPS does indeed produce more efficient schedules than local scheduling for Compaq's Alpha 21164 server system, particularly for floating-point programs that avoid the use of global variables. This work has demonstrated that considerable flexibility in placement of code is possible even when compensation copies are not allowed. Although more research is required to look into possible uses for this flexibility, the global instruction scheduling method described here (DPS) shows promise for ILP architectures.

Acknowledgments

This research was supported in part by an External Research Program grant from Digital Equipment Corporation and by the National Science Foundation under grant CCR-9308348.

References

1. G. Tjaden and M. Flynn, "Detection of Parallel Execution of Independent Instructions," *IEEE Transactions on Computers,* C-19(10) (October 1970): 889–895.

2. A. Nicolau and J. Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures," *IEEE Transactions on Computers,* 33(11) (November 1984): 968–976.

3. J. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers,* C-30(7) (July 1981): 478–490.

4. J. Ellis, *Bulldog: A Compiler for VLIW Architectures* (Cambridge, MA: MIT Press, 1985), Ph.D. thesis, Yale University (1984).

5. D. DeWitt, "A Machine-Independent Approach to the Production of Optimal Horizontal Microcode," Ph.D. thesis, University of Michigan, Ann Arbor, Mich. (1976).

6. D. Landskov, S. Davidson, B. Shriver, and P. Mallett, "Local Microcode Compaction Techniques," *ACM Computing Surveys,* 12(3) (September 1980): 261–294.

7. V. Allan, S. Beaty, B. Su, and P. Sweany, "Building a Retargetable Local Instruction Scheduler," *Software—Practice & Experience,* 28(3) (March 1998): 249–284.

8. E. Coffman, *Computer and Job-Shop Scheduling Theory* (New York: John Wiley & Sons, 1976).

9. D. Padua, D. Kuck, and D. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Transactions on Computers,* C-29(9) (September 1980): 763–776.

10. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools* (Reading, MA: Addison-Wesley, 1986).

11. H. Reif and R. Tarjan, "Symbolic Program Analysis in Almost-Linear Time," *Journal of Computing,* 11 (1) (February 1981): 81–93.

12. P. Sweany, "Interblock Code Motion without Copies," Ph.D. thesis, Computer Science Department, Colorado State University (1992).

13. R. Mueller, M. Duda, P. Sweany, and J. Walicki, "Horizon: A Retargetable Compiler for Horizontal Microarchitectures," *IEEE Transactions on Software Engineering: Special Issue on Microprogramming,* 14(5) (May 1998): 575–583.

14. B. Huber, "Path-Selection Heuristics for Dominator-Path Scheduling," Master's thesis, Department of Computer Science, Michigan Technological University (1995).

15. M. Bourke, P. Sweany, and S. Beaty, "Extending List Scheduling to Consider Execution Frequency," *Proceedings of the 28th Hawaii International Conference on System Sciences* (January 1996).

16. A. Nicolau, "Percolation Scheduling: A Parallel Compilation Technique," Technical Report TR85-678, Department of Computer Science, Cornell University (May 1985).

17. A. Aiken and A. Nicolau, "A Development Environment for Horizontal Microcode," *IEEE Transactions on Software Engineering,* 14(5) (May 1988): 584–594.

18. R. Gupta and M. Soffa, "Region Scheduling: An Approach for Detecting and Redistributing Parallelism," *IEEE Transactions on Software Engineering,* 16(4) (April 1990): 421–431.

19. S. Mahlke, W. Chen, W.-M. Hwu, B. Rao, and M. Schlansker, "Sentinel Scheduling for VLIW and Superscalar Processors," *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems,* Boston, Mass. (October 1992): 238–247.

20. C. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B. Rau, and M. Schlansker, "Profile-Driven Instruction-Level-Parallel Scheduling with Application to Super Blocks," *Proceedings of the 29th International Symposium on Microarchitecture* (MICRO-29), Paris, France (December 1996): 58–67.

21. V. Allan, R. Jones, R. Lee, and S. Allan, "Software Pipelining," *ACM Computing Surveys,* 27(3) (September 1995).

22. B. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," *Proceedings of the 27th International Symposium on Microarchitecture* (MICRO-27), San Jose, Calif. (December 1994): 63–74.

23. S.-M. Moon and K. Ebcioglu, "Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining," *ACM Transactions on Programming Languages and Systems,* 18(6) (November 1997): 853–898.

24. S. Novak and A. Nicolau, "An Efficient Global Resource-Directed Approach to Exploiting Instruction-Level Parallelism," *Proceedings of the 1996 International Conference on Parallel Architectures and Compiler Techniques* (PACT 96), Boston, Mass. (October 1996): 87–96.

25. D. Bernstein and M. Rodeh, "Global Instruction Scheduling for Superscalar Machines," *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation,* Toronto, Canada (June 1991): 241–255.

26. D. Bernstein, D. Cohen, and H. Krawczyk, "Code Duplication: An Assist for Global Instruction Scheduling," *Proceedings of the 24th International Symposium on Microarchitecture* (MICRO-24), Albuquerque, N. Mex. (November 1991): 103–113.

27. D. Bernstein, D. Cohen, Y. Lavon, and V. Rainish, "Performance Evaluation of Instruction Scheduling on the IBM RS/6000," *Proceedings of the 25th International Symposium on Microarchitecture* (MICRO-25), Portland, Oreg. (December 1992): 226–235.

28. J. Ferrante, K. Ottenstein, and J. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems,* 9(3) (July 1987): 319–349.

Biographies

Philip H. Sweany
Associate Professor Phil Sweany has been a member of Michigan Technological University's Computer Science faculty since 1991. He has been investigating compiler techniques for instruction-level parallel (ILP) architectures, co-authoring several papers on instruction scheduling, register assignment, and the interaction between these two optimizations. Phil has been the primary designer and implementer of Rocket, a highly optimizing compiler that is easily retargetable for a wide range of ILP architectures. His research has been significantly assisted by grants from Digital Equipment Corporation and the National Science Foundation. Phil received a B.S. in computer science in 1983 from Washington State University, and M.S. and Ph.D. degrees in computer science from Colorado State University in 1986 and 1992, respectively.

Steven M. Carr
Steve Carr is an assistant professor in the Department of Computer Science at Michigan Technological University. The focus of his research at the university is memory-hierarchy management and optimization of instruction-level parallel architectures. Steve's research has been supported by both the National Science Foundation and Digital Equipment Corporation. He received a B.S. in computer science from Michigan Technological University in 1987 and M.S. and Ph.D. degrees from Rice University in 1990 and 1993, respectively. Steve is a member of ACM and an IEEE Computer Society Affiliate.

Brett L. Huber
Raised in Hope, Michigan, Brett earned B.S. and M.S. degrees in computer science at Michigan Technological University in Michigan's historic Keweenaw Peninsula. He is an engineer in the Software Development Systems group at Texas Instruments, Inc., and is currently developing an optimizing compiler for the TMS320C6x family of VLIW digital signal processors. Brett is a member of the ACM and an IEEE Computer Society Affiliate.

Maximizing Multiprocessor Performance with the SUIF Compiler

Mary W. Hall
Jennifer M. Anderson
Saman P. Amarasinghe
Brian R. Murphy
Shih-Wei Liao
Edouard Bugnion
Monica S. Lam

Parallelizing compilers for multiprocessors face many hurdles. However, SUIF's robust analysis and memory optimization techniques enabled speedups on three fourths of the NAS and SPECfp95 benchmark programs.

The affordability of shared memory multiprocessors offers the potential of supercomputer-class performance to the general public. Typically used in a multiprogramming mode, these machines increase throughput by running several independent applications in parallel. But multiple processors can also work together to speed up single applications. This requires that ordinary sequential programs be rewritten to take advantage of the extra processors.[1–4] Automatic parallelization with a compiler offers a way to do this.

Parallelizing compilers face more difficult challenges from multiprocessors than from vector machines, which were their initial target. Using a vector architecture effectively involves parallelizing repeated arithmetic operations on large data streams—for example, the innermost loops in array-oriented programs. On a multiprocessor, however, this approach typically does not provide sufficient granularity of parallelism: Not enough work is performed in parallel to overcome processor synchronization and communication overhead. To use a multiprocessor effectively, the compiler must exploit coarse-grain parallelism, locating large computations that can execute independently in parallel.

Locating parallelism is just the first step in producing efficient multiprocessor code. Achieving high performance also requires effective use of the memory hierarchy, and multiprocessor systems have more complex memory hierarchies than typical vector machines: They contain not only shared memory but also multiple levels of cache memory.

These added challenges often limited the effectiveness of early parallelizing compilers for multiprocessors, so programmers developed their applications from scratch, without assistance from tools. But explicitly managing an application's parallelism and memory use requires a great deal of programming knowledge, and the work is tedious and error-prone. Moreover, the resulting programs are optimized for only a specific machine. Thus, the effort required to develop efficient parallel programs restricts the user base for multiprocessors.

This article describes automatic parallelization techniques in the SUIF (Stanford University Intermediate

Format) compiler that result in good multiprocessor performance for array-based numerical programs. We provide SUIF performance measurements for the complete NAS and SPECfp95 benchmark suites. Overall, the results for these scientific programs are promising. The compiler yields speedups on three fourths of the programs and has obtained the highest ever performance on the SPECfp95 benchmark, indicating that the compiler can also achieve efficient absolute performance.

Finding Coarse-grain Parallelism

Multiprocessors work best when the individual processors have large units of independent computation, but it is not easy to find such coarse-grain parallelism. First the compiler must find available parallelism across procedure boundaries. Furthermore, the original computations may not be parallelizable as given and may first require some transformations. For example, experience in parallelizing by hand suggests that we must often replace global arrays with private versions on different processors. In other cases, the computation may need to be restructured—for example, we may have to replace a sequential accumulation with a parallel reduction operation.

It takes a large suite of robust analysis techniques to successfully locate coarse-grain parallelism. General and uniform frameworks helped us manage the complexity involved in building such a system into SUIF. We automated the analysis to privatize arrays and to recognize reductions to both scalar and array variables. Our compiler's analysis techniques all operate seamlessly across procedure boundaries.

Scalar Analyses

An initial phase analyzes scalar variables in the programs. It uses techniques such as data dependence analysis, scalar privatization analysis, and reduction recognition to detect parallelism among operations with scalar variables. It also derives symbolic information on these scalar variables that is useful in the array analysis phase. Such information includes constant propagation, induction variable recognition and elimination, recognition of loop-invariant computations, and symbolic relation propagation.[5,6]

Array Analyses

An array analysis phase uses a unified mathematical framework based on linear algebra and integer linear programming.[3] The analysis applies the basic data dependence test to determine if accesses to an array can refer to the same location. To support array privatization, it also finds array dataflow information that determines whether array elements used in an iteration refer to the values produced in a previous iteration. Moreover, it recognizes commutative operations on sections of an array and transforms them into parallel reductions. The reduction analysis is powerful enough to recognize commutative updates of even indirectly accessed array locations, allowing parallelization of sparse computations.

All these analyses are formulated in terms of integer programming problems on systems of linear inequalities that represent the data accessed. These inequalities are derived from loop bounds and array access functions. Implementing optimizations to speed up common cases reduces the compilation time.

Interprocedural Analysis Framework

All the analyses are implemented using a uniform interprocedural analysis framework, which helps manage the software engineering complexity. The framework uses interprocedural dataflow analysis,[4] which is more efficient than the more common technique of inline substitution.[1] Inline substitution replaces each procedure call with a copy of the called procedure, then analyzes the expanded code in the usual intraprocedural manner. Inline substitution is not practical for large programs, because it can make the program too large to analyze.

Our technique analyzes only a single copy of each procedure, capturing its side effects in a function. This function is then applied at each call site to produce precise results. When different calling contexts make it necessary, the algorithm selectively clones a procedure so that code can be analyzed and possibly parallelized under different calling contexts (as when different constant values are passed to the same formal parameter). In this way the full advantages of inlining are achieved without expanding the code indiscriminately.

In Figure 1 the boxes represent procedure bodies, and the lines connecting them represent procedure calls. The main computation is a series of four loops to compute three-dimensional fast Fourier transforms. Using interprocedural scalar and array analyses, the SUIF compiler determines that these loops are parallelizable. Each loop contains more than 500 lines of code spanning up to nine procedures with up to 42 procedure calls. If this program had been fully inlined, the loops presented to the compiler for analysis would have each contained more than 86,000 lines of code.

Memory Optimization

Numerical applications on high-performance microprocessors are often memory bound. Even with one or more levels of cache to bridge the gap between processor and memory speeds, a processor may still waste half its time stalled on memory accesses because it frequently references an item not in the cache (a cache miss). This

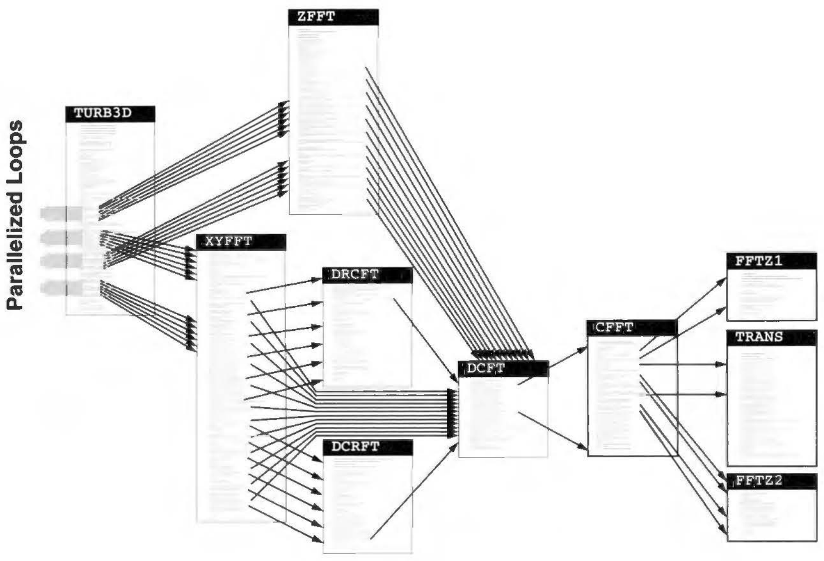


Figure 1
The compiler discovers parallelism through interprocedural array analysis. Each of the four parallelized loops at left consists of more than 500 lines of code spanning up to nine procedures (boxes) with up to 42 procedure calls (lines).

memory bottleneck is further exacerbated on multiprocessors by their greater need for memory traffic, resulting in more contention on the memory bus.

An effective compiler must address four issues that affect cache behavior:

- Communication: Processors in a multiprocessor system communicate through accesses to the same memory location. Coherent caches typically keep the data consistent by causing accesses to data written by another processor to miss in the cache. Such misses are called *true sharing misses.*
- Limited capacity: Numeric applications tend to have large working sets, which typically exceed cache capacity. These applications often stream through large amounts of data before reusing any of it, resulting in poor temporal locality and numerous capacity misses.
- Limited associativity: Caches typically have a small *set associativity;* that is, each memory location can map to only one or just a few locations in the cache. Conflict misses—when an item is discarded and later retrieved—can occur even when the application's working set is smaller than the cache, if the data are mapped to the same cache locations.
- Large line size: Data in a cache are transferred in fixed-size units called cache lines. Applications that do not use all the data in a cache line incur more misses and are said to have poor spatial locality. On a multiprocessor, large cache lines can also lead to cache misses when different processors use different parts of the same cache line. Such misses are called *false sharing misses.*

The compiler tries to eliminate as many cache misses as possible, then minimize the impact of any that remain by

- ensuring that processors reuse the same data as many times as possible and
- making the data accessed by each processor contiguous in the shared address space.

Techniques for addressing each of these subproblems are discussed below. Finally, to tolerate the latency of remaining cache misses, the compiler uses *compiler-inserted prefetching* to move data into the cache before it is needed.

Improving Processor Data Reuse

The compiler reorganizes the computation so that each processor reuses data to the greatest possible extent.[7-9] This reduces the working set on each processor, thereby minimizing capacity misses. It also reduces interprocessor communication and thus minimizes true sharing misses. To achieve optimal reuse, the compiler uses *affine partitioning.* This technique analyzes reference patterns in the program to derive an affine mapping (linear transformation plus an offset) of the computation of the data to the processors. The affine mappings are chosen to maximize a processor's reuse of data while maintaining sufficient parallelism to keep all processors busy. The compiler also uses loop blocking to reorder the computation executed on a single processor so that data is reused in the cache.

Making Processor Data Contiguous

The compiler tries to arrange the data to make a processor's accesses contiguous in the shared address space. This improves spatial locality while reducing conflict misses and false sharing. SUIF can manage data placement within a single array and across multiple arrays. The data-to-processor mappings computed by the affine partitioning analysis are used to determine the data being accessed by each processor.

Figure 2 shows how the compiler's use of data permutation and data strip-mining[10] can make contiguous the data within a single array that is accessed by one processor. Data permutation interchanges the dimensions of the array—for example, transposing a two-dimensional array. Data strip-mining changes an array's dimensionality so that all data accessed by the same processor are in the same plane of the array.

To make data across multiple arrays accessed by the same processor contiguous, we use a technique called *compiler-directed page coloring*.[11] The compiler uses its knowledge of the access patterns to direct the operating system's page allocation policy to make each processor's data contiguous in the physical address space. The operating system uses these hints to determine the virtual-to-physical page mapping at page allocation time.

Experimental Results

We conducted a series of performance evaluations to demonstrate the impact of SUIF's analyses and optimizations. We obtained measurements on a Digital AlphaServer 8400 with eight 21164 processors, each with two levels of on-chip cache and a 4-Mbyte external cache. Because speedups are harder to obtain on machines with fast processors, our use of a state-of-the-art machine makes the results more meaningful and applicable to future systems.

We used two complete standard benchmark suites to evaluate our compiler. We present results for the 10

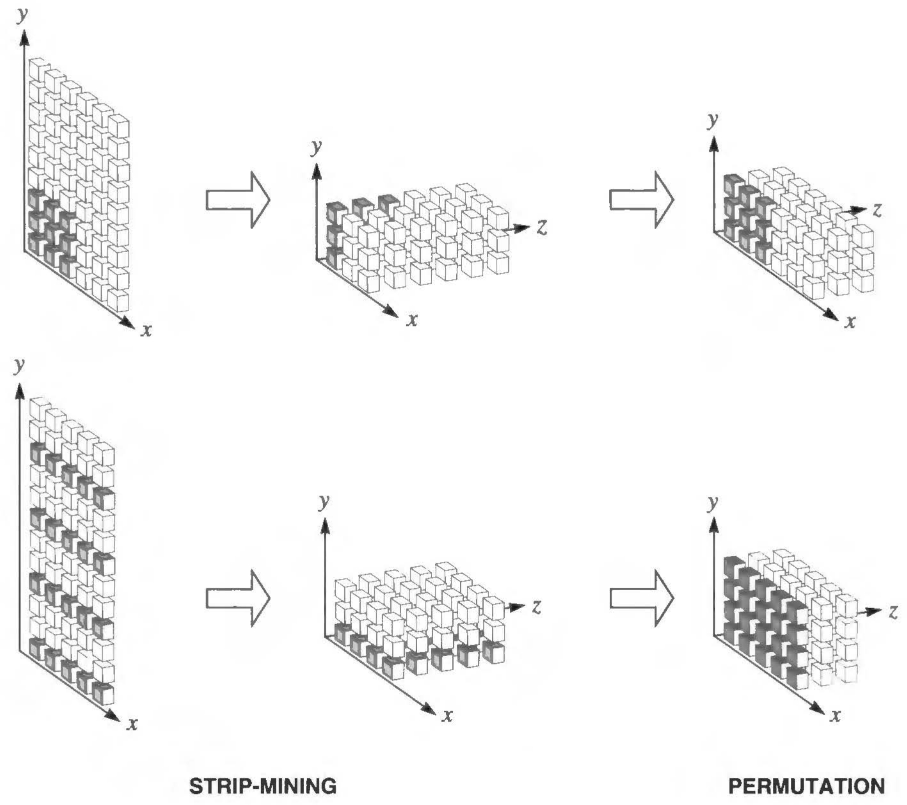


Figure 2
Data transformations can make the data accessed by each processor contiguous in the shared address space. In the two examples above, the original arrays are two-dimensional; the axes are identified to show that elements along the first axis are contiguous. First the affine partitioning analysis determines which data elements are accessed by the same processor (the shaded elements are accessed by the first processor.) Second, data strip-mining turns the 2D array into a 3D array, with the shaded elements in the same plane. Finally, applying data permutation rotates the array, making data accessed by each processor contiguous.

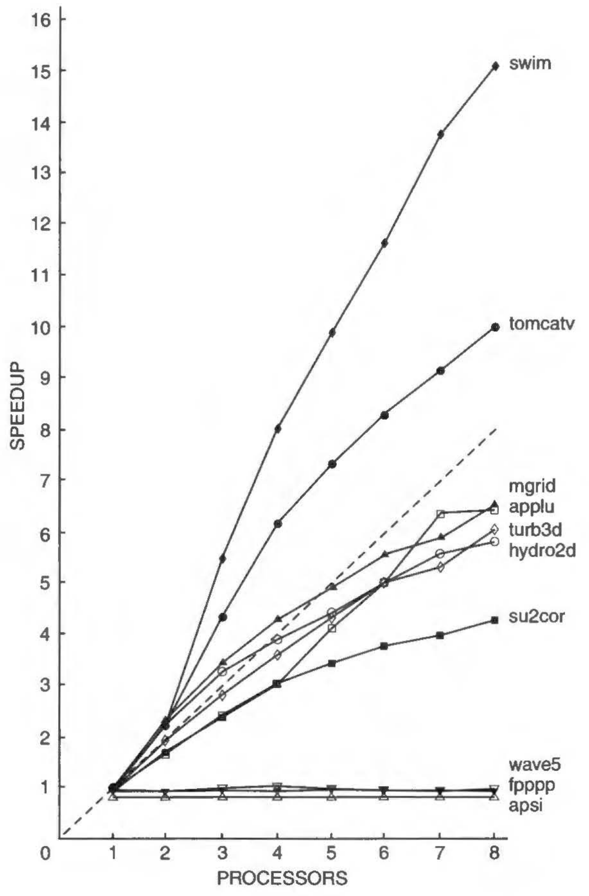
programs in the SPECfp95 benchmark suite, which is commonly used for benchmarking uniprocessors. We also used the eight official benchmark programs from the NAS parallel-system benchmark suite, except for embar; here we used a slightly modified version from Applied Parallel Research.

Figure 3 shows the SPECfp95 and NAS speedups, measured on up to eight processors on a 300-MHz AlphaServer. We calculated the speedups over the best sequential execution time from either officially reported results or our own measurements. Note that mgrid and applu appear in both benchmark suites (the program source and data set sizes differ slightly).

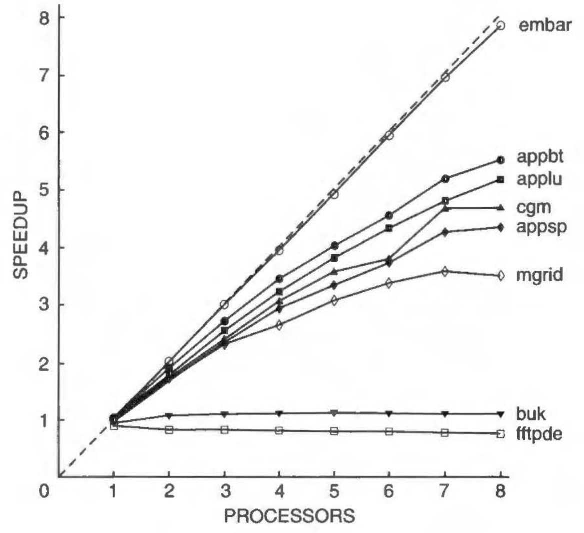
To measure the effects of the different compiler techniques, we broke down the performance obtained on eight processors into three components. In Figure 4, baseline shows the speedup obtained with parallelization using only intraprocedural data dependence analysis, scalar privatization, and scalar reduction transformations. Coarse grain includes the baseline techniques as well as techniques for locating coarse-grain parallel loops—for example, array privatization and reduction transformations, and full interprocedural analysis of both scalar and array variables. Memory includes the coarse-grain techniques as well as the multiprocessor memory optimizations we described earlier.

Figure 3 shows that of the 18 programs, 13 show good parallel speedup and can thus take advantage of additional processors. SUIF's coarse-grain techniques and memory optimizations significantly affect the performance of half the programs. The swim and tomcatv programs show superlinear speedups because the compiler eliminates almost all cache misses and their 14 Mbyte working sets fit into the multiprocessor's aggregate cache.

For most of the programs that did not speed up, the compiler found much of their computation to be parallelizable, but the granularity is too fine to yield good multiprocessor performance on machines with fast processors. Only two applications, fpppp and buk, have



(a) SPECfp95



(b) NAS Parallel Benchmarks

Figure 3
SUIF compiler speedups over the best sequential time achieved on the (a) SPECfp95 and (b) NAS parallel benchmarks.

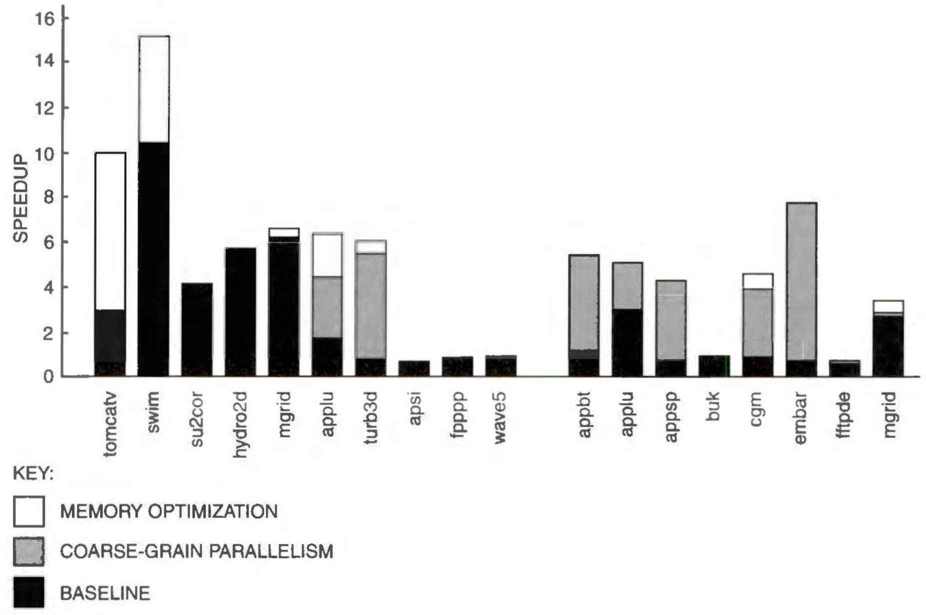


Figure 4
The speedup achieved on eight processors is broken down into three components to show how SUIF's memory optimization and discovery of coarse-grain parallelism affected performance.

no statically analyzable loop-level parallelism, so they are not amenable to our techniques.

Table 1 shows the times and SPEC ratios obtained on an eight-processor, 440-MHz Digital AlphaServer 8400, testifying to our compiler's high absolute performance. The SPEC ratios compare machine performance with that of a reference machine. (These are not official SPEC ratings, which among other things require that the software be generally available. The ratios we obtained are nevertheless valid in assessing our compiler's performance.) The geometric mean of the SPEC ratios improves over the uniprocessor execution by a factor of 3 with four processors and by a factor of 4.3 with eight processors. Our eight-processor ratio of 63.9 represents a 50 percent improvement over the highest number reported to date.[12]

Table 1
Absolute Performance for the SPECfp95 Benchmarks Measured on a 440-MHz Digital AlphaServer Using One Processor, Four Processors, and Eight Processors

	Execution Time (secs)			SPEC Ratio		
Benchmark	**1P**	**4P**	**8P**	**1P**	**4P**	**8P**
tomcatv	219.1	30.3	18.5	16.9	122.1	200.0
swim	297.9	33.5	17.2	28.9	256.7	500.0
su2cor	155.0	44.9	31.0	9.0	31.2	45.2
hydro2d	249.4	61.1	40.7	9.6	39.3	59.0
mgrid	185.3	42.0	27.0	13.5	59.5	92.6
applu	296.1	85.5	39.5	7.4	25.7	55.7
turb3d	267.7	73.6	43.5	15.3	55.7	94.3
apsi	137.5	141.2	143.2	15.3	14.9	14.7
fpppp	331.6	331.6	331.6	29.0	29.0	29.0
wave5	151.8	141.9	147.4	19.8	21.1	20.4
Geometric Mean				15.0	44.4	63.9

Acknowledgments

This research was supported in part by the Air Force Materiel Command and ARPA contracts F30602-95-C-0098, DABT63-95-C-0118, and DABT63-94-C-0054; a Digital Equipment Corporation grant; an NSF Young Investigator Award; an NSF CISE postdoctoral fellowship; and fellowships from AT&T Bell Laboratories, DEC Western Research Laboratory, Intel Corp., and the National Science Foundation.

References

1. J.M. Anderson, S.P. Amarasinghe, and M.S. Lam, "Data and Computation Transformations for Multiprocessors," *Proc. Fifth ACM SIGPlan Symp. Principles and Practice of Parallel Programming,* ACM Press, New York, 1995, pp. 166–178.
2. J. M. Anderson and M.S. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," *Proc. SIGPlan '93 Conf. Programming Language Design and Implementation,* ACM Press, New York, 1993, pp. 112–125.
3. P. Banerjee et al., "The Paradigm Compiler for Distributed-Memory Multicomputers," *Computer,* Oct. 1995, pp. 37–47.
4. W. Blume et al., "Effective Automatic Parallelization with Polaris," *Int'l. J. Parallel Programming,* May 1995.
5. E. Bugnion et al., "Compiler-Directed Page Coloring for Multiprocessors," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, New York, 1996, pp. 244–257.
6. K. Cooper et al., "The ParaScope Parallel Programming Environment," *Proc. IEEE,* Feb. 1993, pp. 244–263.
7. Standard Performance Evaluation Corp., "Digital Equipment Corporation AlphaServer 8400 5/440 SPEC CFP95 Results," *SPEC Newsletter,* Oct. 1996.
8. M. Haghighat and C. Polychronopolous, "Symbolic Analysis for Parallelizing Compilers," *ACM Trans. Programming Languages and Systems,* July 1996, pp. 477–518.
9. M.W. Hall et al., "Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler," *Proc. Supercomputing '95,* IEEE CS Press, Los Alamitos, Calif., 1995 (CD-ROM only).
10. P. Havlak, *Interprocedural Symbolic Analysis,* PhD thesis, Dept. of Computer Science, Rice Univ., May 1994.
11. F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical Interprocedural Parallelization: An Overview of the PIPS Project," *Proc. 1991 ACM Int'l Conf. Supercomputing,* ACM Press, New York, 1991, pp. 244–251.
12. K. Kennedy and U. Kremer, "Automatic Data Layout for High Performance Fortran," *Proc. Supercomputing '95,* IEEE CS Press, Los Alamitos, Calif., 1995 (CD-ROM only).

Editors' Note: With the following section, the authors provide an update on the status of the SUIF compiler since the publication of their paper in Computer *in December 1996.*

Addendum: The Status and Future of SUIF

Public Availability of SUIF-parallelized Benchmarks

The SUIF-parallelized versions of the SPECfp95 benchmarks used for the experiments described in this paper have been released to the SPEC committee and are available to any license holders of SPEC (see http://www.specbench.org/osg/cpu95/par-research). This benchmark distribution contains the SUIF output (C and FORTRAN code), along with the source code for the accompanying run-time libraries. We expect these benchmarks will be useful for two purposes: (1) for technology transfer, providing insight into how the compiler transforms the applications to yield the reported results; and (2) for further experimentation, such as in architecture-simulation studies.

The SUIF compiler system itself is available from the SUIF web site at http://www-suif.stanford.edu. This system includes only the standard parallelization analyses that were used to obtain our baseline results.

New Parallelization Analyses in SUIF

Overall, the results of automatic parallelization reported in this paper are impressive; however, a few applications either do not speed up at all or achieve limited speedup at best. The question arises as to whether SUIF is exploiting all the available parallelism in these applications. Recently, an experiment to answer this question was performed in which loops left unparallelized by SUIF were instrumented with run-time tests to determine whether opportunities for increasing the effectiveness of automatic parallelization remained in these programs.[1] Run-time testing determined that eight of the programs from the NAS and SPEC95fp benchmarks had additional parallel loops, for a total of 69 additional parallelizable loops, which is less than 5% of the total number of loops in these programs. Of these 69 loops, the remaining parallelism had a significant effect on coverage (the percentage of the program that is parallelizable) or granularity (the size of the parallel regions) in only four of the programs: apsi, su2cor, wave5, and fftpde.

We found that almost all the significant loops in these four programs could potentially be parallelized using a new approach that associates predicates with array data-flow values.[2] Instead of producing conservative results that hold for all control-flow paths and all possible program inputs, predicated array data-flow analysis can derive optimistic results guarded by predicates. Predicated array data-flow analysis can lead to more effective automatic parallelization in three ways: (1) It improves compile-time analysis by ruling out infeasible control-flow paths. (2) It provides a framework for the compiler to introduce predicates that, if proven true, would guarantee safety for desirable data-flow values. (3) It enables the compiler to derive low-cost run-time parallelization tests based on the predicates associated with desirable data-flow values.

SUIF and Compaq's GEM Compiler

The GEM compiler system is the technology Compaq has been using to build compiler products for a variety of languages and hardware/software platforms.[3] Within Compaq, work has been done to connect SUIF with the GEM compiler. SUIF's intermediate representation was converted into GEM's intermediate representation, so that SUIF code can be passed directly to GEM's optimizing back end. This eliminates the loss of information suffered when SUIF code is translated to C/FORTRAN source before it is passed to GEM. It also enables us to generate more efficient code for Alpha-microprocessor systems.

SUIF and the National Compiler Infrastructure

The SUIF compiler system was recently chosen to be part of the National Compiler Infrastructure (NCI) project funded by the Defense Advanced Research Projects Agency (DARPA) and the National Science Foundation (NSF). The goal of the project is to develop a common compiler platform for researchers and to facilitate technology transfer to industry. The SUIF component of the NCI project is the result of the collaboration among researchers in five universities (Harvard University, Massachusetts Institute of Technology, Rice University, Stanford University, University of California at Santa Barbara) and one industrial partner, Portland Group Inc. Compaq is a corporate sponsor of the project and is providing the FORTRAN front end.

A revised version of the SUIF infrastructure (SUIF 2.0) is being released as part of the SUIF NCI project (a preliminary version of SUIF 2.0 is available at the SUIF web site). The completed system will be enhanced to support parallelization, interprocedural analysis, memory hierarchy optimizations, objected-oriented programming, scalar optimizations, and machine-dependent optimizations. An overview of the SUIF NCI system is shown in Figure A1. See www-suif.stanford.edu/suif/NCI/suif.html for more information about SUIF and the NCI project, including a complete list of optimizations and a schedule.

References

1. B. So, S. Moon, and M. Hall, "Measuring the Effectiveness of Automatic Parallelization in SUIF," *Proceedings of the International Conference on Supercomputing '98,* July 1998.
2. S. Moon, M. Hall, and B. Murphy, "Predicated Array Data-Flow Analysis for Run-Time Parallelization," *Proceedings of the International Conference on Supercomputing '98,* July 1998.
3. D. Blickstein et al., "The GEM Optimizing Compiler System," *Digital Technical Journal,* vol. 4, no. 4 (Special Issue, 1992): 121–136.

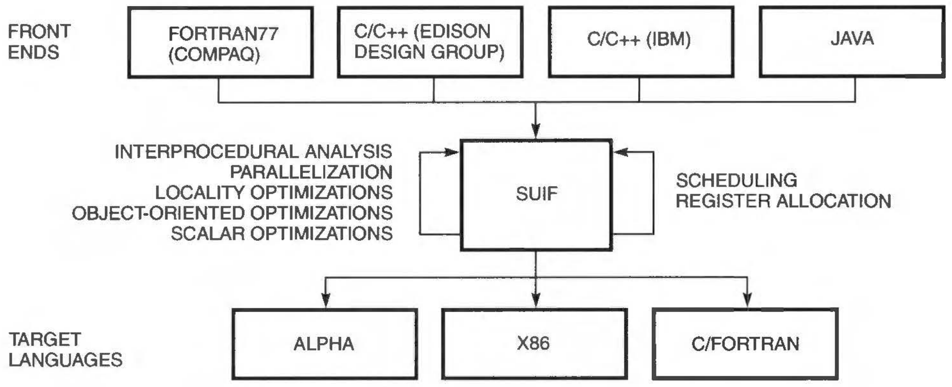


Figure A1
The SUIF Compiler Infrastructure

Biographies

Mary W. Hall
Mary Hall is jointly a research assistant professor and project leader at the University of Southern California, Department of Computer Science and at USC's Information Sciences Institute, where she has been since 1996. Her research interests focus on compiler support for high-performance computing, particularly interprocedural analysis and automatic parallelization. She graduated magna cum laude with a B.A. in computer science and mathematical sciences in 1985 and received an M.S. and a Ph.D. in computer science in 1989 and 1991, respectively, all from Rice University. Prior to joining USC/ISI, she was a visiting assistant professor and senior research fellow in the Department of Computer Science at Caltech. In earlier positions, she was a research scientist at Stanford University, working with the SUIF Compiler group, and in the Center for Research on Parallel Computation at Rice University.

Jennifer M. Anderson
Jennifer Anderson is a research staff member at Compaq's Western Research Laboratory where she has worked on the Digital Continuous Profiling Infrastructure (DCPI) project. Her research interests include compiler algorithms, programming languages and environments, profiling systems, and parallel and distributed systems software. She earned a B.S. in information and computer science from the University of California at Irvine and received M.S. and Ph. D. degrees in computer science from Stanford University.

Saman P. Amarasinghe
Saman Amarasinghe is an assistant professor of computer science and engineering at the Massachusetts Institute of Technology and a member of the Laboratory for Computer Science. His research interests include compilers and computer architecture. He received a B.S. in electrical engineering and computer science from Cornell University and M.S. and Ph.D. degrees in electrical engineering from Stanford University.

Brian R. Murphy
A doctoral candidate in computer science at Stanford University, Brian Murphy is currently working on advanced program analysis under SUIF as part of the National Compiler Infrastructure Project. He received a B.S. in computer science and engineering and an M.S. in electrical engineering and computer science from the Massachusetts Institute of Technology. His master's thesis work on program analysis was carried out with the Functional Languages group at the IBM Almaden Research Center. Brian was elected to the Tau Beta Pi and Eta Kappa Nu honor societies.

Shih-Wei Liao
Shih-Wei Liao is a doctoral candidate at the Stanford University Computer Systems Laboratory. His research interests include compiler algorithms and design, programming environments, and computer architectures. He received a B.S. in computer science from National Taiwan University in 1991 and an M.S. in electrical engineering from Stanford University in 1994.

Edouard Bugnion
Ed Bugnion holds a Diplom in engineering from the Swiss Federal Institute of Technology (ETH), Zurich (1994) and an M.S. from Stanford University (1996), where he is a doctoral candidate in computer science. His research interests include operating systems, computer architecture, and machine simulation. From 1996 to 1997, Ed was also a research consultant to Compaq's Western Research Laboratory. He is the recipient of a National Science Foundation Graduate Research Fellowship.

Monica S. Lam
Monica Lam is an associate professor in the Computer Science Department at Stanford University. She leads the SUIF project, which is aimed at developing a common infrastructure to support research in compilers for advanced languages and architectures. Her research interests are compilers and computer architecture. Monica earned a B.S. from the University of British Columbia in 1980 and a Ph.D. in computer science from Carnegie Mellon University in 1987. She received the National Science Foundation Young Investigator award in 1992.

Ronald F. Brender
Jeffrey E. Nelson
Mark E. Arsenault

Debugging Optimized Code: Concepts and Implementation on DIGITAL Alpha Systems

Effective user debugging of optimized code has been a topic of theoretical and practical interest in the software development community for almost two decades, yet today the state of the art is still highly uneven. We present a brief survey of the literature and current practice that leads to the identification of three aspects of debugging optimized code that seem to be critical as well as tractable without extraordinary efforts. These aspects are (1) split lifetime support for variables whose allocation varies within a program combined with definition point reporting for currency determination, (2) stepping and setting breakpoints based on a semantic event characterization of program behavior, and (3) treatment of inlined routine calls in a manner that makes inlining largely transparent. We describe the realization of these capabilities as part of Compaq's GEM back-end compiler technology and the debugging component of the OpenVMS Alpha operating system.

Introduction

In software development, it is common practice to debug a program that has been compiled with little or no optimization applied. The generated code closely corresponds to the source and is readily described by a simple and straightforward debugging symbol table. A debugger can interpret and control execution of the code in a fashion close to the user's source-level view of the program.

Sometimes, however, developers find it necessary or desirable to debug an optimized version of the program. For instance, a bug—whether a compiler bug or incorrect source code—may only reveal itself when optimization is applied. In other cases, the resource constraints may not allow the unoptimized form to be used because the code is too big and/or too slow. Or, the developer may need to start analysis using the remains, such as a core file, of the failed program, whether or not this code has been optimized. Whatever the reason, debugging optimized code is harder than debugging unoptimized code—much harder—because optimization can greatly complicate the relationship between the source program and the generated code.

Zellweger[1] introduced the terms *expected behavior* and *truthful behavior* when referring to debugging optimized code. A debugger provides expected behavior if it provides the behavior a user would experience when debugging an unoptimized version of a program. Since achieving that behavior is often not possible, a secondary goal is to provide at least truthful behavior, that is, to never lie to or mislead a user. In our experience, even truthful behavior can be challenging to achieve, but it can be closely approached.

This paper describes three improvements made to Compaq's GEM back-end compiler system and to OpenVMS DEBUG, the debugging component of the OpenVMS Alpha operating system. These improvements address

1. Split lifetime variables and currency determination
2. Semantic events
3. Inlining

Before presenting the details of this work, we discuss the alternative approaches to debugging optimized code that we considered, the state of the art, and the operating strategies we adopted.

Alternative Approaches

Various approaches have been explored to improve the ability to debug optimized code. They include the following:

- Enhance debugger analysis
- Limit optimization
- Limit debugging to preplanned locations
- Dynamically deoptimize as needed
- Exploit an associated program database

We touch on these approaches in turn.

In probably the oldest theoretical analysis that supports debugging optimized code, Hennessy[2] studies whether the value displayed for a variable is current, that is, the expected value for that variable at a given point in the program. The value displayed might not be current because, for example, assignment of a later value has been moved forward or the relevant assignment has been delayed or omitted. Hennessy postulates that a flow graph description of a program is communicated to the debugger, which then solves certain flow analysis equations in response to debug commands to determine currency as needed. Copperman[3] takes a similar though much more general approach. Conversely, commercial implementations have favored more complete preprocessing of information in the compiler to enable simpler debugger mechanisms.[4–6]

If optimization is the "problem," then one approach to solving the problem is to limit optimization to only those kinds that are actually supported in an available debugger. Zurawski[7] develops the notion of a *recovery function* that matches each kind of optimization. As an optimization is applied during compilation, the compensating recovery function is also created and made available for later use by a debugger. If such a recovery function cannot be created, then the optimization is omitted. Unfortunately, code-motion-related optimizations generally lack recovery functions and so must be foregone. Taking this approach to the extreme converges with traditional practice, which is simply to disable all optimization and debug a completely unoptimized program.

If full debugger functionality need only be provided at some locations, then some debugger capabilities can be provided more easily. Zurawski[7] also employed this idea to make it easier to construct appropriate recovery functions. This approach builds on a language-dependent concept of *inspection points,* which generally must include all call sites and may correspond to most statement boundaries. His experience suggests, however, that even limiting inspection points to statement boundaries severely limits almost all kinds of optimization.

Hölzle et al.[8] describe techniques to dynamically deoptimize part of a program (replace optimized code with its unoptimized equivalent) during debugging to enable a debugger to perform requested actions. They make the technique more tractable, in part by delaying asynchronous events to well-defined *interruption points,* generally backward branches and calls. Optimization between interruption points is unrestricted. However, even this choice of interruption points severely limits most code motion and many other global optimizations.

Pollock and others[9,10] use a different kind of deoptimization, which might be called preplanned, incremental deoptimization. During a debugging session, any debugging requests that cannot be honored because of optimization effects are remembered so that a subsequent compilation can create an executable that can honor these requests. This scheme is supported by an incremental optimizer that uses a program database to provide rapid and smooth forward information flow to subsequent debugging sessions.

Feiler[11] uses a program database to achieve the benefits of interactive debugging while applying as much static compilation technology as possible. He describes techniques for maintaining consistency between the primary tree-based representation and a derivative compiled form of the program in the face of both debugging actions and program modifications on-the-fly. While he appears to demonstrate that more is possible than might be expected, substantial limitations still exist on debugging capability, optimization, or both.

A comprehensive introduction and overview to these and other approaches can be found in Copperman[3] and Adl-Tabatabi.[12] In addition, "An Annotated Bibliography on Debugging Optimized Code" is available separately on the *Digital Technical Journal* web site at http://www.digital.com/info/DTJ. This bibliography cites and summarizes the entire literature on debugging optimized code as best we know it.

State of the Art

When we began our work in early 1994, we assessed the level of support for debugging optimized code that was available with competitive compilers. Because we have not updated this assessment, it is not appropriate for us to report the results here in detail. We do however summarize the methodology used and the main results, which we believe remain generally valid.

We created a series of example programs that provide opportunities for optimization of a particular kind or of related kinds, and which could lead a traditional debugger to deviate from expected behavior. We compiled and executed these programs under the control of each system's debugger and recorded how the system handled the various kinds of optimization. The range of observed behaviors was diverse.

At one extreme were compilers that automatically disable all optimization if a debugging symbol table is requested (or, equivalently for our purposes, give an error if both optimization and a debugging symbol table are requested). For these compilers, the whole exercise becomes moot; that is, attempting to debug optimized code is not allowed.

Some compiler/debugger combinations appeared to usefully support some of our test cases, although none handled all of them correctly. In particular, none seemed able to show a traceback of subroutine calls that compensated for inlining of routine calls and all seemed to produce a lot of jitter when stepping by line on systems where code is highly scheduled.

The worst example that we found allowed compilation using optimization but produced a debugging symbol table that did not reflect the results of that optimization. For example, local variables were described as allocated on the stack even though the generated code clearly used registers for these variables and never accessed any stack locations. At debug time, a request to examine such a variable resulted in the display of the irrelevant and never-accessed stack locations.

The bottom line from this analysis was very clear: the state of the art for support of debugging optimized code was generally quite poor. DIGITAL's debuggers, including OpenVMS DEBUG, were not unusual in this regard. The analysis did indicate some good examples, though. Both the CONVEX CXdb[4,5] and the HP 9000 DOC[6] systems provide many valuable capabilities.

Biases and Goals

Early in our work, we adopted the following strategies:

- Do not limit or compromise optimization in any way.
- Stay within the framework of the traditional edit-compile-link-debug cycle.
- Keep the burden of analysis within the compiler.

The prime directive for Compaq's GEM-based compilers is to achieve the highest possible performance from the Alpha architecture and chip technology. Any improvements in debugging such optimized code should be useful in the face of the best that a compiler has to offer. Conversely, if a programmer has the luxury of preparing a less optimized version for debugging purposes, there is little or no reason for that version to be anything other than completely unoptimized. There seems to be no particular benefit to creating a special intermediate level of combined debugger/optimization support.

Pragmatically, we did not have the time or staffing to develop a new optimization framework, for example, based on some kind of program database. Nor were we interested in intruding into those parts of the GEM compiler that performed optimization to create more complicated options and variations, which might be needed for dynamic deoptimization or recovery function creation.

Finally, it seemed sensible to perform most analysis activities within the compiler, where the most complete information about the program is already available. It is conceivable that passing additional information from the compiler to the debugger using the object file debugging symbol table might eventually tip the balance toward performing more analysis in the debugger proper. The available size data (presented later in this paper in Table 3) do not indicate this.

We identified three areas in which we felt enhanced capabilities would significantly improve support for debugging optimized code. These areas are

1. The handling of split lifetime variables and currency determination
2. The process of stepping though the program
3. The handling of procedure inlining

In the following sections we present the capabilities we developed in each of these areas together with insight into the implementation techniques employed.

First, we review the GEM and OpenVMS DEBUG framework in which we worked. The next three sections address the new capabilities in turn. The last major section explores the resource costs (compile-time size and performance, and object and image sizes) needed to realize these capabilities.

Starting Framework

Compaq's GEM compiler system and the OpenVMS DEBUG component of the OpenVMS operating system provide the framework for our work. A brief description of each follows.

GEM

The GEM compiler system[13] is the technology Compaq is using to build state-of-the-art compiler products for a variety of languages and hardware and software platforms. The GEM system supports a range of languages (C, C++, FORTRAN including HPF, Pascal, Ada, COBOL, BLISS, and others) and has been successfully retargeted and rehosted for the Alpha, MIPS, and Intel IA-32 architectures and for the

OpenVMS, DIGITAL UNIX, Windows NT, and Windows 95 operating systems.

The major components of a GEM compiler are the front end, the optimizer, the code generator, the final code stream optimizer, and the compiler shell.

- The front end performs lexical analysis and parsing of the source program. The primary outputs are intermediate language (IL) graphs and symbol tables. Front ends for all source languages translate to the same common representation.
- The optimizer transforms the IL generated by the front end into a semantically equivalent form that will execute faster on the target machine. A significant technical achievement is that a single optimizer is used for all languages and target platforms.
- The code generator translates the IL into a list of code cells, each of which represents one machine instruction for the target hardware. Virtually all the target machine instruction-specific code is encapsulated in the code generator.
- The final phase performs pattern-based peephole optimizations followed by instruction scheduling.
- The shell is a portable interface to the external environment in which the compiler is used. It provides common compiler functions such as listing generators, object file emitters, and command line processors in a form that allows the other components to remain independent of the operating system.

The bulk of the GEM implementation work described in this paper occurs at the boundary between the final phase and the object file output portion of the shell. A new debugging optimized code analysis phase examines the generated code stream representation of the program, together with the compiler symbol table, to extract the information necessary to pass on to a debugger through the debugging symbol table. Most of the implementation is readily adapted to different target architectures by means of the same instruction property tables that are used in the code generator and final optimizer.

OpenVMS DEBUG

The OpenVMS Alpha debugger, originally developed for the OpenVMS VAX system,[14] is a full-function, source-level, symbolic debugger. It supports symbolic debugging of programs written in BLISS, MACRO-32, MACRO-64, FORTRAN, Ada, C, C++, Pascal, PL/1, BASIC, and COBOL. The debugger allows the user to control the execution and to examine the state of a program. Users can

- Set breakpoints to stop at certain points in the program
- Step through the execution of the program a line at a time
- Display the source-level view of the program's execution using either a graphical user interface or a character-based user interface
- Examine user variables and hardware registers
- Display a stack traceback showing the current call stack
- Set watch points
- Perform many other functions[15]

Split Lifetime Variables and Currency Determination

Displaying (printing) the value of a program variable is one of the most basic services that a debugger can provide. For unoptimized code and traditional debuggers, the mechanisms for doing this are generally based on several assumptions.

1. A variable has a single allocation that remains fixed throughout its lifetime. For a local or a stack-allocated variable that means throughout the lifetime of the scope in which the variable is declared.
2. Definitions and uses of the values of user variables occur in the same order in the generated code as they do in the original program source.
3. The set of instructions that belong to a given scope (which may be a routine body) can be described by a single contiguous range of addresses.

The first and second assumptions are of interest in this discussion because many GEM optimizations make them inappropriate. Split lifetime optimization (discussed later in this section) leads to violation of the first assumption. Code motion optimization leads to violation of the second assumption and thereby creates the so-called currency problem. We treat both of these problems together, and we refer to them collectively as *split lifetime support*. Statement and instruction scheduling optimization leads to violation of the third assumption. This topic is addressed later, in the section Inlining.

Split Lifetime Variable Definition

A variable is said to have split lifetimes if the set of fetches and stores of the variable can be partitioned such that none of the values stored in one subset are ever fetched in another subset. When such a partition exists, the variable can be "split" into several independent "child" variables, each corresponding to a partition. As independent variables, the child variables can be allocated independently. The effect is that the original variable can be thought to reside in different locations at different points in time—sometimes in a register, sometimes in memory, and sometimes nowhere at all. Indeed, it is even possible for the different child variables to be active simultaneously.

Split Lifetime Example A simple example of a split lifetime variable can be seen in the following straight-line code fragment:

```
A = ...;        ! Define (assign value to) A
...
B = ...A...;    ! Use definition (value of) A
A = ...;        ! Define A again
...
C = ...A...;    ! Use latter definition A
```

In this example, the first value assigned to variable *A* is used later in the assignment to variable *B* and then never used again. A new value is assigned to *A* and used in the assignment to variable *C*.

Without changing the meaning of this fragment, we can rewrite the code as

```
A1 = ...;       ! Define A1
...
B = ...A1...;   ! Use A1
A2 = ...;       ! Define A2
...
C = ...A2...;   ! Use A2
```

where variables *A*1 and *A*2 are split child variables of *A*.

Because *A*1 and *A*2 are independent, the following is also an equivalent fragment:

```
A1 = ...;       ! Define A1
...
A2 = ...;       ! Define A2
B = ...A1...;   ! Use A1
...
C = ...A2...;   ! Use A2
```

Here, we see that the value of *A*2 is assigned while the value of *A*1 is still alive. That is, the split children of a single variable have overlapping lifetimes.

This example illustrates that split lifetime optimization is possible even in simple straight-line code. Moreover, other optimizations can create opportunities for split lifetime optimization that may not be apparent from casual examination of the original source. In particular, loop unrolling (in which the body of a loop is replicated several times in a row) can create loop bodies for which split lifetime optimization is feasible and desirable.

Variables of Interest Our implementation deals only with scalar variables and parameters. This includes Alpha's extended precision floating-point (128-bit X_Floating) variables as well as variables of any of the complex types (see Sites[16]). These latter variables are referred to as two-part variables because each requires two registers to hold its value.

Currency Definition

The value of a variable in an optimized program is current with respect to a given position in the source program if the variable holds the value that would be expected in an unoptimized version of the program. Several kinds of optimization can lead to noncurrent variables. Consider the currency example in Figure 1.

As shown in Figure 1, the optimizing compiler has chosen to change the order of operations so that line 4 is executed prior to line 3. Now suppose that execution has stopped at the instruction in line 3 of the unoptimized code, the line that assigns a value to variable *C*.

Given a request to display (print) the value of *A*, a traditional debugger will display whatever value happens to be contained in the location of *A*, which here, in the optimized code, happens to be the result of the second assignment to *A*. This displayed value of *A* is a correct value, but it is not the expected value that should be displayed at line 3. This scenario might easily mislead a user into a frustrating and fruitless attempt to determine how the assignment in line 1 is computing and assigning the wrong value. The problem occurs because the compiler has moved the second assignment so that it is early relative to line 3.

Another currency example can be seen in the fragment (taken from Copperman[3]) that appears in Figure 2. In this case, the optimizing compiler has chosen to omit the second assignment to variable *A* and to assign that value directly into the actual parameter location used for the call of routine FOO. Suppose that the debugger is stopped at the call of routine FOO. Given a request to display *A*, a traditional debugger is likely to display the result of the first assignment to *A*. Again, this value is an actual value of *A*, but it is not the expected value.

Alternatively, it is possible that prior to reaching the call, the optimizing compiler has decided to reuse the

```
Line  Unoptimized                                        Optimized
1     A = ...;      ! Define A                           A = ...;
2     B = ...A...;  ! Use A                              B = ...A...;
3     C = ...;      ! C does not depend on A             A = ...;
4     A = ...;      ! Define A                           C = ...;
5     D = ...A...;  ! Use second definition of A         D = ...A...;
```

Figure 1
Currency Example 1

```
Line  Unoptimized                                Optimized
1     A = expression1;                           A = expression1;
2     B = ...A...;       ! Use 1st def. of A     B = ...A...;
3     A = expression2;
4     FOO(A);            ! Use 2nd def. of A     FOO(expression2);
```

Figure 2
Currency Example 2

location that originally held the first value of *A* for another purpose. In this case, no value of *A* is available to display at the call of routine FOO.

Finally, consider the example shown in Figure 3, which illustrates that the currency of a variable is not a property that is invariant over time. Suppose that execution is stopped at line 5, inside the loop. In this case, *A* is not current during the first time through the loop body because the actual value comes from line 3 (moved from inside the loop); it should come from line 1. On subsequent times through the loop, the value from line 3 is the expected value, and the value of *A* is current.

As discussed earlier, most approaches to currency determination involve making certain kinds of flow graph and compiler optimization information available to the debugger so that it can report when a displayed value is not current. However, we wanted to avoid adding major new kinds of analysis capability to DIGITAL's debuggers.

More fundamentally, as the degree of optimization increases, the notion of *current position* in the program itself becomes increasingly ambiguous. Even when the particular instruction at which execution is pending can be clearly and unequivocally related to a particular source location, this location is not automatically the best one to use for currency determination. Nevertheless, the source location (or set of locations) where a displayed value was assigned can be reliably reported without needing to establish the current position.

Accordingly, we use an approach different than those considered in the literature. We use a straightforward flow analysis formulation to determine what locations hold values of user variables at any given point in the program and combine this with the set of definition locations that provide those values. Because there may be more than one source location, the user is given the basic information to determine where in the source the value of a variable may have originated. Consequently, the user can determine whether the value displayed is appropriate for his or her purpose.

Compiler Processing

A compiler performs most split lifetime analysis on a routine-by-routine basis. A preliminary walk over the entire symbol table identifies the variable symbols that are of interest for further analysis. Then, for each routine, the compiler performs the following steps:

- Code cell prepass
- Flow graph construction
- Basic block processing
- Parameter processing
- Backward propagation
- Forward propagation
- Information promotion and cleanup

After the compiler completes this processing for all routines, a symbol table postwalk performs final cleanup tasks. The following contains a brief discussion of these steps.

In this summary, we highlight only the main characteristics of general interest. In particular, we assume that each location, such as a register, is independent of all other locations. This assumption is not appropriate to locations on the stack because variables of different sizes

```
Line  Unoptimized                                Optimized
1     A = ...;                                   A = ...;
2     ...A...;                                   ...A...;
3                                                A = ...;
4     while (...) {                              while (...) {
5     ...;                                       ...;
6     A = ...;          // A is loop invariant
7     }                                          }
```

Figure 3
Currency Example 3

may overlay each other. The complexity of dealing with overlapping allocations is beyond the scope of this paper.

Of special importance in this processing is the fact that each operand of every instruction includes a *base symbol* field that refers to the compiler's symbol table entry for the entity that is involved.

Symbol Table Prewalk The symbol table prewalk identifies the variables of interest for analysis. As discussed, we are interested in scalars corresponding to user variables (not compiler-created temporaries), including Alpha's extended precision floating-point (128-bit X_Floating) and complex values.

DIGITAL's FORTRAN implementations pass parameters using a by-reference mechanism with bind (rather than copy-in/copy-out) semantics. GEM treats the hidden reference value as a variable that is subject to split lifetime optimization. Since the reference variable must be available to effect operations on the logical parameter variable, it follows that both the abstract parameter and its reference value must be treated as interesting variables.

Code Cell Prepass The code cell prepass performs a single walk over all code cells to determine

- The maximum and minimum offsets in the stack frame that hold any interesting variables
- The highest numbered register that is actually referenced by the code
- Whether the stack frame uses a frame pointer that is separate from the stack pointer

The compiler uses these characteristics to preallocate various working storage areas.

Flow Graph Construction A flow graph is built, in which each basic block is a node of the graph.

Basic Block Processing Basic block processing performs a kind of symbolic execution of the instructions of each block, keeping track of the effect on machine state as execution progresses.

When an instruction operand writes to a location with a base symbol that indicates an interesting variable, the compiler updates the location description to indicate that the variable is now known to reside in that location—this begins a lifetime segment. The instruction that assigned the value is also recorded with the lifetime segment.

If there was previously a known variable in that location, that lifetime segment is ended (even if it was for the same variable). The beginning and ending instructions for that segment are then recorded with the variable in the symbol table.

When an instruction reads an operand with a base symbol that indicates an interesting variable, some more unusual processing applies.

If the variable being read is already known to occupy that location, then no further processing is required. This is the most common case.

If the location already contains some other known variable, then the variable being read is added to the set of variables for that location. This situation can arise when there is an assignment of one variable to another and the register allocator arranges to allocate them both to the same location. As a result, the assignment happens implicitly.

If the location does not contain a known variable but there is a write operation to that location earlier in the same block (a fact that is available from the location description), the prior write is retroactively treated as though it did write that variable at the earlier instruction. This situation can arise when the result of a function call is assigned to a variable and the register allocator arranges to allocate that variable in the register where the call returns its value. The code cell representation for the call contains nothing that indicates a write to the variable; all that is known is that the return value location is written as a result of the call. Only when a later code cell indicates that it is using the value of a known variable from that location can we infer more of what actually happened.

If the location does not contain a known variable and there is no write to that same location earlier in this same basic block, then the defining instruction cannot be immediately determined. A location description is created for the beginning of the basic block indicating that the given variable or set of variables must have been defined in some predecessor block. Of course, the contents known as a result of the read operation can also propagate forward toward the end of the block, just as for any other read or write operation.

Special care is needed to deal with a two-part variable. Such a variable does not become defined until both instructions that assign the value have been encountered. Similarly, any reuse of either of the two locations ends the lifetime segment of the variable as a whole.

At the end of basic block processing, location descriptions specify what is known about the contents of each location as a result of read and write operations that occurred in the block. This description indicates the set of variables that occupy the location, or that the location was last written by some value that is not the value of a user variable, or that the location does not change during execution of the block.

Parameter Processing The compiler models parameters as locations that are defined with the contents of a known variable at the entry point of a routine.

Backward Propagation Backward propagation iterates over the flow graph and uses the locations with known contents at the beginning of a block to work backward to predecessor blocks looking for instructions that write to that location. For each variable in each input location, any such prior write instruction is retroactively made to look like a definition of the variable. Note that this propagation is not a flow algorithm because no convergence criteria is involved; it is simply a kind of spanning walk.

Forward Propagation Forward propagation iterates over the flow graph and uses the locations with known contents at the end of each block to work forward to successor blocks to provide known contents at the beginning of other blocks. This is a classic "reaching definitions" flow algorithm, in which the input state of a location for a block is the intersection of the known contents from the predecessors.

In our case, the compiler also propagates definition points, which are the addresses of the instructions that begin the lifetime segments. For those variables that are known to occupy a location, the set of definitions is the union of all the definitions that flow into that location.

Information Promotion and Cleanup The final step of compiler processing is to combine information for adjacent blocks where possible. This action saves space in the debugging symbol table but does not affect the accuracy of the description. Descriptions for by-reference bind parameters are next merged with the descriptions for the associated reference variables. Finally, lifetime segment information not already associated with symbol table entries is copied back.

Object File Representation

The object file debugging symbol table representation for split lifetime variables is actually quite simple. Instead of a single address for a variable, there is a sequence of lifetime segment descriptions. Each lifetime segment consists of

- The range of addresses over which the child location applies
- The location (in a register, at a certain offset in the current stack frame, indirect through a register or stack location, etc.)
- The set of addresses that provide definitions for this lifetime segment

By convention, the last segment in the sequence can have the address range 0 to FFFFFFFF (hex). This address range is used for a static variable, for example in a FORTRAN COMMON block, that has a default allocation that applies whenever no active children exist.

Debugger Processing

Name resolution, that is, binding a textual name to the appropriate entry in the debug symbol table, is in no way affected by whether or not a variable has split lifetime segments. After the symbol table entry is found, any sequence of lifetime segments is searched for one that includes the current point of execution indicated by the program counter (PC). If found, the location of the value is taken from that segment. Otherwise, the value of the variable is not available.

Usage Example

To illustrate how a user sees the results of this processing, consider the small C program in Figure 4. Note that the numbers in the left column are listing line numbers.

When DOCT8 is compiled, linked, and executed under debugger control, the dialogue shown in Figure 5 appears. The figure also includes interpretive comments.

Known Limitations

The following limitations apply to the existing split lifetime support.

Multiple Active Split Children While the compiler analysis correctly determines multiple active split child variables and the debug symbol table correctly describes them, OpenVMS DEBUG does not currently support multiple active child variables. When searching a symbol's lifetime segments for one that includes the current PC, the first match is taken as the only match.

Two-part Variables Support for two-part variables (those occupying two registers) assumes that a complete definition will occur within a single basic block.

```
385  doct8 () {
386
387      int i, j, k;
388
389      i = 1;
390      j = 2;
391      k = 3;
392
393      if (foo(i)) {
394          j = 17;
395          }
396      else {
397          k = 18;
398          }
399
400      printf("%d, %d, %d\n", i, j, k);
401
402      }
```

Figure 4
C Example Routine DOCT8 (Source with Listing Line Numbers)

```
$ run doct8
        OpenVMS Alpha Debug64 Version T7.2-001
%I, language is C, module set to DOCT8
DBG> step/into
stepped to DOCT8\doct8\%LINE 391
   391:     k = 3;
DBG> examine i, j, k
%W, entity 'i' was not allocated in memory (was optimized away)
%W, entity 'j' does not have a value at the current PC
%W, entity 'k' does not have a value at the current PC
```

Note the difference in the message for variable *i* compared to the messages for variables *j* and *k*. We see that variable *i* was not allocated in memory (registers or otherwise), so there is no point in ever trying to examine its value again. Variables *j* and *k*, however, do not have a value "at the current PC." Somewhere later in the program they will have a value, but not here.

The dialogue continues as follows:

```
DBG> step 6
stepped to DOCT8\doct8\%LINE 391
   391:     k = 3;
DBG> step
stepped to DOCT8\doct8\%LINE 393
   393:     if (foo(i)) {
DBG> examine j, k
%W, entity 'j' does not have a value at the current PC
DOCT8\doct8\k:  3
    value defined at DOCT8\doct8\%LINE 391
```

Here we see that *j* is still undefined but *k* now has a value, namely 3, which was assigned at line 391. The source indicates that *j* was assigned a value at line 390, before the assignment to *k*, but *j*'s assignment has yet to occur.

Skipping ahead in the dialogue to the print statement at line 400, we see the following:

```
DBG> set break %line 400
DBG> go
break at DOCT8\doct8\%LINE 400
   400:     printf("%d, %d, %d\n", i, j, k);
DBG> examine j
DOCT8\doct8\j:  2
    value defined at DOCT8\doct8\%LINE 390
    value defined at DOCT8\doct8\%LINE 394
DBG> examine k
DOCT8\doct8\k:  18
    value defined at DOCT8\doct8\%LINE 397+4
    value defined at DOCT8\doct8\%LINE 391
```

This portion of the message shows that more than one definition location is given for both *j* and *k*. Which of each pair applies depends on which path was taken in the `if` statement. If a variable has an apparently inappropriate value, this mechanism provides a means to take a closer look at those places, and only those places, from which that value might have come.

Figure 5
Dialogue Resulting from Running DOCT8

That is, at the end of a basic block, if the second part of a definition is missing then the initial part is discarded and forgotten.

Consider the following FORTRAN fragment:

```
COMPLEX X, Y
...
X = ...
Y = X + (1.0, 0.0)
```

Suppose that the last use of variable *X* occurs in the assignment to variable *Y* so that *X* and *Y* can be and are allocated in the same location, in particular, the same register pair. In this case, the definition of *Y* requires only one instruction, which adds 1.0 to the real part of the location shared by *X* and *Y*. Because there is no second instruction to indicate completion of the definition, the definition will be lost by our implementation.

Semantic Stepping

A major problem with stepping by line though optimized code is that the apparent source program location "bounces" back and forth, with the same line often appearing again and again. In large part this bouncing is due to a compiler optimization called *code scheduling,* in which instructions that arise from the same source line are scheduled, that is, reordered and intermixed with other instructions, for better execution performance.

OpenVMS DEBUG, like most debuggers, interprets the `STEP/LINE` (step by line) command to mean that the program should execute until the line number changes. Line numbers change more frequently in scheduled code than in unoptimized code.

For example, in sample programs from the SPEC95 Benchmark Suite, the average number of instructions in sequence that share the same line number is typically between 2 and 3—and typically 50 to 70 percent of those sequences consist of just 1 instruction! In contrast, if only instruction-level scheduling is disabled, then the average number of instructions is between 4 and 6, with 20 to 30 percent consisting of one instruction. In a compilation with no optimization, there are 8 to 12 instructions in a sequence, with roughly 5 percent consisting of a single instruction.

A second problem with stepping by line through an optimized program is that, because of the behavior of revisiting the same line again and again, the user is never quite sure when the line has finished executing. It is unclear when an assignment actually occurs or a control flow decision is about to be made.

In unoptimized code, when a user requests a breakpoint on a certain line, the user expects execution to stop just before that line, hence before the line is carried out. In optimized code, however, there is no well-defined location that is "before the line is carried out," because the code for that line is typically scattered about, intermixed, and even combined with the code for various other lines. It is usually possible, however, to identify *the* instruction that actually carries out the effect of the line.

Semantic Event Concept

We introduce a new kind of stepping mode called semantic stepping to address these problems. Semantic stepping allows the program to execute up to, but not including, an instruction that causes a semantic effect. Instructions that cause semantic effects are instructions that

- Assign a value to a user variable
- Make a control flow decision
- Make a routine call

Not all such instructions are appropriate, however. We start with an initial set of candidate instructions and refine it. The following sections describe the heuristics that are currently in use.

Assignment The candidates for assignment events are the instructions that assign a value to a variable (or to one of its split children). The second instruction in an assignment to a two-part variable is excluded. Stopping between the two assignments is inadvisable because at that point the variable no longer has the complete old state and does not yet have the complete new state.

Branches There are two kinds of branch: unconditional and conditional. An unconditional branch may have a known destination or an unknown destination. Unconditional branches with known destinations most often arise as part of some larger semantic construct such as an if-then-else or a loop. For example, code for an if-then-else construct generally has an implicit join that occurs at the end of the statement. The join takes the form of a jump from the end of one alternative to the location just past the last instruction of the other (which has no explicit jump and falls through into the next statement). This jump turns the inherently symmetric join at the source level into an asymmetric construction at the code stream level.

Unconditional jumps almost never define interesting semantic events—some related instruction usually provides a more useful event point, such as the termination test in the case of a loop. One exception is a simple goto statement, but these are very often optimized away in any case. Consequently, unconditional branches with known destinations are not treated as semantic events.

Unconditional branches with unknown destinations are really conditional branches: they arise from constructs such as a C `switch` statement implemented as a table dispatch or a FORTRAN assigned `GO TO` statement. These branches definitely are interesting points at which to allow user interaction before the new direction is taken. Thus, the compiler retains unconditional branches as semantic events.

Similarly, in general, conditional branches to known destinations are important semantic event points. Often more than one branch instruction is generated for a single high-level source construct, for example, a decision tree of tests and branches used to implement a small C `switch` statement. In this case, only the first in the execution sequence is used as the semantic event point.

Calls Most calls are visible to a user and constitute semantically interesting events. However, calls to some run-time library routines are usually not interesting because these calls are perceived to be merely software implementations of primitive operations, such as integer division in the case of the Alpha architecture. GEM internally marks calls to all its own run-time support routines as not semantically interesting. Compiler front ends accomplish this where appropriate for their own set of run-time support routines by setting a flag on the associated entry symbol node.

Compiler Processing

In most cases, the compiler can identify semantic event locations by simple predicates on each instruction. The exceptions are

- The second of the two instructions that assign values to a two-part variable is identified during split lifetime analysis.
- Conditional branches that are part of a larger construct are identified during a simple pass over the flow graph.

Object Module Representation

The object module debugging semantic event representation contains a sequence of address and event kind pairs, in ascending address order.

Debugger Processing

Semantic stepping in the debugger involves a new algorithm for determining the range of instructions to execute. This algorithm is built on a debugger primitive mechanism that supports full-speed execution of user instructions within a given range of addresses but traps any transfer out of that range, whether by reaching the end or by executing any kind of branch or call instruction.

Semantic stepping works as follows. Starting with the current program counter address, OpenVMS DEBUG finds the next higher address that is a semantic event point; this is the target event point. OpenVMS DEBUG executes instructions in the address range that starts at the address of the current instruction and ends at the instruction that precedes the target event point. The range execution terminates in the following two cases:

1. If the next instruction to execute is the target event point, then execution reached the end of target range and the step operation is complete.
2. If the next instruction to execute is not the target event point, then the next address becomes the current address and the process repeats (silently).

Note that, unlike the algorithm that determines the range for stepping by line, the new algorithm does not require an explicit test for the kind of instruction, in particular, to test if it is a kind of branch. The compiler already marks branches with the semantic event attribute, if appropriate. Also unlike the traditional stepping-by-line algorithm, the new algorithm does not consider the source line number.

Visible Effect

With semantic stepping, a user's perception of forward progress through the code is no longer dominated by the side effects of code scheduling, that is, stopping every few instructions regardless of what is happening. Rather, this perception is much more closely related to the actual semantic behavior, that is, stopping every statement or so, independent of how many instructions from disparate statements may have executed.

Note that jumping forward and backward in the source may still occur, for example, when code motions have changed the order in which semantic actions are performed. Nothing about semantic event handling attempts to hide such reordering.

Inlining

Procedure call inlining can be confusing when using a traditional debugger. For example, if routine INNER is inlined into routine CALLER and the current point of execution is within INNER, should the debugger report the current source location as at a location in the caller routine or in the called routine? Neither is completely satisfactory by itself. If the current line is reported as at the location within INNER, then that information will appear to conflict with information from a call stack traceback, which would not show routine INNER. If the current line is reported as though in CALLER, then relevant location information from the callee will be obscured or suppressed. Worse yet, in the case of nested inlining, potentially crucial information about the intermediate call path may not be available in any form.

The problem of dealing with inlining was solved long ago by Zellweger[1]—at least the topic has not been treated again since. Zellweger's approach adds additional information to an otherwise traditional table that maps from instruction addresses to the corresponding source line numbers. Our approach is different: it includes additional information in the scope description of the debugging symbol table.

A key underpinning for inline support is the ability to accurately describe scopes that consist of multiple discontiguous ranges of instruction addresses, rather than the traditional single range. This capability is quite independent of inlining as such. However, because code from an inlined routine is freely scheduled with other code from the calling context, dealing accurately with the resulting disjoint scopes is an essential building block for effective support.

Goals for Debugger Support

Our overall goal is to support debugging of inlined code with expected behavior, that is, as though the inlining has not occurred. More specifically, we seek to provide the ability to

- Report the source location corresponding to the current position in the code
- Display parameters and local variables of an inlined routine
- Show a traceback that includes call frames corresponding to inlined routines
- Set a breakpoint at a given routine entry
- Set a breakpoint at a given line number (from within an inlined routine)
- Call an inlined routine

We have achieved these goals to a substantial extent.

GEM Locators

Before describing the mechanisms for inlining, we introduce the GEM notion of a *locator.* A locator describes a place in the source text. The simplest kinds of locator describe a point in the source, including the name of the file, the line within that file, and the column within that line; they even describe the point at which that file was included by another file (as for a C or C++ `#include` directive), if applicable.

A crucial characteristic of locators is that they are all of a uniform fixed size that is no larger than an integer or pointer. (How this is achieved is beyond the scope of this paper.) In particular, locators are small enough that every tuple node in the intermediate language (IL) and every code cell in the generated code stream contains one. Moreover, GEM as a whole is quite meticulous about maintaining and propagating high-quality locator information throughout its optimization and code generation.

An additional kind of locator was introduced for inlining support. This *inline locator* encodes a pair that consists of a locator (which may also be an inline locator) and the address of an associated scope node in the GEM symbol table.

Compiler Processing

Debugging optimized code support for inlining generally builds on and is a minor enhancement of the GEM inlining mechanism. Inlining occurs during an early part of the GEM optimizer phase.

Inlining is implemented in GEM as follows:

- Within the scope that contains the call site, an *inline scope* block is introduced. This scope represents the result of the inlining operation. It is populated with local variable declarations that correspond one-to-one with the formal parameters of the inlined routine.
- The actual arguments of the call are transformed into assignments that initialize the values of the surrogate parameter variables.
- The inline scope is also made to contain a *body scope,* which is a copy of the body of the inlined routine, including a copy of its local variables.
- The original call is replaced with a jump to a copy of the IL for the body of the routine, in which references to declarations or parameters of the routine are replaced with references to their corresponding copied declarations. In addition, returns from the routine are replaced with jumps back to the tuple following the original call.
- Similar "boundary adjustments" are made to deal with function results, output parameters, choice of entry point (when there is more than one, as might occur for FORTRAN alternate entry statements), etc. (The bookkeeping is a bit intricate, but it is conceptually straightforward.)

The calling routine, which now incorporates a copy of the inlined routine, is then further processed as a normal (though larger) routine.

Inlining Annotations for Debugging The main changes introduced for debugging optimized code support are as follows.

- The newly created inline scope block is annotated with additional information, namely,
  - A pointer to the routine declaration being inlined.
  - The locator from the call that is replaced. In a simple call with no arguments, there may be nothing left in the IL from the original call after inlining is completed; this locator captures the original call location for possible later use, for example, as a supplement to the information that maps instruction addresses to source line numbers.
- As the code list of the original inlined routine is copied, each locator from the original is replaced by a new inline locator that records
  - The original locator.
  - The newly created inline scope into which it is being copied.

As a result of these steps, every inlined instruction can be related back to the scope into which it was inlined and hence to the routine from which it was inlined, regardless of how it may be modified or moved as a result of subsequent optimization.

Note that these additional steps are an exception to the general assertion that debugging optimized code support occurs after code generation and just prior to object code emission. These steps in no way influence the generated code—only the debugging symbol table that is output.

Prologue and Epilogue Sets The prologue of a routine generally consists of those instructions at the beginning of the routine that establish the routine stack frame (for example, allocate stack and save the return address and other preserved registers) and that must be executed before a debugger can usefully interpret the state of the routine. For this reason, setting a breakpoint at the beginning of a routine is usually (transparently) implemented by setting a breakpoint after the prologue of that routine is completed.

Conversely, the epilogue of a routine consists of those instructions at the end of a routine that tear down the stack frame, reestablish the caller's context, and make the return value, if any, available to the caller. For this reason, stopping at the end of a routine is usually (transparently) implemented by setting a breakpoint before the epilogue of that routine begins.

One benefit of inlining is that most prologue and epilogue code is avoided; however, there may still be some scope management associated with scope entry and exit. Also, some programming language–related environment management associated with the scope may exist and should be treated in a manner analogous to traditional prologue and epilogue code. The problem is how to identify it, because most of the traditional compiler code generation hooks do not apply.

The model we chose takes advantage of the semantic event information that we describe in the section Semantic Stepping. In particular, we define the first semantic event that can be executed within the inlined routine to be the end of the prologue. For reasons discussed later, we define the last instruction (not the last semantic event) of the inlined code as the beginning of the epilogue. As a result of unrelated optimization effects, each of these may turn out to be a set of instructions. Determination of inline prologue and epilogue sets occurs after split lifetime and semantic event determination is completed so that the results of those analyses can be used.

To determine the set of prologue instructions, for each inline instance, GEM starts with every possible entry block and scans forward through the flow graph looking for the first semantic event instruction that can be reached from that entry. The set of such instructions constitutes the prologue set for that instance of the inlined routine.

This is a spanning walk forward from the routine entry (or entries) that stops either when a block is found to contain an instruction from the given inline instance or when the block has already been encountered (each block is considered at most once). Note that there may be execution paths that include one or more instructions from an inlining, none of which is a semantic event instruction.

The set of epilogue instructions is determined using an inverse of the prologue algorithm. The process starts with each possible exit block and scans backward through the flow graph looking for the last instruction (that is, the instruction closest to the routine exit) of an inline instance that can reach an exit.

Note that prologue and epilogue sets are not strictly symmetric: prologue sets consist of only instructions that are also semantic events, whereas epilogue sets include instructions that may or may not be semantic events.

Object Module Representation

To describe any inlining that may have occurred during compilation, we include three new kinds of information in the debugging symbol table.

If the instructions contained in a scope do not form a single contiguous range, then the description of the scope is augmented with a discontiguous range description. This description consists of a sequence of ranges. (The scope itself indicates the traditional approximate range description to provide backward compatibility with older versions of OpenVMS DEBUG). This augmented description applies to all scopes, whether or not they are the result of inlining.

For a scope that results from inlining a call, the description of the scope is augmented with a record that refers to the routine that was inlined as well as the line number of the call. Each scope also contains two entries that consist of the sequence of prologue and epilogue addresses, respectively.

Backward compatibility is fully maintained. An older version of OpenVMS DEBUG that does not recognize the new kinds of information will simply ignore it.

Debugger Processing

As the debugger reads the debugging symbol table of a module, it constructs a list of the inlined instances for each routine. This process makes it possible to find all instances of a given routine. Note, however, that if every call of the routine is expanded inline and the routine cannot otherwise be called from outside that module, then GEM does not create a noninlined (closed-form) version of the routine.

Report Source Location It is a simple process to report the source location that corresponds to the current code address. When stopped inside the code resulting from an inlined routine, the program counter maps directly to a source line within the inlined routine.

Display Parameters and Local Variables As is the case for a noninlined routine, the scope description for an inlined routine contains copies of the parameters and the local variables. No special processing is required to perform name binding for such entities.

Include Inlined Calls in Traceback The debugger presents inlined routines as if they are real routine calls. A stack frame whose current code address corresponds to an inlined routine instance is described with two or more virtual stack frames: one or more for the inlined instance(s) and one for the ultimate caller. (An example is shown later in Figure 7.)

Set Breakpoints at Inlined Routine Instances The strategy for setting breakpoints at inlined routines is based on a generalization of processing that previously existed for C++ member functions. Compilation of C++ modules can result in code for a given member function being compiled every time the class or template definition that contains the member function is compiled. We refer to all these compilations as *clones.* (It is not necessary to distinguish which of them is the original.) In our generalization, an inlined routine call instance is treated like a clone. To set a breakpoint at a routine, the debugger sets breakpoints at all the end-of-prologue addresses of every clone of the given routine in all the currently active modules.

Set Breakpoints at Inlined Line Number Instances The strategy for setting breakpoints on line numbers shares some features of setting breakpoints on routines, with additional complications. Compiler-reported line numbers on OpenVMS systems are unique across all the files included in a compilation. It follows that the same file included in more than one compilation may have different associated line numbers.

To set a breakpoint at a particular line number, that line number needs to be first normalized relative to the containing file. This normalized line number value is then compared to normalized line numbers for that same file that are included in other compilations. (If different versions of the same named file occur in different compilations, the versions are treated as unrelated.) The original line number is converted into the set of address ranges that correspond to it in all modules, taking into account inlining and cloning.

Call a Routine That Is Inlined If the compiler creates a closed-form version of a routine, then the debugger can call that routine independent of whether there may also be inlined instances of the routine. If no such version of the routine exists, then the debugger cannot call the routine.

Usage Example

Inlining support has many aspects, but we will illustrate only one—a call traceback that includes inlined calls. Consider the sample program shown in Figure 6. This program has four routines: three are combined in a single file (enabling the GEM FORTRAN compiler to perform inline optimizations), and the last is in a separate file. To help correlate the lines of code in

```
Line +++++ File DOCFJ-INLINE-2.FOR
---
1    C      Main routine
2    C
3           INTEGER A, C
4           TYPE *, A(3, C(0))
5           END
6    C
7           FUNCTION A(I, L)
8           INTEGER A, B
9           A = B(5, I) + 2*L
10          RETURN
11          END
12   C
13          FUNCTION B(J, K)
14          INTEGER B, C
15          B = C(9) + J + K
16          END

     +++++ File DOCFJ-INLINE-2A.FOR
1    C
2           FUNCTION C(I)
3           INTEGER C
4           C = 2*I
5           RETURN
6           END
```

Figure 6
Program to Illustrate Inlining Support

these two files with those in Figure 7, we added line numbers to the left of the code. Note that these numbers are not part of the program.

If we compile, link, and run this program using the OpenVMS DEBUG option, we can step to a place in routine B that is just before the call to routine C and then request a traceback of the call stack. This dialogue is shown in Figure 7.

Figure 7 shows that pseudo stack frames are reported for routines A and B, even though the call of routine B has been inlined into routine A and the call of routine A has been inlined into the main program. The main difference from a real stack frame is the extra line that reports that the "above routine is inlined."

Limitations

In a real stack frame, it is possible to examine (and even deposit into) the real machine registers, rather than examine the variables that happen to be allocated in machine registers. In an inlined stack frame, this operation is not well defined and consequently not supported. In a noninlined stack frame, these operations are still allowed.

An attractive feature that would round out the expected behavior of inlined routine calls would be to support stepping into or over the inlined call in the same way that is possible for noninlined calls. This feature is not currently supported—execution always steps into the call.

```
GEMEVN$ run DOCFJ-INLINE-2
         OpenVMS Alpha Debug64 Version T7.2-001
%I, Language: FORTRAN, Module: DOCFJ-INLINE-2$MAIN
...
DBG> step/semantic
stepped to DOCFJ-INLINE-2$MAIN\A\B\%LINE 15+8
    15:        B = C(9) + J + K
DBG> show calls
 module name  routine name line      rel PC           abs PC
*DOCFJ-INLINE-2$MAIN
              B             15  000000000000001C 000000000002006C
----- above routine is inlined
*DOCFJ-INLINE-2$MAIN
              A              9  0000000000000004 0000000000020054
----- above routine is inlined
*DOCFJ-INLINE-2$MAIN
              DOCFJ-INLINE-2$MAIN
                             4  0000000000000038 0000000000020038
                                0000000000000000 FFFFFFFF8590716C
```

Figure 7
OpenVMS DEBUG Dialogue to Illustrate Inlining Support

Performance and Resource Usage

We gathered a number of statistics to determine typical resource requirements for using the enhanced debugging optimized code capability compared to the traditional practice of debugging unoptimized code. A short summary of the findings follows.

- All metrics tend to show wide variance from program to program, especially small ones.
- Generating traditional debugging symbol information increases the size of object modules typically by 50 to 100 percent on the OpenVMS system. Executable image sizes show similar but smaller size increases.
- Generating enhanced symbol table information adds about 2 to 5 percent to the typical compilation time, although higher percentages have been seen for unusually large programs.
- Generating enhanced symbol table information uses significant memory during compilation but does not affect the peak memory requirement of a compilation.
- Generating enhanced symbol table information further increases the size of the symbol table information compared to that for an unoptimized compilation. On the OpenVMS system, this adds 100 to 200 percent to the debugging symbol table of object modules and perhaps 50 to 100 percent for executable images.
- Compiling with full optimization reduces the resulting image size. Total net image size increases typically by 50 to 80 percent.

A more detailed presentation of findings follows. Tables 1 through 3 present data collected using production OpenVMS Alpha native compilers built in December 1996. In developing these results, we used five combinations of compilation options as follows:

S1: no optimization (noopt), no debugging information (nodebug, nodbgopt)

S2: no optimization (noopt), normal debugging information (debug, nodbgopt)

S4: full (default) optimization (opt), no debugging information (nodebug, nodbgopt)

S5: full optimization (opt), normal debugging information only (debug, nodbgopt)

S8: full optimization (opt), enhanced debugging information (debug, dbgopt)

Note that the option combination numbering system is historical; we retained the system to help keep data logs consistent over time.

Compile-time Speed

The incremental compile-time cost of creating enhanced symbol table information is presented in Table 1 for a sampling of BLISS, C, and FORTRAN modules. The data in this table can be summarized as follows:

- Traditional debugging (column 1) increases the total compilation time by about 1 percent.
- Enhanced debugging (column 2) increases the compilation time by about 4 percent. The largest component of that time, approximately 3 percent, is attributed to the flow analysis involved in handling split lifetime variables (column 3).
- Debugging tends to increase as a percentage of time in larger modules, which suggests that processing time is slightly nonlinear in program size; however, this increase does not seem to be excessive even in very large modules.

Compile-time Space

The compile-time memory usage during the creation of enhanced symbol information is presented in Table 2.

Table 1
Percent of Compilation Time Used to Create/Output Debugging Information

Module	S2 (noopt, debug, nodbgopt)	S8 (opt, debug, dbgopt)	(Split Lifetime Analysis Only)
		BLISS CODE	
GEM_AN	0.3%	1.1%	0.7%
GEM_DB	0.9	1.8	1.3
GEM_DF	0.8	5.2	4.4
GEM_FB	0.7	3.5	2.7
GEM_IL_PEEP	0.6	14.4	13.9
		C CODE	
C_METRIC	1.5	5.2	4.1
GRAM	0.5	2.9	2.2
INTERP	1.2	4.5	3.2
		FORTRAN CODE	
MATRIX300X	nm	nm	nm
NAGL	1.4	13.0	11.9
SPICE_V07	3.0	6.4	4.7
WAVEX	2.5	6.3	4.8
	----	----	----
Average	1.2%	4.3%	3.2%
Typical range	(0.5%–1.5%)	(3.0%–7.0%)	(2.0%–5.0%)

Note: "nm" represents "not meaningful," that is, too small to be accurately measured.

Table 2
Key Dynamic Memory Zone Sizes during BLISS GEM Compilations

File	Peak Total	SYMBOL ZONE	EIL ZONE	CODE ZONE	OM ZONE	% Peak	% Larg	% EIL
				BLISS CODE				
GEM_AN	2,507	130	85	184	15	6%	8%	18%
GEM_DF	11,305	836	1,672	2,056	1,180	10	57	71
GEM_FB	4,694	316	522	457	304	6	58	58
GEM_IL_PEEP	40,419	1,606	17,666	4,411	14,143	34	80	80
				C CODE				
C_METRIC	7,381	1,115	494	2,563	167	2	6	34
GRAM	3,031	82	815	211	267	9	33	33
INTERP	3,563	354	308	688	131	4	20	43
				FORTRAN CODE				
MATRIX300X	934	143	227	101	58	6	26	26
NAGL	6,267	1,520	1,791	1,742	68	11	38	38
SPICE_V07	6,234	1,051	3,256	885	459	7	14	14
WAVEX	12,812	4,676	3,119	3,482	68	5	14	22
						---	---	---
Average						9%	32%	40%

Note: All numbers to the left of the vertical bar are thousands of bytes, not multiples of 1,024.

Column Key:

Column	Description
Peak Total	The peak dynamic memory allocated in all zones during the compilation
SYMBOL ZONE	The zone that holds the GEM symbol table
EIL ZONE	The zone that holds the largest EIL ZONE (used for the expanded intermediate representation)
CODE ZONE	The zone that holds the GEM generated code list
OM ZONE	The zone that holds split lifetime and other working data
%Peak	The OM ZONE size as a percentage of the Peak Total size
%Larg	The OM ZONE size as a percentage of the largest single zone in the compilation
%EIL	The OM ZONE size as a percentage of the EIL ZONE size

The following is a summary of the data, where OM ZONE refers to the temporary working virtual memory zone used for split lifetime analysis:

- The OM ZONE size averages about 10 percent of the peak compilation size.
- The OM ZONE size is one-quarter to one-half of the EIL ZONE size. (The latter is well known for typically being the largest zone in a GEM compilation.)
- Since the OM ZONE is created and destroyed after all EIL ZONEs are destroyed, the OM ZONE does not contribute to establishing the peak total size.

Object Module Size

The increased size of enhanced symbol table information for both object files and executable image files is shown in Table 3.

In Table 3, the application or group of modules is identified in the first column. The columns labeled S1, S2, etc. give the resulting size for the combination of compilation options described earlier. Object module and executable image data is presented in successive rows.

Three ratios of particular interest are computed.

S2/S1: This ratio shows the object or image size with traditional debugging information compared to a base compilation without any debugging information. This ratio indicates the additional cost, in terms of increased object and image file size, associated with doing traditional symbolic debugging.

(S8-S5)/(S2-S1): This ratio shows the increase in debugging symbol table size (exclusive of base object, image text, etc.) due to the inclusion of enhanced information compared to the traditional symbol table size.

S8/S2: This ratio shows the object or image size with enhanced debugging information with optimization compared to the traditional debugging size without optimization.

The last ratio, S8/S2, is especially interesting because it combines two effects: (1) the reduction in size as a result of compiler optimization, and (2) the increase in size because the larger debugging symbol table needed to describe the result of the optimization. The resulting net increase is reasonably modest.

Summary and Conclusions

There exists a small but significant literature regarding the debugging of optimized code, yet very few debuggers take advantage of what is known. In this paper we describe the new capabilities for debugging optimized code that are now supported in the GEM compiler system and the OpenVMS DEBUG component of the OpenVMS Alpha operating system. These capabilities deal with split lifetime variables and currency determination, semantic stepping, and procedure inlining. For each case, we describe the problem addressed and then present an overview of GEM compiler and OpenVMS DEBUG processing and the object module representation that mediates between them. All but the inlining support are included in OpenVMS DEBUG V7.0 and in GEM-based compilers for Alpha systems that have been shipping since 1996. The inlining support is

Table 3
Object/Executable (.OBJ/.EXE) File Sizes (in Number of Blocks) for Various OpenVMS Components

File	S1 noopt nodebug nodbgopt	S2 noopt debug nodbgopt	S2/S1 Ratio	S4 opt nodebug nodbdopt	S5 opt debug nodbgopt	S8 opt debug dbgopt	(S8-S5)/ (S2-S1) Ratio	S8/S2 Ratio
				BLISS CODE				
GEM_*.OBJ	31,477	51,069	1.62	27,483	47,031	68,728	1.11	1.35
GEM_*.EXE	12,160	29,543	2.43	10,373	27,755	32,288	0.26	1.09
				C CODE				
C_METRIC.OBJ	436	653	1.50	478	733	1,680	4.36	2.57
C_METRIC.EXE	250	348	1.39	250	385	581	2.00	1.67
GRAM.OBJ	102	120	1.19	100	117	224	5.94	1.87
GRAM.EXE	60	70	1.17	58	69	91	2.20	1.30
INTERP.OBJ	140	207	1.48	134	205	450	3.66	2.17
INTERP.EXE	80	113	1.41	75	113	167	1.64	1.47
				FORTRAN CODE				
MATRIX300X.OBJ	20	34	1.70	16	29	71	3.00	2.08
MATRIX300X.EXE	19	29	1.53	15	25	34	0.90	1.17
NAGL.OBJ	42	63	1.51	288	509	1,178	3.11	1.84
NAGL.EXE	289	388	1.34	187	333	469	1.37	1.21
SPICE.OBJ	1,652	3,117	1.89	1,073	2,571	4,916	1.60	1.58
SPICE.EXE	1,031	1,660	1.61	549	1,318	1,803	0.77	1.09
WAVEX.OBJ	555	1,639	2.95	393	1,556	2,949	1.29	1.80
WAVEX.EXE	634	1,190	1.88	490	1,167	1,437	0.49	1.21

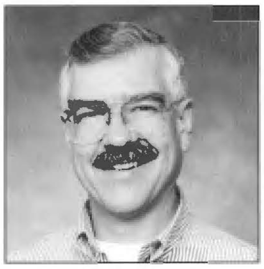
currently in field test. Work is under way to provide similar capabilities in the ladebug debugger[17,18] component of the DIGITAL UNIX operating system.

There are and will always be more opportunities and new challenges to improve the ability to debug optimized code. Perhaps the biggest problem of all is to figure out where best to focus future attention. It is easy to see how the capabilities described in this paper provide major benefits. We find it much harder to see what capability could provide the next major increment in debugging effectiveness when working with optimized code.

References

1. P. Zellweger, "Interactive Source-Level Debugging of Optimized Programs," Ph.D. Dissertation, University of California, Xerox PARC CSL-84-5 (May 1984).
2. J. Hennessy, "Symbolic Debugging of Optimized Code," *ACM Transactions on Programming Languages and Systems,* vol. 4, no. 3 (July 1982): 323–344.
3. M. Copperman, "Debugging Optimized Code Without Being Misled," Ph.D. Dissertation, University of California at Santa Cruz, UCSC Technical Report UCSC-CRL-93-21 (June 11, 1993).
4. G. Brooks, G. Hansen, and S. Simmons, "A New Approach to Debugging Optimized Code," *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, SIGPLAN Notices,* vol. 27, no. 7 (July 1992): 1–11.
5. Convex Computer Corporation, *CONVEX CXdb Concepts* (Richardson, Tex.: Convex Press, Order No. DSW–471, May 1991).
6. D. Coutant, S. Meloy, and M. Ruscetta, "DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* Atlanta, Ga. (June 22–24, 1988): 125–134.
7. L. Zurawski, "Source-Level Debugging of Globally Optimized Code with Expected Behavior," Ph.D. Dissertation, University of Illinois at Urbana-Champaign (1989).
8. U. Hölzle, C. Chambers, and D. Ungar, "Debugging Optimized Code with Dynamic Deoptimization," *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* San Francisco, Calif. (June 17–19, 1992) and SIGPLAN Notices, vol. 27, no. 7 (July 1992): 32–43.
9. L. Pollock and M. Soffa, "High-level Debugging with the Aid of an Incremental Optimizer," *Proceedings of the 21st Hawaii International Conference on System Sciences* (January 1988): 524–532.
10. L. Pollock, M. Bivens, and M. Soffa, "Debugging Optimized Code via Tailoring," *International Symposium on Software Testing and Analysis* (August 1994).
11. P. Feiler, "A Language-Oriented Interactive Programming Environment Based on Compilation Technology," Ph.D. Dissertation, Carnegie-Mellon University, CMU-CS-82-117 (May 1982).
12. A. Adl-Tabatabi, "Source-Level Debugging of Globally Optimized Code," Ph.D. Dissertation, Carnegie Mellon University, CMU-CS-96-133 (June 1996).
13. D. Blickstein et al., "The GEM Optimizing Compiler System," *Digital Technical Journal,* vol. 4, no. 4 (Special Issue 1992): 121–136.
14. B. Beander, "VAX DEBUG: An Interactive, Symbolic, Multilingual Debugger," *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, ACM SIGPLAN Notices,* vol. 18, no. 8 (August 1983): 173–179.
15. *OpenVMS Debugger Manual,* Order No. AA-QSBJB-TE (Maynard, Mass.: Digital Equipment Corporation, November 1996).
16. R. Sites, ed., *Alpha Architecture Reference Manual,* 3d ed. (Woburn, Mass.: Digital Press, 1998).
17. T. Bingham, N. Hobbs, and D. Husson, "Experiences Developing and Using an Object-Oriented Library for Program Manipulation," *OOPSLA Conference Proceedings, ACM SIGPLAN Notices,* vol. 12, no. 10 (October 1993): 83–89.
18. *Digital UNIX Ladebug Debugger Manual,* Order No. AA-PZ7EE-T1TE (Maynard, Mass.: Digital Equipment Corporation, March 1996).

Biographies



Ronald F. Brender
Ronald F. Brender is a senior consultant software engineer in Compaq's Core Technology Group, where he is working on both the GEM compiler and the UNIX ladebug projects. During his career, Ron has worked in advanced development and product development roles for BLISS, FORTRAN, Ada, and multilanguage debugging on DIGITAL's DECsystem-10, PDP-11, VAX, and Alpha computer systems. He served as a representative on the ANSI and ISO standards committees for FORTRAN 77 and later for Ada 83, also serving as a U.S. Department of Defense invited Distinguished Reviewer and a member of the Ada Board and the Ada Language Maintenance Committee for more than eight years. Ron joined Digital Equipment Corporation in 1970, after earning the degrees of B.S.E. (engineering sciences), M.S. (applied mathematics), and Ph.D. (computer and communication sciences) in 1965, 1968, and 1969, respectively, all from the University of Michigan. He is a member of the Association for Computing Machinery and the IEEE Computer Society. Ron holds seven patents and has published several papers in the area of programming language design and implementation.

Jeffrey E. Nelson
Jeffrey E. Nelson is a senior software developer at Candle Corporation in Minneapolis, Minnesota. He currently develops message broker software for Roma BSP, Candle's middleware framework for integrating business applications. Previously at DIGITAL, Jeff was a principal software engineer on the OpenVMS and ladebug debugger projects. He specialized in debug symbol table formats, run-time language support, and computer architecture support. He contributed to porting the OpenVMS debugger from the VAX to the Alpha platform. He represented DIGITAL on the industry-wide PLSIG committee that developed the DWARF debugging symbol table format. Jeff holds an M.S. degree in computer science and applications from Virginia Polytechnic Institute and State University and a B.S. degree in computer science from the University of Wisconsin–LaCrosse. Jeff is an alumnus of the Graduate Engineering Education Program (GEEP), has been awarded one patent, and has previously published and presented work in the area of real-time, object-oriented garbage collection.

Mark E. Arsenault
Mark E. Arsenault is a principal software engineer in Compaq's OpenVMS Engineering Group working on the OpenVMS debugger. Mark has implemented support in the debugger for 64-bit addressing, C++, and inlining. He joined DIGITAL in 1981 and has worked on several software development tools, including the BLISS compiler and the Source Code Analyzer. Mark holds two patents, one each for the Heap Analyzer and for the Correlation Facility. He received a B.A. in physics from Boston University in 1981.

William M. McKeeman

Differential Testing for Software

Differential testing, a form of random testing, is a component of a mature testing technology for large software systems. It complements regression testing based on commercial test suites and tests locally developed during product development and deployment. Differential testing requires that two or more comparable systems be available to the tester. These systems are presented with an exhaustive series of mechanically generated test cases. If (we might say when) the results differ or one of the systems loops indefinitely or crashes, the tester has a candidate for a bug-exposing test. Implementing differential testing is an interesting technical problem. Getting it into use is an even more interesting social challenge. This paper is derived from experience in differential testing of compilers and run-time systems at DIGITAL over the last few years and recently at Compaq. A working prototype for testing C compilers is available on the web.

The Testing Problem

Successful commercial computer systems contain tens of millions of lines of handwritten software, all of which is subject to change as competitive pressures motivate the addition of new features in each release. As a practical matter, quality is not a question of correctness, but rather of how many bugs are fixed and how few are introduced in the ongoing development process. If the bug count is increasing, the software is deteriorating.

Quality

Testing is a major contributor to quality—it is the last chance for the development organization to reduce the number of bugs delivered to customers. Typically, developers build a suite of tests that the software must pass to advance to a new release. Three major sources of such tests are the development engineers, who know where to probe the weak points; commercial test suites, which are the arbiters of conformance; and customer complaints, which developers must address to win customer loyalty. All three types of test cases are relevant to customer satisfaction and therefore have value to the developers. The resultant test suite for the software under test becomes intellectual property, encapsulates the accumulated experience of problem fixes, and can contain more lines of code than the software itself.

Testing is always incomplete. The simplest measure of completeness is statement coverage. Instrumentation can be added to the software before it is tested. When a test is run, the instrumentation generates a report detailing which statements are actually executed. Obviously, code that is not executed was not tested. Random testing is a way to make testing more complete. One value of random testing is introducing the unexpected test—1,000 monkeys on the keyboard can produce some surprising and even amusing input! The traditional approach to acquiring such input is to let university students use the software.

Testing software is an active field of endeavor. Interesting starting points for gathering background information and references are the web site maintained by Software Research, Inc.[1] and the book *Software Testing and Quality Assurance*.[2]

Developer Distaste

A development team with a substantial bug backlog does not find it helpful to have an automatic bug finder continually increasing the backlog. The team priority is to address customer complaints before dealing with bugs detected by a robot. Engineers argue that the randomly produced tests do not uncover errors that are likely to bother customers. "Nobody would do *that,*" "That error is not important," and "Don't waste our time; we have plenty of *real* errors to fix" are typical developer retorts.

The complaints have a substantial basis. During a visit to our development group, Professor C. A. R. Hoare of Oxford University succinctly summarized one class of complaints: "You cannot fix an infinite number of bugs one at a time." Some software needs a stronger remedy than a stream of bug reports. Moreover, a stream of bug reports may consume the energy that could be applied in more general and productive ways.

The developer pushback just described indicates that a differential testing effort must be based on a perceived need for better testing from within the product development team. Performing the testing is pointless if the developers cannot or will not use the results.

Differential testing is most easily applicable to software whose quality is already under control, that is, software for which there are few known outstanding errors. Running a very large number of tests and expending team effort only when an error is found becomes an attractive alternative. Team members' morale increases when the software passes millions of hard tests and test coverage of their code expands.

The technology should be important for applications for which there is a high premium on correctness. In particular, product differentiation can be achieved for software that has few failures in comparison to the competition. Differential testing is designed to provide such comparisons.

The technology should also be important for applications for which there is a high premium on independently duplicating the behavior of some existing application. Identical behavior is important when old software is being retired in favor of a new implementation, or when the new software is challenging a dominant competitor.

Seeking an Oracle

The ugliest problem in testing is evaluating the result of a test. A regression harness can automatically check that a result has not changed, but this information serves no purpose unless the result is known to be correct. The very complexity of modern software that drives us to construct tests makes it impractical to provide a priori knowledge of the expected results. The problem is worse for randomly generated tests. There is not likely to be a higher level of reasoning that can be applied, which forces the tester to instead follow the tedious steps that the computer will carry out during the test run. An oracle is needed.

One class of results is easy to evaluate: program crashes. A crash is never the right answer. In the triage that drives a maintenance effort, crashes are assigned to the top priority category. Although this paper does not contain an in-depth discussion of crashes, all crashes caused by differential testing are reported and constitute a substantial portion of the discovered bugs.

Differential testing, which is covered in the following section, provides part of the solution to the problem of needing an oracle. The remainder of the solution is discussed in the section entitled Test Reduction.

Differential Testing

Differential testing addresses a specific problem—the cost of evaluating test results. Every test yields some result. If a single test is fed to several comparable programs (for example, several C compilers), and one program gives a different result, a bug may have been exposed. For usable software, very few generated tests will result in differences. Because it is feasible to generate millions of tests, even a few differences can result in a substantial stream of detected bugs. The trade-off is to use many computer cycles instead of human effort to design and evaluate tests. Particle physicists use the same paradigm: they examine millions of mostly boring events to find a few high-interest particle interactions.

Several issues must be addressed to make differential testing effective. The first issue concerns the quality of the test. Any random string fed to a C compiler yields some result—most likely a diagnostic. Feeding random strings to the compiler soon becomes unproductive, however, because these tests provide only shallow coverage of the compiler logic. Developers must devise tests that drive deep into the tested compiler. The second issue relates to false positives. The results of two tested programs may differ and yet still be correct, depending on the requirements. For example, a C compiler may freely choose among alternatives for unspecified, undefined, or implementation-defined constructs as detailed in the C Standard.[3] Similarly, even for required diagnostics, the form of the diagnostic is unspecified and therefore difficult to compare across systems. The third issue deals with the amount of noise in the generated test case. Given a successful random test, there is likely to be a much shorter test that exposes the same bug. The developer who is seeking to fix the bug strongly prefers to use the shorter test. The fourth issue concerns comparing programs that must run on different platforms. Differential testing is easily adapted to distributed testing.

Test Case Quality

Writing good tests requires a deep knowledge of the system under test. Writing a good test generator requires embedding that same knowledge in the generator. This section presents the testing of C compilers as an example.

Testing C Compilers

For a C compiler, we constructed sample C source files at several levels of increasing quality.

1. Sequence of ASCII characters
2. Sequence of words, separators, and white space
3. Syntactically correct C program
4. Type-correct C program
5. Statically conforming C program
6. Dynamically conforming C program
7. Model-conforming C program

Given a test case selected from any level, we constructed additional nearby test cases by randomly adding or deleting some character or word from the given test case. An altered test case is more likely to cause the compilers to issue a diagnostic or to crash. Both the selected and the altered test cases are valuable.

One of the more entertaining testing papers reports the results of feeding random noise to the C run-time library.[4] A typical library function crashed or hung on 30 percent of the test cases. C compilers should do better, but this hypothesis is worth checking. Only rarely would a tested compiler faced with level 1 input execute any code deeper than the lexer and its diagnostics. One test at this level caused the compiler to crash because an input line was too long for the compiler's buffer.

At level 2, given lexically correct text, parser error detection and diagnostics are tested, and at the same time the lexer is more thoroughly covered. The C Standard describes the form of C tokens and C "white-space" (blanks and comments). It is relatively easy to write a lexeme generator that will eventually produce every correct token and white-space. What surprised us was the kind of bugs that the testing revealed at this level. One compiler could not handle 0x000001 if there were too many leading zeros in the hexadecimal number. Another compiler crashed when faced with the floating-point constant 1E1000. Many compilers failed to properly process digraphs and trigraphs.

Stochastic Grammar

A vocabulary is a set of two kinds of symbols: terminal and nonterminal. The terminal symbols are what one can write down. The nonterminal symbols are names for higher level language structures. For example, the symbol "+" is a terminal symbol, and the symbol "additive-expression" is a nonterminal symbol of the C programming language. A grammar is a set of rules for describing a language. A rule has a left side and a right side. The left side is always a nonterminal symbol. The right side is a sequence of symbols. The rule gives one definition for the structure named by the left side. For example, the rule shown in Figure 1 defines the use of "+" for addition in C. This rule is recursive, defining additive-expression in terms of itself.

There is one special nonterminal symbol called the start symbol. At any time, a nonterminal symbol can be replaced by the right side of a rule for which it is the left side. Beginning with the start symbol, nonterminals can be replaced until there are no more nonterminal symbols. The result of many replacements is a sequence of terminal symbols. If the grammar describes C, the sequence of terminal symbols will form a syntactically correct C program. Randomly generated white-space can be inserted during or after generation.

A stochastic grammar associates a probability with each grammar rule.

For level 2, we wrote a stochastic grammar for lexemes and a Tcl script to interpret the grammar,[5,6] performing the replacements just described. Whenever a nonterminal is to be expanded, a new random number is compared with the fixed rule probabilities to direct the choice of right side.

In either case, at this level and at levels 3 through 7, setting the many fixed choice probabilities permits some control of the distribution of output values. Not all assignments of probabilities make sense. The probabilities for the right sides that define a specific nonterminal must add up to 1.0. The probability of expanding recursive rules must be weighted toward a nonrecursive alternative to avoid a recursion loop in the generator. A system of linear equations can be solved for the expected lengths of strings generated by

```
additive-expression additive-expression + multiplicative-expression
```

Figure 1
Rule That Defines the Use of "+" for Addition in C

each nonterminal. If, for some set of probabilities, all the expected lengths are finite and nonnegative, this set of probabilities ensures that the generator does not often run away.

Increasing Test Quality

At level 3, given syntactically correct text, one would expect to see declaration diagnostics while more thoroughly covering the code in the parser. At this level, the generator is unlikely to produce a test program that will compile. Nevertheless, compiler errors were detected. For example, one parser refused the expression `1==1==1`.

The syntax of C is given in the C Standard. Using the concept of stochastic grammar, it is easy to write a generator that will eventually produce every syntactically correct C translation-unit. In fact, we extended our Tcl lexer grammar to all of C.

At level 4, given a syntactically correct generated program in which every identifier is declared and all expressions are type correct, the lexer, the parser, and a good deal of the semantic logic of the compiler are covered. Some generated test programs compile and execute, giving the first interesting differential testing results. Achieving level 4 is not easy but is relatively straightforward for an experienced compiler writer. A symbol table must be built and the identifier use limited to those identifiers that are already declared. The requirements for combining arithmetic types in C (`int, short, char, float, double` with `long` and/or `unsigned`) were expressed grammatically. Grammar rules defining, for example, int-additive-expression replaced the rules defining additive-expression. The replacements were done systematically for all combinations of arithmetic types and operators. To avoid introducing typographical errors in the defining grammar, much of the grammar itself was generated by auxiliary Tcl programs. The Tcl grammar interpreter did not need to be changed to accommodate this more accurate and voluminous grammatical data. We extended the generator to implement declare-before-use and to provide the derived types of C (`struct, union, pointer`). These necessary improvements led to thousands of lines of tricky implementation detail in Tcl. At this point, Tcl, a nearly structureless language, was reaching its limits as an implementation language.

At level 5, where the static semantics of the C Standard have been factored into the generator, most generated programs compile and run.

Figure 2 contains a fragment of a generated C test program from level 5.

A large percentage of level 5 programs terminate abnormally, typically on a divide-by-zero operation. A peculiarity of C is that many operators produce a Boolean value of 0 or 1. Consequently, a lot of expression results are 0, so it is likely for a division operation to have a zero denominator. Such tests are wasted. The number of wasted tests can be reduced somewhat by setting low probabilities for using divide, for creating Boolean values, or for using Boolean values as divisors.

Regarding level 6, dynamic standards violations cannot be avoided at generation time without a priori choosing not to generate some valid C, so instead we implement post-run analysis. For every discovered difference (potential bug), we regenerate the same test case, replacing each arithmetic operator with a function call, inside which there is a check for standards violations.

The following is a function that checks for "integer shift out of range." (If we were testing C++, we could have used overloading to avoid having to include the type signature in the name of the checking function.)

```
int
int_shl_int_int(int val, int amt) {
    assert(amt >= 0 && amt < sizeof(int)*8);
    return val << amt;
}
```

For example, the generated text

```
a << b
```

is replaced upon regeneration by the text

```
int_shl_int_int(a, b)
```

```
++ ul15 + -- ui8 * ++ ul16 - ( ui17 + ++ ui20 * ( sl21 & ( argc <<=
c14 ) ? ( us23 ) < ++ argc <= ++ sl22 : -- ( ( * & * & sl24 ) ) ==
0160030347u < ++ ( t5u7 ) . sit5m6 & 1731044438u * ++ ui25 * (
unsigned int ) ++ ( ld26 ) ) & ( ( ( 0761 ) * 2137167721L * sl27 ?
ul28 & d12 * ++ d9 * DBL_EPSILON * 7e+4 * ++ d11 + ++ d10 * d12 * (
++ ld31 * .4L * 9.1 - ld32 * ++ f33 - - .7392E-6L * ++ ld34 + 22.82L
+ 1.91 * -- ld35 >= ++ ld37 ) == 9.F + ( ++ f38 ) + ++ f39 *f40 > (
float ) ++ f41 * f42 >= c14 ++ : sc43 & ss44 ) ^ uc13 & .9309L - (
ui18 * 007101U * ui19 ? sc46 -- ? -- ld47 + ld48 : ++ ld49 - ld48 *
++ ld50 : ++ ld51 ) >= 239.611 ) ^ - ++ argc == ( int signed ) argc -
++ ui54 )- ++ ul57 >= ++ ul58 * argc - 9ul * ++ * & ul59 * ++ ul60 ;
```

Figure 2
Generated C Expression

If, on being rerun, the regenerated test case asserts a standards violation (for example, a shift of more than the word length), the test is discarded and testing continues with the next case.

Two problems with the generator remain: (1) obtaining enough output from the generated programs so that differences are visible and (2) ensuring that the generated programs resemble real-world programs so that the developers are interested in the test results. Solving these two problems brings the quality of test input to level 7. The trick here is to begin generating the program not from the C grammar nonterminal symbol translation-unit but rather from a model program described by a more elaborate string in which some of the program is already fully generated. As a simple example, suppose you want to generate a number of print statements at the end of the test program. The starting string of the generating grammar might be

```
# define P(v) printf(#v "=%x\\n", v)

int main() {
    declaration-list
    statement-list
    print-list
    exit(0);
}
```

where the grammatical definition of `print-list` is given by

```
print-list P ( identifier ) ;
print-list print-list P ( identifier ) ;
```

In the starting string above there are three nonterminals for the three lists instead of just one for the standard C start symbol translation-unit. Programs generated from this starting string will cause output just before exit. Because differences caused by rounding error were uninteresting to us, we modified this print macro for types `float` and `double` to print only a few significant digits. With a little more effort, the expansion of `print-list` can be forced to print each variable exactly once.

Alternatively, suppose a test designer receives a bug report from the field, analyzes the report, and fixes the bug. Instead of simply putting the bug-causing case in the regression suite, the test designer can generalize it in the manner just presented so that many similar test cases can be used to explore for other nearby bugs.

The effect of level 7 is to augment the probabilities in the stochastic grammar with more precise and direct means of control.

Forgotten Inputs

The elaborate command-line flags, config files, and environment variables that condition the behavior of programs are also input. Such input can also be generated using the same toolset that is used to generate the test programs. The very first test on the very first run with generated compiler directive flags revealed a bug in a compiler under test—it could not even compile its own header files.

Results

Table 1 indicates the kinds of bugs we discovered during the testing. Only those results that are exhibited by very short text are shown. Some of the results derive from hand generalization of a problem that originally surfaced through random testing.

There was a reason for each result. For example, the server crash occurred when the tested compiler got a stack overflow on a heavily loaded machine with a very large memory. The operating system attempted to dump a gigabyte of compiler stack, which caused all the other active users to thrash, and many of them also dumped for lack of memory. The many disk drives on the server began a dance of the lights that sopped up the remaining free resources, causing the operators to boot the server to recover. Excellent testing can make you unpopular with almost everyone.

Test Distribution

Each tested or comparison program must be executed where it is supported. This may mean different hardware, operating system, and even physical location.

There are numerous ways to utilize a network to distribute tests and then gather the results. One particularly simple way is to use continuously running watcher programs. Each watcher program periodically examines a common file system for the existence of some particular files upon which the program can act. If no files exist, the watcher program sleeps for a while and tries again. On most operating systems, watcher programs can be implemented as command scripts.

There is a test master and a number of test beds. The test master generates the test cases, assigns them to the test beds, and later analyzes the results. Each test bed runs its assigned tests. The test master and test beds share a file space, perhaps via a network. For each test bed there is a test input directory and a test output directory.

A watcher program called the test driver waits until all the (possibly remote) test input directories are empty. The test driver then writes its latest generated test case into each of the test input directories and returns to its watch-sleep cycle. For each test bed there is a test watcher program that waits until there is a file in its test input directory. When a test watcher finds a file to test, the test watcher runs the new test, puts the results in its test output directory, and returns to the watch-sleep cycle. Another watcher program called the test analyzer waits until all the test output directories contain results. Then the results, both input and

Table 1
Results of Testing C Compilers

Source Code	Resulting Problem
if (1.1)	Constant float expression evaluated false
1 ? 1 : 1/0	Several compiler crashes
0.0F/0.0F	Compiler crash
x != 0 ? x/x : 1	Incorrect answer
1 == 1 == 1	Spurious syntax error
-!0	Spurious type error
0x000000000000000	Spurious constant out of range message
0x80000000	Incorrect constant conversion
1E1000	Compiler crash
1 >> INT_MAX	Twenty-minute compile time
'ab'	Inconsistent byte order
int i=sizeof(i=1);	Compiler crash
LDBL_MAX	Incorrect value
(++n,0) ? -- n: 1	Operator ++ ignored
if (sizeof(char)+d) f(d)	Illegal instruction in code generator
i=(unsigned)-1.0F;	Random value
int f(register());	Compiler crash or spurious diagnostic
int (...(x)...);	Enough nested parentheses to kill the compiler Spurious diagnostic (10 parentheses) Compiler crash (100 parentheses) Server crash (10,000 parentheses)
digraphs (<: <% etc.)	Spurious error messages
a/b	The famous Pentium divide bug (we did not catch it but we could have)

output, are collected for analysis, and all the files are deleted from every test input and output directory, thus enabling another cycle to begin.

Using the file system for synchronization is adequate for computations on the scale of a compile-and-execute sequence. Because of the many sleep periods, this distribution system runs efficiently but not fast. If throughput becomes a problem, the test system designer can provide more sophisticated remote execution. The distribution solution as described is neither robust against crashes and loops nor easy to start. It is possible to elaborate the watcher programs to respond to a reasonable number of additional requirements.

Test Analysis

The test analyzer can compare the output in various ways. The goal is to discover likely bugs in the compiler under test. The initial step is to distinguish the test results by failure category, using corresponding directories to hold the results. If the compiler under test crashes, the test analyzer writes the test data to the crash directory. If the compiler under test enters an endless loop, the test analyzer writes the test data to the loop directory. If one of the comparison compilers crashes or enters an endless loop, the test analyzer discards the test, since reporting the bugs of a comparison compiler is not a testing objective. If some, but not all, of the test case executions terminate abnormally, the test case is written to the abend directory. If all the test cases run to completion but the output differs, the case is written to the test diff directory. Otherwise, the test case is discarded.

Test Reduction

A tester must examine each filed test case to determine if it exposes a fault in the compiler under test. The first step is to reduce the test to the shortest version that qualifies for examination.

A watcher called the crash analyzer examines the crash directory for files and moves found files to a working directory. The crash analyzer then applies a shortening transformation to the source of the test case and reruns the test. If the compiler under test still crashes, the original test case is replaced by the shortened test case. Otherwise, the change is backed out and a new transformation is tried. We used 23 heuristic transformations, including

- Remove a statement
- Remove a declaration
- Change a constant to 1
- Change an identifier to 1
- Delete a pair of matching braces
- Delete an if clause

When all the transformations have been systematically tried once, the process is started over again. The process is repeated until a whole cycle leaves the source of the test unchanged. A similar process is used for the loop, abend, and diff directories.

The typical result of the test reduction process is to reduce generated C test programs of 500 to 600 lines to equally useful C programs of only a few lines. It is not unusual to use 10,000 or more compile operations during test reduction. The trade-off is using many computer cycles instead of human effort to analyze the ugly generated test case.

Test Presentation

After the shortest form of the test case is ready, the test analyzer wraps it in a command script that

1. Reports environmental information (compiler version, compiler flags, name of the test platform, time of test, etc.)
2. Reports the test output or crash information
3. Reruns the test (the test input is embedded in the script)

The test analyzer writes the command scripts to a results directory.

Test Evaluation and Report

The person who is managing the differential testing setup periodically runs scripts that have accumulated in the results directory to determine which ones expose a problem of interest to the development team. One problem peculiar to random testing is that once a bug is found, it will be found again and again until it is fixed. This argues the case for giving high priority to the bugs exposed by differential testing. Uninteresting and duplicate tests are manually discarded, and the rest are entered into the development team bug queue.

Summary and Directions

Differential testing, suitably tuned to the tested program, complements traditional software testing processes. It finds faults that would otherwise remain undetected. It is cost-effective. It is applicable to a wide range of large software. It has proven unpopular with the developers of the tested software.

This technology exposed new bugs in C compilers each day during its use at DIGITAL. Most of the bugs were in the comparison compilers, but a significant number of bugs in DIGITAL code were found and corrected.

Numerous special-purpose differential testing harnesses were put into use at DIGITAL, each testing some small part of a large program. For example, the C preprocessor, multidimensional Fortran arrays, optimizer constant folding, and a new `printf` function each were tested by ad hoc differential testers.

The Java API (run-time library) is a large body of relatively new code that runs on a wide variety of platforms. Since "Write once, run anywhere" is the Java motto, the standard for conformance is high; however, experience has shown that the standard is difficult to achieve. Differential testing should help. What needs to be done is to generate a sequence of calls into the API on various Java platforms, comparing the results and reporting differences. Technically, this procedure is much simpler than testing C compilers. Chris Rohrs, an MIT intern at DIGITAL, wrote a system entirely in Java, gathering method signature information directly out of the binary class files. This API tester may be used when the quality of the Java API reaches the point where the implementors are not buried in bug reports and when there are more independent implementations of the Java run time.

Differential testing can be used to increase test coverage. Using the coverage data taken from running the standard regression suite as a baseline, the developers can run random tests to see if coverage can be increased. Developers can freely add coverage-increasing tests to the test suite using the test output as an initial oracle. No harm is done because even if the recorded result is wrong, the compiler is no worse off for it. If at a later time a regression is observed on the generated test, either the new or the old version was wrong. The developers are alerted and can react. John Parks and John Hale applied this technology to DIGITAL's C compilers.

The problem of retiring an old compiler in favor of a new one requires the new one to duplicate old behavior so as not to upset the installed base. Differential testing can compare the old and the new, flagging all new results (correct or not) that disagree with the old results.

Differential testing can be used to measure quality. Supposing that the majority rules, a million tests can be run on a set of competing compilers. The metric is failed tests per million runs. The authors of the failed compilers can either fix the bugs or prove the majority wrong. In any case, quality improves.

At Compaq, differential testing opportunities arise regularly and are often satisfied by testing systems that are less elaborate than the original C testing system, which has been retired.

Acknowledgments

This work was begun in the Digital Cambridge Research Laboratory by Andy Payne based on his earlier experience in testing DIGITAL Alpha hardware. The author and August Reinig continued the development as an advanced development project in the compiler product group in Nashua, New Hampshire. Steve Rogers and Christine Gregowske contributed to the work, and Steve eventually placed a free working prototype on the web.[7] Bruce Foster managed and encouraged the project, giving the implementors ideas faster than they could be used.

References and Notes

1. Information on testing is available at http://www.testworks.com/Institute/HotList/.

2. B. Beizer, *Software Testing and Quality Assurance* (New York: Van Nostrand Reinhold, 1984).

3. *ISO/IEC 9899: 1990, Programming Languages— C,* 1st ed. (Geneva, Switzerland: International Organization for Standardization, 1990).

4. B. Miller, "An Empirical Study of Reliability," *CACM,* vol. 33, no. 12 (December 1990): 32–44.

5. Information on Tcl/Tk is available at http://sunscript.sun.com/.

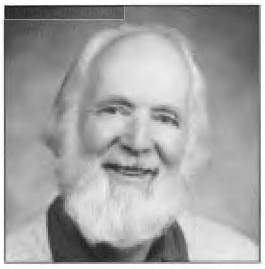
6. J. Ousterhout, *Tcl and the Tk Toolkit* (Reading, Mass.: Addison-Wesley, 1994).

7. Information on DDT distribution is available at http://steve-rogers.com/projects/ddt/.

General Reference

W. McKeeman, A. Reinig, and A. Payne, "Method and Apparatus for Software Testing Using a Differential Testing Technique to Test Compilers," U.S. Patent 5,754,860 (May 1998).

Biography



William M. McKeeman
William McKeeman develops system software for Compaq Computer Corporation. He is a senior consulting engineer in the Core Technology Group. His work encompasses fast-turnaround compilers, unit testing, differential testing, physics simulation, and the Java compiler. Bill came to DIGITAL in 1988 after more than 20 years in academia and research. Most recently, he was a research professor at the Aiken Computation Laboratory of Harvard University, visiting from the Wang Institute Masters in Software Engineering program, where he served as Professor and Chair of the Faculty. He has served on the faculties of the University of California at Santa Cruz and Stanford University and on various state and university computer advisory committees. In addition, he has been an ACM and IEEE National Lecturer and chairman of the 4th Annual Workshop in Microprogramming and is a member of the IFIP Working Group 2.3 on Programming Methodology. Bill founded the Summer Institute in Computer Science programs at Santa Cruz and Stanford and was technical advisor to Boston University for the Wang Institute 1988 Summer Institute. He received a Ph.D. in computer science from Stanford University, an M.A. in mathematics from The George Washington University, a B.A. in mathematics from University of California at Berkeley, and pilot wings from the U.S. Navy. Bill has coauthored 16 patents, 3 books, and numerous published papers in the areas of compilers, programming language design, and programming methodology.

digital™

ISSN 0898-901X