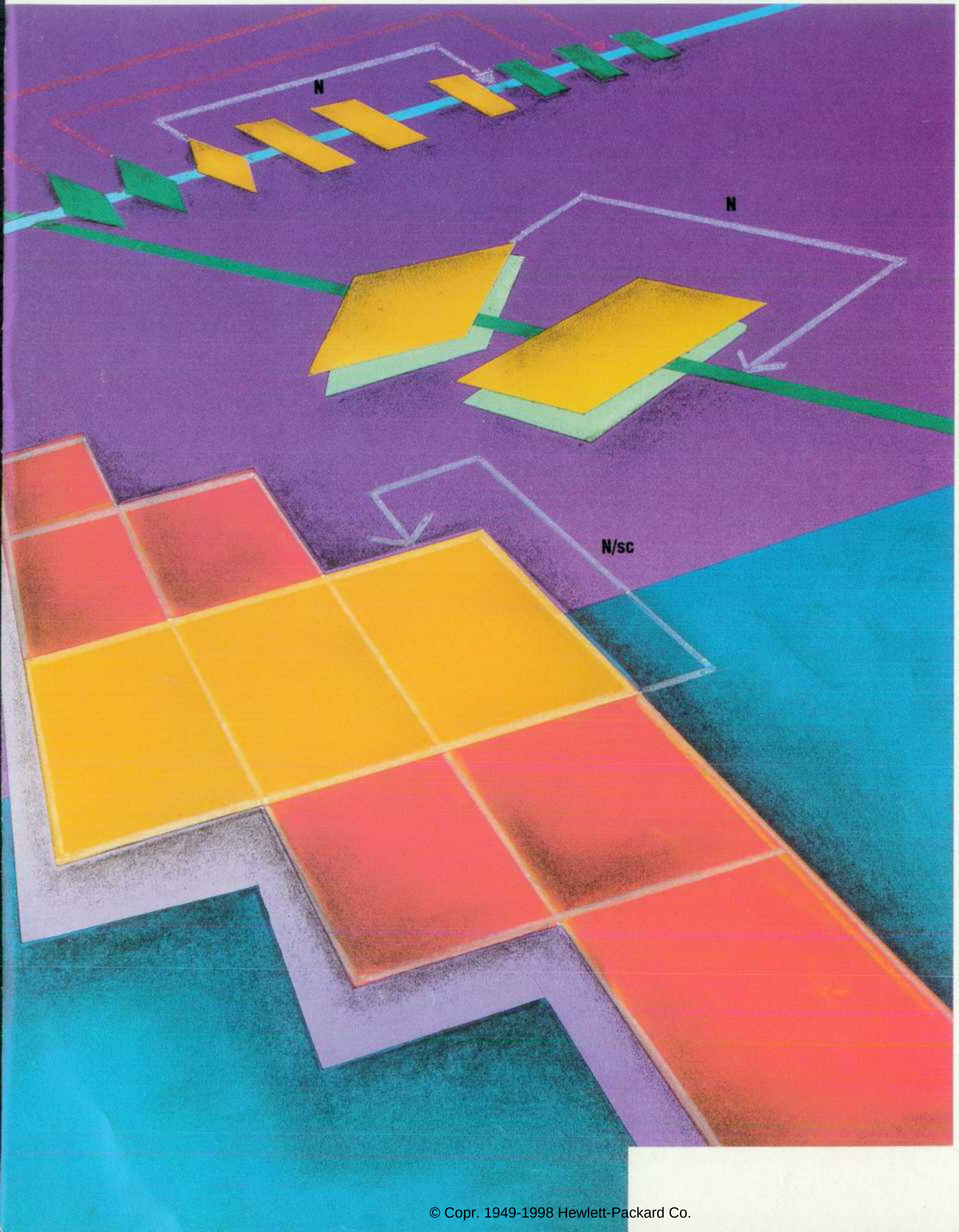
HEWLETT-PACKARD
JOURNAL

June 1992



HEWLETT PACKARD

HEWLETT-PACKARD JOURNAL

June 1992 Volume 43 • Number 3

Articles

Editor, Richard P. Dolan • **Associate Editor**, Charles L. Leath • **Publication Production Manager**, Susan E. Wright • **Illustration**, Renée D. Pighini
Typography/Layout, Rita C. Smith • **Test and Measurement Organization Liaison**, J. Michael Gospe

Research Reports

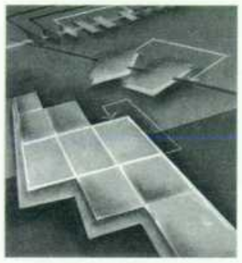
The Hewlett-Packard Journal is published bimonthly by the Hewlett-Packard Company to recognize technical contributions made by Hewlett-Packard (HP) personnel. While the information found in this publication is believed to be accurate, the Hewlett-Packard Company disclaims all warranties of merchantability and fitness for a particular purpose and all obligations and liabilities for damages, including but not limited to indirect, special, or consequential damages, attorney's and expert's fees, and court costs, arising out of or in connection with this publication.

Subscriptions: The Hewlett-Packard Journal is distributed free of charge to HP research, design and manufacturing engineering personnel, as well as to qualified non-HP individuals, libraries, and educational institutions. Please address subscription or change of address requests on printed letterhead (or include a business card) to the HP address on the back cover that is closest to you. When submitting a change of address, please include your zip or postal code and a copy of your old label. Free subscriptions may not be available in all countries.

Submissions: Although articles in the Hewlett-Packard Journal are primarily authored by HP employees, articles from non-HP authors dealing with HP-related research or solutions to technical problems made possible by using HP equipment are also considered for publication. Please contact the Editor before submitting such articles. Also, the Hewlett-Packard Journal encourages technical discussions of the topics presented in recent articles and may publish letters expected to be of interest to readers. Letters should be brief, and are subject to editing by HP.

Please address inquiries, submissions, and requests to: Editor, Hewlett-Packard Journal, 3200 Hillview Avenue, Palo Alto, CA 94304 U.S.A.

In this Issue



Early last year, Hewlett-Packard introduced a family of new workstation computers that surprised the workstation world with their high performance—a huge increase over the previous industry leaders—and their low prices. On standard industry benchmarks, the HP Apollo 9000 Series 700 computers outdistanced the competition by a wide margin. The speed of the Series 700 machines can be attributed to a combination of three factors. One is a new version of HP's PA-RISC architecture called PA-RISC 1.1. (The PA stands for precision architecture and the RISC stands for reduced instruction set computing.) PA-RISC 1.1 was worked on by teams from HP and the former Apollo Computers, Incorporated, then newly acquired by HP. It includes several enhancements specifically aimed at improving workstation performance. The second factor in the new computers' speed is a new set of very large-scale integrated circuit chips capable of operating at clock rates up to 66 megahertz. Called PCX-S, the chipset includes a 577,000-transistor CPU (central processing unit), a 640,000-transistor floating-point coprocessor, and a 185,000-transistor memory and system bus controller. The third factor is a new version of the HP-UX operating system that takes advantage of the architectural enhancements of PA-RISC 1.1 and offers additional compiler optimizations to make programs run faster.

The Series 700 hardware design story will appear in our next issue (August). In this issue we present the software part of the Series 700 speed formula. The article on page 6 summarizes the architectural enhancements of PA-RISC 1.1 and tells how the kernel of the HP-UX operating system was modified to take advantage of them. The article on page 11 describes the development process for the kernel modifications, which was tuned to meet an aggressive schedule without compromising quality. This article includes a brief description of the overall management structure for the Series 700 development project, which is now considered within HP to be a model for future short-time-to-market projects. An overview of the additional compiler optimizations included in the new HP-UX release is provided by the article on page 15, along with performance data showing how the compiler enhancements improve the benchmark performance of the Series 700 workstations. A new optimizing preprocessor for the FORTRAN compiler that improves performance by 30% is described in the article on page 24. Optimization techniques called register reassociation and software pipelining, which help make program loops execute faster, are offered by the new compiler versions and are described in the articles on pages 33 and 39, respectively. The new release of the HP-UX operating system is the first to offer shared libraries, which significantly reduce the use of disk space and allow the operating system to make better use of memory. The HP-UX implementation of shared libraries is described in the article on page 46.

The three research reports in this issue are based on presentations given at the 1991 HP Technical Women's Conference. The first paper (page 54) discusses the integration of an electronic dictionary into HP-NL, HP's natural language understanding system, which was under development at HP Laboratories from 1982 to 1991. Dictionaries are important components of most computational linguistic products, such as machine translation systems, natural language understanding systems, grammar checkers, spelling checkers, and word analyzers. Electronic dictionaries began as word lists and have been evolving, becoming more complex and flexible in response to the needs of linguistic applications. While the electronic dictionary integrated into HP-NL was one of the more advanced and greatly increased the system's capabilities, the integration was not without problems, which the researchers feel should help guide the potential applications of electronic dictionaries. The paper concludes with a survey of applications that can use electronic dictionaries today or in the future.

The paper on page 68 presents the results of research on automated laser printer print quality measurement using spatial frequency methods. Printers using different print algorithms, dot sizes, stroke widths, resolutions, ehnhancement techniques, and toners were given a test pattern to print consisting of concentric circles spaced progressively closer (higher spatial frequency) with increasing radius. The printed test patterns were analyzed by optical methods and measures of relative print quality were computed. These machine evaluations were then compared with the judgments of fourteen trained human observers shown printed samples from the same printers. In all cases, the human jury agreed with the machine evaluations. The method is capable of showing whether printer changes can be expected to improve text, graphics, neither, or both.

Computer graphics rendering is the synthesis of an image on a screen from a mathematical model contained in a computer. Photorealistic renderings, which are produced using global illumunation models, are the most accurate, but they are computation-intensive, requiring minutes for simple models and hours for complex subjects. The paper on page 76 presents the results of simulations of an experimental parallel processor architecture for photorealistic rendering using the raytracing rendering technique. The results so far indicate that four processors operating in parallel can speed up the rendering process by a factor of three. Work continues at HP Laboratories to develop actual hardware to test this architectural concept.

R.P. Dolan
Editor

Cover

The cover shows an artist's rendition of the transformations that take place when source code goes through register reassociation and software pipelining compiler optimizations. The multiple-loop flowchart represents the original source code, the smaller flowchart represents the optimization performed on the innermost loop by register reassociation, and the diagram in the foreground represents software pipelining.

What's Ahead

The August issue will present the hardware design of the HP Apollo 9000 Series 700 workstation computers. Also featured will be the design and manufacturing of the new color print cartridge for the HP DeskJet 500C and DeskWriter C printers, and the driver design for the DeskWriter C. There will also be an article on the HP MRP Action Manager, which provides an interactive user interface for the HP MM materials management software.

HP-UX Operating System Kernel Support for the HP 9000 Series 700 Workstations

Because much of the Series 700 hardware design was influenced by the system's software architecture, engineers working on the kernel code were able to make changes to the kernel that significantly improved overall system performance.

by Karen Kerschen and Jeffrey R. Glasson

When the HP 9000 Series 700 computers were introduced, we in the engineering and learning products organization in the HP-UX* kernel laboratory had a chance to see how our year-long project stacked up against the competition. In a video, we watched a Model 720 workstation pitted against one of our comparably priced competitor's systems. Both systems were running Unigraphics, which is a suite of compute-intensive mechanical modeling programs developed by McDonnell Douglas Corp. The two computers converted images of a General Motors Corvette ZR1 from two to three dimensions, rotated the drawings, contoured the surfaces, and recreated a four-view layout. The Model 720, the lowest-priced of our new systems, performed over eight times faster than the competition.

The Series 700 is based on the first processor to implement the PA-RISC 1.1 architecture, which includes enhancements designed specifically for the technical needs of the workstation market. This was a departure from the previous HP processor design, which served general computation needs.

The new system SPU (system processing unit) features three new chips: an integer processor, a floating-point coprocessor, and a memory and system bus controller. In addition, the Series 700 was developed to provide I/O expandability through the Extended Industry Standard Architecture (EISA) bus. For the software project teams, this new hardware functionality raised some basic questions, such as "What can the user do with these hardware capabilities?" and "What can we do to take advantage of the hardware features?" The answer to the first question was fairly obvious because we knew that key users would be engineers running CAE application programs such as compute-intensive graphics for modeling mechanical engineering designs. We also realized that the Series 700 systems were not intended as specialized systems, but were aimed at a broad spectrum of high-performance workstation applications, and they had to be fast everywhere, without making trade-offs to computer-aided design. Thus, addressing the second question gave direction to the year-long software development effort.

The engineering challenges faced by our kernel development teams were to identify the new features of the hardware that could be exploited by the operating system, and then to add or alter the kernel code to take advantage of these features. By studying the hardware innovations, the software team identified four areas for kernel modification: CPU-related changes, floating-point extensions, TLB (translation lookaside buffer) miss routines, and I/O and memory controller changes. Underlying the entire effort was an essential factor—performance. To succeed in the marketplace, the Series 700 had to have very fast response time and throughput.

The Series 700 performance accomplishments were achieved by a working partnership between hardware and software engineers. Both realized that an integrated system approach was key to making the Series 700 a high-performance machine. New hardware components were engineered to ensure a balanced system, which meant that I/O performance matched CPU performance. Software architecture was considered in designing the hardware, and much of the hardware suggested opportunities for streamlining throughput and response time through changes in the kernel code.

The hardware architecture of the Series 700 is shown in Fig. 1. Each of these components is architected to ensure that the software runs faster. The rest of this article describes the changes to the kernel code to take advantage of the Series 700 hardware features. The management structure and development process are described in the article on page 11.

CPU-Related Changes to Kernel Code

From a hardware perspective, the CPU chip performs all processor functions (except floating-point) including integer arithmetic (except multiplication), branch processing, interrupt processing, data and instruction cache control, and data and instruction memory management. Additional interrupt processing and cache flush instructions were added to the hardware, along with cache hints

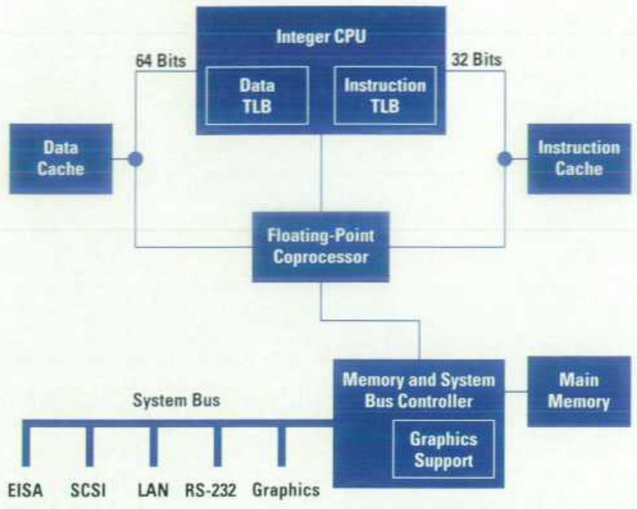


Fig. 1. A block diagram of the HP 9000 Series 700 hardware (TLBs are translation lookaside buffers).

to prefetch cache lines from memory (see the article on page 15 for more information about cache hints).

The software release for the Series 700 operating system was designed to address key features of the CPU chip. To tailor the kernel to the CPU's capabilities required the following changes:

- Emulation of floating-point instructions, which also supports the floating-point coprocessor enhancements
- Cache flush instructions to the I/O and memory controller for the benefit of graphics applications
- Shadow registers for improved TLB (translation lookaside buffer)* miss handling
- 4K-byte page size to reduce TLB miss rate
- Sparse PDIR (page directory), which reduces overhead for the EISA I/O address space and is faster
- New block TLB entries to map the kernel and graphics frame buffers.

Emulation of Floating-Point Instructions

Although all Series 700 systems have floating-point hardware, kernel instructions can now emulate all the new floating-point instructions in software. This redundancy was designed into the software to deal with floating-point exceptions. PA-RISC 1.1 was defined to allow hardware designers the freedom to implement what they wanted efficiently, while still providing a consistent view of the system to software. If someone executes an instruction, the system doesn't care whether it was done in hardware or software—the result is functionally identical, although performance differs. The computation proceeds much more slowly in software than in hardware, but this solution provides a machine without a floating-point coprocessor that can still execute the floating-point instructions and be binary compatible.

The software implementation capability also provides certain classes of operations that the hardware cannot execute. For example, the Series 700 floating-point coprocessor cannot multiply and divide denormalized numbers.** When it encounters denormalized numbers, the hardware generates an assist trap to signal the operating system to emulate the required instruction.

Software engineers modified the kernel to accommodate the expanded floating-point register file and to make these registers accessible as destinations. The additional registers allow more floating-point data to be accessed quickly, which reduces the system's need to access memory in floating-point-intensive applications.

Cache and Cache Flush Instructions

The Series 700 system has separate instruction and data caches (see Fig. 1). This design allows better pipelining of instructions that reference data by giving two ports to the CPU's ALU (arithmetic logic unit). This amounts to a degree of parallel processing in the CPU. To maximize this parallel processing, both cache arrays interface directly to the CPU and the floating-point coprocessor.

The data path from the CPU to the data caches was widened from 32 to 64 bits. This allows two words to be transferred in one cycle between memory and registers. The operating system exploits the 64-bit-wide data path to allow higher throughput between the CPU and memory. The operating system also takes advantage of the widened data path when using floating-point double-word LOADs, STOREs, and quad-word STOREs to COPY and ZERO data in the kernel.

New cache flush instructions have been added to access special dedicated hardware in the memory and system bus controller (discussed in detail later in this article). This hardware does direct memory access (DMA) block moves to and from memory without involving the CPU. It also handles color interpolation and hidden surface removal. These features benefit graphics applications, which use the enhanced cache flush instructions to access data more efficiently.

Shadow Registers

Another CPU feature is the addition of shadow registers. Shadow registers are extensions of the processor that reduce the number of instructions needed to process certain interrupts, particularly TLB misses. The new PA-RISC processor shadows seven general registers. Without shadow registers, when the processor receives an interrupt the operating system must save (reserve) some registers before they can be used to service the interrupt. This is because the operating system has no idea how the general registers are being used at the time of the interrupt. (A user program might be running or executing a system call in the kernel.) Shadow registers eliminate the need for the operating system to store registers before they are used in the interrupt handler. The CPU automatically stores the shadowed registers when the interrupt occurs and before the processor jumps to the interrupt handler. This shortens the interrupt handlers by several

* A translation lookaside buffer or TLB is a hardware address translation table. The TLB and cache memory typically provide an interface to the memory system for PA-RISC processors. The TLB speeds up virtual-to-real address translations by acting as a cache for recent translations. More detailed information about the TLB can be found in reference 1.

**In the IEEE 754 floating-point standard, a denormalized number is a nonzero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significant bit is zero.

instructions. Shadow registers are used for TLB interrupts, which are the most time-critical interrupts.

The CPU has another new instruction—RFIR (return from interrupt and restore)—to return from an interrupt and restore (copy) the shadow registers back to the general registers. RFIR exists along with RFI (return from interrupt), which doesn't copy the shadow registers. RFIR has specific and limited applicability to TLB interrupts because interrupts using the shadow registers cannot be nested. Most of the time, the operating system still uses RFI.

4K-Byte Page Size

To further improve memory access, the page size was increased from 2K bytes to 4K bytes. This reduces the number of TLB misses a typical application will encounter. In the software, changes were made in the low levels of the operating system's virtual memory subsystem. A lot of work was done so that the PA-RISC 1.0 systems, which have a 2K-byte page size, can have a logical 4K-byte page size.

Sparse Page Directory

If we had used the old page directory (PDIR) architecture that maps virtual to physical pages of memory, gaps in the EISA address space would have wasted a significant amount of physical memory to store unused PDIR entries. Therefore, it was decided to redefine the page directory from an array to a linked list. Now, instead of taking the virtual address and calculating an offset in the table, a hash function produces a pointer to a page directory entry (PDE) that corresponds to the physical address. In most cases, the hashing algorithm produces a direct mapping to the point in the table. In some cases, such as a hash collision, the first PDE on the list has to link to another PDE as shown in Fig. 2.

If the translation does not exist in the PDIR, a PDE is taken off the PDE free list and inserted into the correct hash chain. The sparse PDIR reduces the amount of memory needed to store the page tables.

TLB Miss Improvements

The TLB, which is on the processor chip, consists of two 96-entry fully associative TLBs—one for instructions and one for data. Each of these TLBs has block TLB entries—four each for instructions and data. Each fully associative entry maps only a single page, while each block entry is capable of mapping large contiguous ranges of memory, from 128K bytes to 16M bytes. These block entries help reduce TLB misses by permanently mapping large portions of the operating system and graphics frame buffer.

Block entries reduce the number of total TLB entries used by the operating system. Block entry mapping leaves more general TLB entries for user programs and data, thus reducing the frequency of TLB misses and improving overall system performance. We map most of the kernel's text space and a good portion of the kernel's data using block TLB entries.

TLB misses are handled differently by the Series 700 processor than in earlier processor implementations. Miss handler code is invoked when the TLB miss interrupt is generated by the processor. The processor saves some registers in its shadow registers and transfers control to the software TLB miss handler. The miss handler hashes into the sparse PDIR in memory to find a virtual-to-physical translation of the address that caused the interrupt. If it finds it, the translation is installed in the TLB and the transaction is retried. If it doesn't find it, page fault code is executed. (In another case, protection identifiers, which govern access rights, might prevent translation, that is, the address might exist but the process trying to access the data might not have access rights to the data.)

Floating-Point Coprocessor Extensions

The PA-RISC 1.1 architecture features a floating-point coprocessor with an extended floating-point instruction set that has 32 double-precision registers (previously, there were 16). These registers are also accessible as 64 single-precision registers (compared to 16 single-precision registers in the previous implementation). The additional floating-point registers add flexibility in terms of how

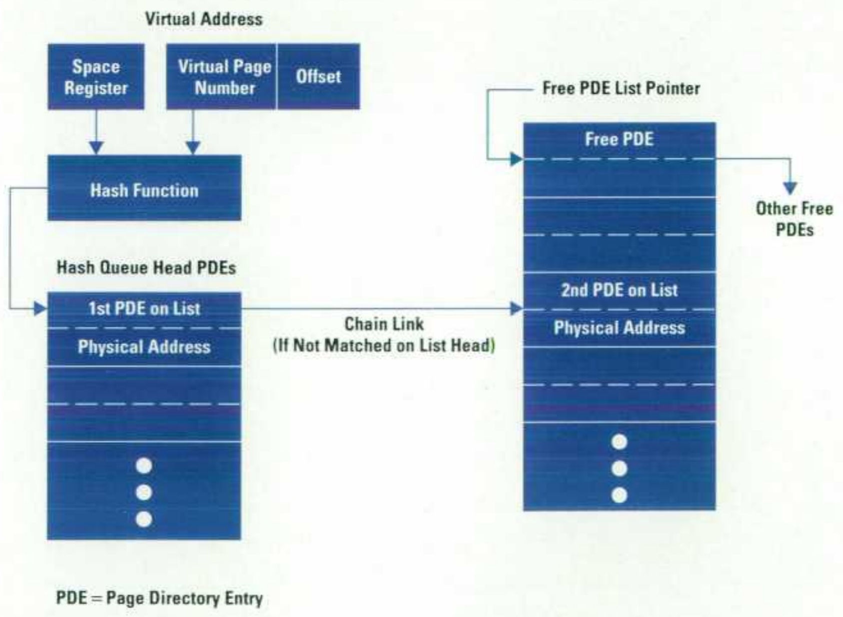


Fig. 2. Virtual-to-physical address translation using linked lists.

many registers a programmer can access. More data can be held in registers that are quickly accessible by the CPU.

Many new floating-point instructions were added to the floating-point instruction set to accommodate more demanding graphics applications and improve matrix manipulations. From the software perspective, the following areas of kernel code were changed:

- Save states
- Store instructions
- Extensions to FTEST
- Multiply instructions
- Floating-point exception handling.

Save States. When a user process gets a signal, the system copies the contents of all its registers (general and floating-point) to the user stack so that the user's signal handler can access and modify them. However, to maintain binary compatibility with older implementations and applications, hooks were added to the kernel to identify the size of the save-state data structure. Knowing the size of this data structure ensures that the system will copy the correct number of registers to the user stack (or copy back from the user stack), so that programs compiled on older (PA-RISC 1.0) hardware will run without having to be recompiled on the new hardware.

Store Instructions. Quad-word store instructions in the floating-point processor store four words (two double-word registers) at once, and execute in fewer cycles than two double-word store instructions. The kernel uses the quad-word store instruction in copy and zero routines, if it detects its presence. Quad store instruction code is not portable to the PA-RISC 1.0 implementation or to other 1.1 systems.

FTEST Extensions. Extensions to FTEST streamline graphics clip tests, which benefits two-dimensional and three-dimensional graphics performance.

FTEST is an instruction used to check the status of subsets of the floating-point compare queue. In previous implementations, FTEST could test only the result of the last FCMP (floating-point compare). The Series 700 extends FCMP to keep the last twelve compare results in a queue, using bits 10 through 20 of the floating-point status register in addition to the C bit (bit 5). FTEST now looks at different pieces of the queue to determine whether to nullify the next instruction. An example of how the FTEST extension can save processor cycles is given on page 10.

FTEST extensions are not part of PA-RISC 1.1 architecture, but are specific to the Series 700. Therefore, any code using the extensions is not portable to other PA-RISC implementations.

Multiply Instructions. New multiple-operation instructions, including multiply-and-add (FMPYADD), multiply-and-subtract (FMPYSUB), and multiply-and-convert from floating-point format to (fixed) integer format (FMPYCFXT), more fully exploit the ALU and MPY computational units in the floating-point coprocessor. This approach reduces the number of cycles required to execute typical computational combinations, such as multiplies and adds.

Also, an integer multiply instruction was added to the instruction set to execute in the floating-point coprocessor. Previously a millicode library routine was called to do integer multiplies. This new implementation is much faster.

Floating-Point Exception Handling. The floating-point coprocessor's computational speed results from the floating-point instructions embedded in the hardware. This provides the biggest performance boost to graphics, particularly for transformations. However, certain circumstances (such as operations on denormalized numbers) cause the hardware to generate an exception, which requires the kernel to emulate the instruction in software. The emulation of the floating-point instruction set provides much-needed backup and auxiliary computational support.

Memory and System Bus Controller

The memory and system bus controller, which was implemented as a new chip, has two new features designed specifically to streamline graphics functionality:

- The ability to read or write from the graphics frame buffer (video memory) to main memory using direct memory access (DMA) circuitry. DMA allows block moves of data to and from the graphics card without having to go through the CPU.
- The capability to do color interpolation (for light source shading) and hidden surface removal.

Accommodating the new hardware system bus and memory functionality required extensive changes to the kernel code.

Besides the operating system changes, the HP Starbase graphics drivers[2] were rewritten to take advantage of block moves and color interpolation. These drivers are used by the X server to improve X Window System performance and allow the use of lower-cost 3D graphics hardware. This is because the drivers use the memory and system bus controller to produce the graphical effects, rather than relying on dedicated graphics hardware.

The memory and system bus controller features new registers in its graphics portion. Kernel code was enhanced to allow user processes to use these additional registers. Any user graphics process can request the kernel to map the memory and system bus controller registers into its address space. A new ioctl system call parameter provides access to the memory and system bus controller registers. The new call maps the controller register set into the user's address space to enable reads and writes from those registers to instruct the memory and system bus controller to perform graphics functions. Once the user sets up the memory and system bus controller registers, the transactions are initiated by issuing a special flush data cache instruction.

Finally, new kernel code allows multiple processes to share the memory and system bus controller. At context-switch time, extra work happens if a process is using the controller. The operating system saves and restores the set of memory and system bus controller registers only if the process has mapped them into its address space.

An Example of the FTEST Instruction

Bits 10 through 20 of the floating-point status register serve two purposes. They are used to return the model and revision of the coprocessor following a **COPR 0,0** instruction as defined by the PA-RISC architecture. Their other use is for a queue of floating-point compare results.

Whenever a floating-point compare instruction (**FCMP**) is executed, the queue is advanced as follows:

```
FPstatus[11:20] = FPstatus[10:19]
FPstatus[10] = FPstatus[5]
FPstatus[5] = FCMP result (the C-bit)
```

The FTEST instruction has been extended to allow checking various combinations of the compare queue. For example, to evaluate **(fr4 == fr5) && (fr6 == fr7) && (fr8 ==fr9) && (fr10 == fr11)** would take 24 cycles to execute using the PA-RISC 1.0 instructions:

```
FCMP,= fr4,fr5
FTEST
branch
FCMP,= fr6,fr7
FTEST
branch
FCMP,=fr8,fr9
FTEST
branch
FCMP,= fr10,fr11
FTEST
branch
```

By comparison, using the Series 700 floating-point compare queue:

```
FCMP,= fr4,fr5
FCMP,= fr6,fr7
FCMP,=fr8,fr9
FCMP,= fr10,fr11
FTEST, ACC4
branch
```

takes only 12 cycles to execute.

Because of this, nongraphics processes are not penalized by the operating system's saving and restoring of unused registers.

Conclusion

Noted author Alvin Toffler identified an underlying challenge to today's computer industry in his book, *Power Shift*. "From now on," he wrote, "the world will be split between the fast and the slow." Thanks to a successful partnership of hardware innovation and kernel tuning, the Series 700 can definitely be classified as one of the fast.

Acknowledgments

The authors wish to thank the engineering teams at the HP-UX kernel laboratory in Cupertino, California for creating the first release of the Series 700 operating system and the engineers at the HP-UX development laboratory in Fort Collins, Colorado for subsequent kernel enhancements. Jon Bayh deserves special recognition for bringing up the HP-UX operating system on the first prototype. For technical reviews, help, and advice, we'd like to extend warm thanks to Eric Hamilton, Jeff Yetter, and Will Walker. Credit for organizing the writing project goes to Sandi Hoag, and for encouragement, Carolyn Godfrey.

References

1. M.J. Mahon, et. al., "Hewlett-Packard Precision Architecture: The Processor," *Hewlett-Packard Journal*, Vol. 30, no. 8, August 1986, pp. 16-17.
2. K. Bronstein, D. Sweetser, W. Yoder, "System Design for Compatibility of a High-Performance Graphics Library and the X Window System," *Hewlett-Packard Journal*, Vol. 40, no. 6, December 1989, p.7.

Providing HP-UX Kernel Functionality on a New PA-RISC Architecture

To ensure customer satisfaction and produce a high-performance, high-quality workstation on a very aggressive schedule, a special management structure, a minimum product feature set, and a modified development process were established.

by Donald E. Bollinger, Frank P. Lemmon, and Dawn L. Yamine

The aggressive schedule for the development of the HP 9000 Series 700 systems required the development team in the HP-UX* kernel laboratory to consider some modifications to the normal software development process, the number of product features, and the management structure. The goals for the product features were to change or add the minimum number of HP-UX kernel functions that would ensure customer satisfaction, meet our performance goals, and adapt to a new I/O system. This version of the HP-UX kernel code became known as minimum core functionality, or MCF.

Series 700 Management Structure

To accomplish the goals set for the HP 9000 Series 700 system required a special management structure that included a program manager with leadership and responsibility for the whole program, and a process that allowed rapid and sound decisions to be made. The resulting management structure delegated the bulk of the management to focused teams of individual developers and first-level managers. The program manager owned every facet of the release, from the software feature set to the allocation of prototypes and production of chips in the fabrication shop. Since the Series 700 program was multidivisional and located in two geographical locations, the program manager had to maintain a desk at both locations, close to the the hardware and software development teams.

The rapid decision policy promoted by the management team enabled small teams of individual developers and first-level managers to make important program decisions quickly and directly. Decision time itself was measured and tracked around the program. For example, the system team's goal was to have no open issues over two weeks old. Also, the MCF kernel team tracked kernel defects on a daily basis. If a defect aged over three days, additional help was assigned immediately. The process to determine the disposition of defects ran on a 24-hour clock. The defect data was posted in the evening, votes were collected by the team captain the next morning, the team reviewed the votes and made decisions in the afternoon, and approved fixes were incorporated into the build that night.

One key decision made early in the program was whether to base the kernel on HP-UX 7.0, which was stable and shipping, or HP-UX 8.0, which was not yet shipping to customers. HP-UX 8.0 offered the advantage of being the basis for future releases, and thus the developers and customers of the follow-on release to MCF could avoid the overhead of first having to update to 8.0. This was a critical decision. The R&D team promoted the advantages of 8.0, while the program manager weighed the risks. Within two weeks the program manager and the team decided to base the operating system on HP-UX 8.0 and the issue was never revisited.

Each team worked systemwide, with individual developers focusing on a facet of the system. The performance team, with members from hardware, kernel, languages, graphics, and performance measurement groups, focused on the overall goal of maximizing system performance in computation, graphics, and I/O. The value added business (VAB) team focused on delivering high-quality prototype hardware and software to key VAB partners, allowing their software applications to release simultaneously with the HP 9000 Model 720. There was also an integration team, a release team, and a quality and testing team.

The members of these teams were not merely representatives who collected action items and returned them to their respective organizations. The team members were the engineers and managers involved in the development work. Thus, multidivisional problems were solved right at the team meetings.

The overall program structure glued these teams together. Key decisions were made by the program manager and other top-level members of the management team. The system team managed the tactical issues and the coordination of the focused teams. Most people were members of multiple teams, providing crucial linkage between individual team goals and organizational goals. There was a rich, almost overwhelming flow of information. The system team appended team reports and product status information to their weekly minutes, which were distributed widely so everyone saw the results.

The rest of this article will discuss the activities performed in the software development process to create the MCF kernel. Technical details of the kernel MCF can be found in the article on page 6.

Quality Control Plan

Historically, the reliability of HP-UX software has been measured in terms of the following items:

- Defect density (the number of defects per one thousand lines of noncomment source statements, or KNCSS)
- Functional test coverage (the number of external interfaces tested and the branch flow coverage values)
- Reliability under stress (continuous hours of operation, or CHO).

The MCF team added the following measures:

- Design and code reviews to ensure high software component quality before delivery to system integration and test
- Weekly integration cycles with full testing participation by the development partners, which included development teams outside of the kernel laboratory.

The weekly integration cycles uncovered a number of interaction and system defects rapidly and early.

The program team designated schedule and quality as the top two priorities of the MCF release. Another program team decision reduced the functionality to the minimum core requirements, which in turn reduced the time to market. The program team also chose to release only one system initially (the Model 720) rather than three (the Models 720, 730, and 750) and to sell the Model 720 in a stand-alone configuration only, rather than supporting it in a diskless cluster.

These decisions resulted in reduced testing complexity. The test setup times highlight the complexity reduction. For the MCF release, the test setup time represented about 1% of the total test time. For the subsequent releases (with Models 710, 720, 730, and 750 participating, in both stand-alone and diskless configurations) the test setup time rose to about 12%. The increase is significant since the test setup time cannot be automated and represents valuable engineering time.

Certification Process

The program team's decision to limit functionality guaranteed a better, more stable system faster. There were fewer components in which defects occurred and there were fewer interfaces where interaction problems arose.

The system test team was able to capitalize on both these benefits. A team decision was made to set a lower reliability goal of 48 continuous hours of operation (CHO) under stress, instead of the traditional 96 CHO. This decision substantially reduced the number of system test cycles required. The system test team next decided to attempt the 48 CHO reliability goal in a single, four-week test cycle. Previous HP-UX releases had required four test cycles, each ranging from two to six weeks.

The single-test-cycle model, a benefit of reduced functionality, emphasized one of the key development goals: "Do it right the first time." This goal was important, because the aggressive MCF schedule did not permit the development teams any time for rework.

In summary, the MCF quality plan featured the following objectives:

- A reduction in configuration and testing complexity
- A single test cycle
- A 48-CHO software certification goal
- The use of design and code reviews before delivering new functionality
- The use of traditional quality measurements before delivery to system integration and test
- Weekly integration cycles with full partner testing participation
- An early baseline established by the quality requirements of the VAB team activities.

Design and Code Reviews

The software engineers in the HP-UX kernel laboratory determined that the best way to achieve the MCF quality objectives was to focus on design and code reviews. Engineers evaluated the effectiveness of their existing review process to find defects before kernel integration and determined that it was not adequate to meet the MCF quality goals. This led to a search for a new design and code review process. Several of the engineers had used a formal review process called software inspection[1] on previous projects, and felt that it would find key defects before kernel integration.

The inspection process was used during the design phase with moderate success. A handful of the engineers had been previously trained on the process. The rest of the engineers simply received a document that described the inspection process. There was no formal training given on inspection roles, criteria, checklist, time requirements, or meeting conduct.

When the inspection meetings began, several of the first-level managers felt that the inspection process was not as successful as it could be. They heard complaints from the engineers about the design documents, insufficient preparation by the inspectors, rambling meetings, and the absence of time estimates in the MCF schedule to perform the process.

The managers put the inspection process on hold and asked an inspection consultant about the complaints they had heard. The consultant gave guidance about the team member's roles, how inspectors should prepare for the meetings, what to focus on during the meetings, and the amount of time required when the process is operating properly.

The managers took this feedback back to the engineers so they could make changes. For example, the time estimate to do the inspections was added to the MCF schedule. This change showed the engineers that they had the opportunity to do inspections, and that the process was considered important. Performing inspections also caused the MCF schedule to slip by two weeks. The program team made an adjustment elsewhere in the program to recover the two weeks.

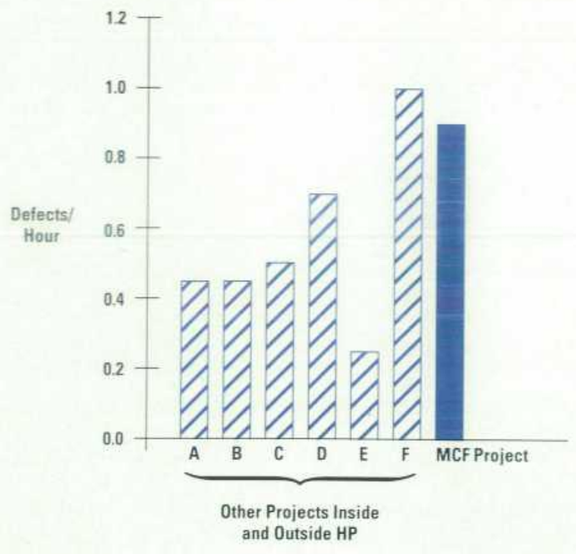


Fig. 1. MCF inspection efficiency compared to other projects inside and outside HP.

The main benefit of using inspections was that important defects were found early. The advance defect visibility minimized schedule delays by reducing the impact on critical path activities. One defect uncovered during an inspection was later estimated to require at least two weeks to isolate and repair if it had been found during system integration.

Fig. 1 compares the MCF inspection efficiency with the results of other projects inside and outside HP. MCF achieved the second best results. Although data was not collected on the number of defects found during product and system testing, the general feeling was that there were fewer defects found in comparison to other HP-UX releases. This feeling was confirmed when the MCF kernel took six weeks to achieve the 48 continuous hours of operation quality goal, compared to previous HP-UX kernels which had taken at least eight weeks.

Branch and Source Management

The kernel sources had been managed by a source control system that permitted multiple development branches to be open at any time. This permitted different development efforts to proceed independently. When the time came to merge branch development into the main trunk, it was necessary to lock the branch. Branch locks ranged on the order of a few days to two weeks, depending on the number of changes and the stability of the resulting kernel. The delays frustrated the engineers who were waiting to include critical path functionality and important defect fixes.

The basic MCF source management philosophy was: "Keep the branch open!" Thus, locking the branch for two weeks was unacceptable.

Two branches were required to implement the aggressive MCF schedule: one to implement the new 4K-byte page size, and the other to implement software support for a new I/O backplane.

Both branches began from the same snapshot of the HP-UX 8.0 kernel sources. As work progressed, a snapshot of the 4K-byte page size team's work was merged with the I/O team's branch. The merge was done in an ongoing, incremental fashion so that no big surprises would appear late in the release and the branch lock time would be minimized.

The merge was accomplished by running a software tool that checked every line, in every file, on both branches. If a file had no changes on either branch the original file was kept. If a file changed on one branch but not the other, the change was incorporated. If a file changed on both branches it was flagged for an engineer to review and resolve manually.

The MCF merge goal was to lock the branch and require engineering review for no more than 36 hours. The goal was consistently met because of the careful attention of the kernel branch administrator and the high degree of team cooperation when assistance was required.

Automated Nightly Build and Test

What new testing challenges did the MCF release present? The key goal was to do a full kernel build and regression test cycle five nights a week, not just once a week as had been done in the past. Could we push the existing process this far? The kernel integration team was uncertain, but was confident that the minimum core functionality model could be capitalized on.

Regression testing revisits the software that has already been tested by the development team. What did the kernel integration team expect to gain from redundant testing? First, to observe, characterize, and resolve any problems detected in the nightly kernel build. Second, at least to match the test results from the previous build, with the goal of converging to zero test failures rapidly.

The MCF regression test plan featured the following:

- A test setup process that was bootstrapped
- Automated software that ran the regression tests five nights a week
- An emphasis placed on parallel operation and the reliable presence of test results
- Automated software that updated the test machines with the system integration team's good system on a weekly basis.

The regression tests for kernel integration included the following:

- File system tests: hierarchical, distributed, networked, and CD-ROM
- Kernel functional tests
- Disk quota functional tests
- Database send and receive functional tests.

The software developers created their own tests to cover new functionality (e.g., SCSI, Centronics, and digital tape interfaces). These tests were run by the development teams directly.

The Test Setup Process

At first test machines were scarce because there were only a handful of hardware prototypes available to the MCF team. Therefore, regression testing began on a standby basis. Eventually, one hardware prototype became available for use on weeknights. This allowed the test setup process to begin in earnest.

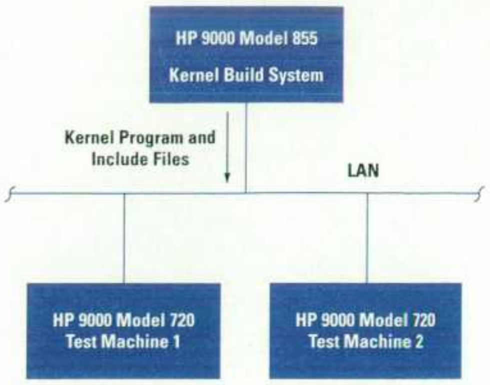

HP 9000 Model 855
Kernel Build System
Kernel Program and Include Files
LAN
HP 9000 Model 720 Test Machine 1
HP 9000 Model 720 Test Machine 2

Fig. 2. The MCF redundancy testing setup.

The least-demanding tests such as the distributed and hierarchical file system tests were installed and run first. After any kernel or test problems were observed, characterized, and resolved by analyzing the results, the setup for the more difficult tests for areas such as the kernel, the network, and the CD-ROM file system began.

Automatic Testing

Software was developed to automate running the regression tests nightly. The software was designed to perform the following tasks:

- Put the kernel program and include files on the test systems
- Reboot the test systems
- Start the tests
- Mail the test results directly to the kernel integration team.

At one point, the kernel began to panic and the regression tests were prematurely interrupted. This caused a problem in receiving a complete set of test results. Fortunately, by this time, two functional prototype machines were available for nightly regression testing (see Fig. 2). The solution was to have both machines run the regression tests each night, but in reverse order. The first machine ran the easier file system tests first, followed by the more demanding kernel functional and remaining tests. The second system ran the same test groups, but in reverse order. The "redundant but reverse order" solution ensured the presence of a full set of test results each morning by combining the output of both systems if required.

Once all the test groups were set up and running, it proved impossible for the automated software to complete them within the six-hour time limit. The problem was solved by modifying the automated software to start as many of the test groups as possible in parallel. The plan was to capitalize on the HP-UX process scheduling abilities and maximize the throughput. One assumption was made using this approach—the tests would not adversely interact with each other. The assumption proved to be true in general. The exceptions were the disk quota, CD-ROM, and system accounting tests, which had conflicts. The automated software was modified to serialize the execution of the disk quota and CD-ROM test groups and run them as a separate stream in parallel with the other test groups. The test administrator chose to handle the system accounting test manually, which continued to fail occasionally because of known conflicts.

Weekly Delivery to System Integration

A complete internally consistent system was built every week, allowing up-to-date software with the latest fixes to be used by the development partners for system integration. To deliver the new system to system integration, the kernel build administrator examined the logs, handled any exceptional conditions, communicated with partners, and then wrote, for review by the management teams, a report that explained what changed from week to week.

On Monday mornings, before the kernel build administrator had arrived to check the logs, the kernel program and the include files were automatically sent from Cupertino, California to the HP-UX commands team in Fort Collins, Colorado. After delivery, the HP-UX commands, which required the include files, began automatically building the system. If the kernel build administrator detected a problem in the error logs, the commands build administrator was called. The two administrators consulted over the telephone whether to let the commands build complete, or to interrupt it. Often, if there was a problem, the kernel delivery was still useful. For example, it was only necessary to interrupt the commands build two or three times out of twenty or more kernel deliveries.

In summary, the weekly delivery process offered the following features:

- Files were delivered in advance, before the tests had certified them
- Rapid team communication was used to notify the partners depending on the delivery if any problem was detected
- Systems delivered were often usable by the partners even when problems were detected
- Problems, status, and any changes were communicated quickly and directly.

Conclusion

The HP-UX kernel laboratory produced a version of the HP-UX operating system that achieved excellent performance and rapid time to market for a new workstation computer, the HP 9000 Model 720. This achievement was made possible by a simplified management structure, the specification of minimum core functionality, a quality control plan that used design and code reviews, and a kernel integration process that featured full automation of the software build, test, and delivery activities.

References

1. M.E. Fagan, "Advances in Software Inspections," *IEEE Transactions of Software Engineering*, Vol. SE-12, no. 7, July 1986, pp. 744-751.

New Optimizations for PA-RISC Compilers

Extensions to the PA-RISC architecture exposed opportunities for code optimizations that enable compilers to produce code that significantly boosts the performance of applications running on PA-RISC machines.

by Robert C. Hansen

Hewlett-Packard's involvement in reduced instruction set computers (RISC) began in the early 1980s when a group was formed to develop a computer architecture powerful and versatile enough to excel in all of Hewlett-Packard's markets including commercial, engineering, scientific, and manufacturing. The designers on this team possessed unusually diverse experience and training. They included compiler designers, operating system designers, microcoders, performance analysts, hardware designers, and system architects. The intent was to bring together different perspectives, so that the team could deal effectively with design trade-offs that cross the traditional boundaries between disciplines. After 18 months of iterative, measurement-oriented evaluation of what computers do during application execution, the group produced an architecture definition known today as Precision Architecture RISC, or PA-RISC.[1,2,3]

In the late 1980s, there were a number of groups that were looking for ways to make Hewlett-Packard more successful in the highly competitive workstation market. These groups realized the need for better floating-point performance and virtual memory management in PA-RISC to produce a low-cost, high-performance, PA-RISC-based workstation product. Experts from these groups and other areas were brought together to collaborate on their ideas and to propose a set of extensions to PA-RISC. Many members of this team were from the then newly acquired Apollo Computers (now HP Apollo). With the knowledge gained from years of experience with PA-RISC and PRISM from Apollo Computers, suggested extensions to the architecture were heavily scrutinized and only accepted after their benefits could be validated. The result was a small but significant set of extensions to PA-RISC now known as PA-RISC 1.1.

Although not a rigid rule, most of the architecture extensions of PA-RISC 1.1 were directed at improving Hewlett-Packard's position in the technical workstation market. Many of the extensions aimed at improving application performance required strong support in the optimizer portion of the PA-RISC compilers. Key technical engineers were reassigned to increase the staff of what had previously been a small optimizer team in HP's California Language Laboratory. In addition, engineers responsible for compiler front ends became involved with supporting new optimization and compatibility options for the two versions of the architecture. Finally, many compiler members from the HP Apollo group shared their insights on how to improve the overall code generation of the PA-RISC 1.1 compilers. The PA-RISC 1.1 extensions, together with enhancements to the optimizing compilers, have enabled Hewlett-Packard to build a low-cost high-performance desktop workstation with industry-leading performance.

The first release of the PA-RISC 1.1 architecture is found in the HP 9000 Series 700 workstations running version 8.05 of the HP-UX* operating system (HP-UX 8.05). The operating system and the compilers for the Series 700 workstation are based on the HP-UX 8.0 operating system, which runs on the HP 9000 Series 800 machines.

This article presents a brief discussion about the architecture extensions, followed by an overview of the enhancements made to the compilers to exploit these extensions. In addition to enhancements made to the compilers to support architecture extensions, there were a number of enhancements to traditional optimizations performed by the compilers that improve application performance, independent of the underlying architecture. These generic enhancements will also be covered. Finally, performance data and an analysis will be presented.

PA-RISC 1.1 Architecture Overview

Most of the extensions to PA-RISC were motivated by technical workstation requirements and were designed to improve performance in the areas of virtual memory management, numerical applications, and graphics, all at the lowest possible cost. Most of the architecture extensions can be exploited by the compilers available on PA-RISC 1.1 implementations. Additional implementation-specific extensions, like special instructions, have been made to improve performance in critical regions of system code and will not be discussed here.

New Instructions

Most implementations of PA-RISC employ a floating-point assist coprocessor to support high-performance numeric processing.[2] It is common for a floating-point coprocessor to contain at least two functional units: one that performs addition and subtraction operations and one that performs multiplication and other operations. These two functional units can accept and process data in parallel. To dispatch

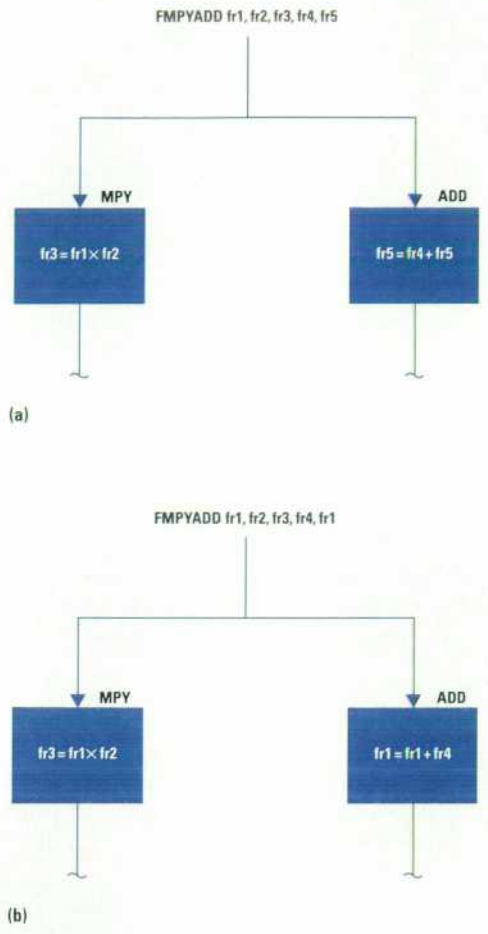


Fig. 1. Legal and illegal uses of the five-operand FMPYADD instruction. Because of parallelism the multiply and add operations execute at the same time. (a) Legal use of the instruction. There is no interdependence between operands. (b) Illegal use of the instruction. The operand in floating-point register fr1 is used in both operations.

operations to these functional units at a higher rate, two five-operand floating-point instructions were added to the instruction set:

- FMPYADD: Floating-point multiply and add
- FMPYSUB: Floating-point multiply and subtract.

In a single instruction, the compiler can specify a floating-point multiplication operation (two source registers and one target) together with an independent floating-point addition or subtraction operation in which one register is both a source and a target. However, because the multiply operation is executed in parallel with the add or subtract operation in a five-operand instruction, the result of one operation cannot be used in the paired operation. For example, in an FMPYADD, the product of the multiplication cannot be used as a source for the addition and vice versa (see Fig. 1).

Since most floating-point multipliers can also perform fixed-point multiplication operations, the unsigned integer multiplication instruction XMPYU was also defined in PA-RISC 1.1. XMPYU operates only on registers in the floating-point register file described below. This dependency implies that fixed-point operands may have to be moved from general registers to floating-point registers and the product moved back to a general-purpose register. Since there is no architected support for moving quantities between general-purpose and floating-point register banks directly, this movement is done through stores and loads from memory. The compiler decides when it is beneficial to use the XMPYU instruction instead of the sophisticated multiplication and division techniques provided in PA-RISC.[4] Signed integer multiplication can also be accomplished using the XMPYU instruction in conjunction with the appropriate extract (EXTRS, EXTRU) instructions.

Additional Floating-Point Registers

To increase the performance for floating-point-intensive code, the PA-RISC 1.1 floating-point register file has been extended. The number of 64-bit (double-precision) registers has been doubled from 16 to 32 (see Fig. 2).

In addition, both halves of each 64-bit register can now be addressed as a 32-bit (single-precision) register, giving a total of 64 single-precision registers compared to only 16 for PA-RISC 1.0. Moreover, contiguous pairs of single-precision values can be loaded or stored using a single double-word load or store instruction. Using a double-word load instruction to load two single-precision quantities can be useful when manipulating single-precision arrays and FORTRAN complex data items.

Cache Hints

On PA-RISC systems, instructions and data are typically fetched and stored to memory through a small, high-speed memory known as a cache. A cache shortens virtual memory access times by keeping copies of the most recently accessed items within its fast memory. The cache is divided into blocks of data and each block has an address tag that corresponds to a block of memory. When the processor accesses an instruction or data, the item is fetched from the appropriate cache block, saving significant time in not having to fetch it from the larger memory system. If the item is not in the cache, a cache

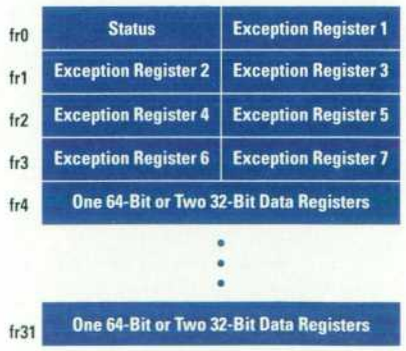


Fig. 2. The floating-point register file contains 28 64-bit data registers and seven 32-bit registers for reporting exceptional conditions. The status register holds information on the current rounding mode, the exception flags, and the exception trap enables for five IEEE exceptions: overflow, underflow, divide by zero, invalid operation, and inexact. If an exception is raised when traps are enabled, an interrupt to the main processor occurs, with the exception and the instruction causing it recorded in an exception register.

miss occurs and the processor may stall until the needed block of memory is brought into the cache.

To increase cache throughput, an extension to cache management has been exposed to the compiler. A bit has been encoded in the store instructions that can be used when the compiler knows that each element in a cache block will be overwritten. This *hint* indicates to the hardware that it is not necessary to fetch the contents of that cache block from memory in the event of a cache miss. This cache hint could be used to provide a significant savings when copying large data structures or initializing pages.

Optimization Enhancements

Optimizing compilers make an important contribution to application performance on PA-RISC processors.[5,6] A single shared optimizer back end* is used in most PA-RISC compilers. When global optimization is enabled, the following traditional transformation phases take place:[7]

- Global data flow and alias analysis (knowing which data items are accessed by code and which data items may overlap is the foundation for many phases that follow)
- Constant propagation (folding and substitution of constant computations)
- Loop invariant code motion (computations within a loop that yield the same result for every iteration)
- Strength reduction (replacing multiplication operations inside a loop with iterative addition operations)
- Redundant load elimination (elimination of loads when the current value is already contained in a register)
- Register promotion (promotion of a data item held in memory to being held in a register)
- Common subexpression elimination (removal of redundant computations and the reuse of the one result)
- Peephole optimizations (use of a dictionary of equivalent instruction patterns to simplify instruction sequences)
- Dead code elimination (removal of code that will not execute)
- Branch optimizations (transformation of branch instruction sequences into more efficient instruction sequences)
- Branch delay slot scheduling (reordering instructions to perform computations in parallel with a branch)
- Graph coloring register allocation (use of a technique called graph coloring to optimize the use of machine registers)
- Instruction scheduling (reordering instructions within a basic block to minimize pipeline interlocks)
- Live variable analysis (removing instructions that compute values that are not needed).

With PA-RISC 1.1, a number of these areas were enhanced to take advantage of the extensions to the architecture. Specifically, the last two transformations, register allocation and instruction scheduling, saw many changes to support the extended floating-point registers and new five-operand instructions.

In addition to the enhancements made to support the architecture extensions, the compiler optimization team spent a considerable amount of time analyzing application code to identify missed optimization opportunities. There was also a very thorough evaluation of Hewlett-Packard's optimizing compilers to see how they matched some key workstation competitors' compilers. Many architecture independent enhancements were identified and added at the same time as the PA-RISC 1.1 enhancements.

These compiler enhancements were integrated with the HP-UX 8.0 compilers available on the HP 9000 Series 800 machines. Because the same base was used, the architecture independent optimization enhancements added to the compilers will also benefit code compiled with the HP-UX 8.0 compilers.

Many of the enhancements to the optimizing compilers led to significant improvements in the Systems Performance Evaluation Cooperative (SPEC) benchmark suite. Release 1.2b of the SPEC suite contains 10 benchmarks that primarily measure CPU (integer and floating-point) performance in the engineering and scientific fields. Performance data for the SPEC benchmarks is presented later in this article.

Improved Register Allocation
Near-optimal use of the available hardware registers is crucial to application performance. Many optimization phases introduce temporary variables or prolong the use of existing register variables over larger portions of a procedure. The PA-RISC optimizer uses an interference graph coloring technique[8] to allocate registers to a procedure's data items. When the coloring register allocator runs out of free registers, it is forced to save or "spill" a register to memory. Spilling a register implies that all instructions that access the item that was spilled must first reload the item into a temporary register, and any new definitions of the item are immediately stored back to memory. Spilling can have a costly impact on run-time performance.

With PA-RISC 1.1, the register allocator was enhanced to support the additional floating-point registers. These additional floating-point registers have greatly decreased the amount of floating-point spill code in floating-point-intensive applications. The register allocator now has more than twice the number of 64-bit (double-precision) floating-point registers available for allocation purposes (see Fig. 2). Also, the PA-RISC 1.1 architecture now allows either half of a 64-bit register to be used as a 32-bit (single-precision) register, resulting in more than four times the number of single-precision registers that are available in PA-RISC 1.0.

Improved Instruction Scheduling
The instruction scheduler is responsible for reordering the machine-level instructions within straight-line code to minimize stalls in the processor's pipeline and to take advantage of the parallelism between the CPU and the floating-point coprocessor. It is also responsible for attempting to fill pipeline delay slots of branch instructions with a useful instruction. Of course, the instruction scheduler must maintain the correctness of the program when it reorders instructions. The instruction scheduling

* The FORTRAN compiler also uses an optimizing preprocessor on the front end that performs some language dependent optimizations before sending the code to the standard FORTRAN compiler and the shared optimizer back end (see article on page 24).

algorithm used in the PA-RISC compilers is based on the technique described in reference 9.

Until recently, instruction scheduling was done just once after register allocation, immediately before the machine instructions are written to the object file. This one-pass approach suffered because the register allocator may allocate registers to data items in a manner that imposes artificial dependencies. These artificial dependencies can restrict the scheduler from moving instructions around to avoid interlocks (i.e., pipeline stalls).

For example, the LDW (load word) instruction on PA-RISC typically takes two cycles to complete. This means that if the very next instruction following the LDW uses the target register of the LDW, the processor will stall for one cycle (load-use interlock) until the load completes. The instruction scheduler is responsible for reordering instructions to minimize these stalls. The software pipelining article on page 39 describes pipeline stalls in more detail.

If the register allocator allocates the same register to two independent data items, this might impair the reordering operations of the instruction scheduler. For example, if register allocation results in the following code:

```
LDW   0 (0, %r30), %r20    ;load some data into register 20
ADD   %r20, %r21, %r22     ;use register 20
LDW   8 (0, %r30), %r20    ;load some other data into register 20
ADD   %r20, ...            ;use register 20
```

the scheduler cannot move any of the instructions upwards or downwards to prevent load-use interlocks because of the dependencies on register 20. This could lead to a situation in which no useful instruction can be placed between the LDW and the instructions that use register 20.

These artificial dependencies imposed by the register allocator could also limit the instruction scheduler's ability to interleave general register instructions with floating-point instructions. Interleaving is crucial in keeping both the general CPU and the floating-point coprocessor busy and exploiting a limited amount of parallelism.

To improve the effectiveness of instruction scheduling, both the PA-RISC 1.0 and 1.1 compilers now perform instruction scheduling twice, once before register allocation and once after. By scheduling before register allocation, the scheduler can now detect a greater amount of instruction-level parallelism within the code and thus have greater freedom in reordering the instructions. Scheduling after register allocation enables the scheduler to reorder instructions in regions where the register allocation may have deleted or added instructions (i.e., spill code instructions).

The instruction scheduler's dependency analysis capabilities have also been improved to recognize many of the cases where indexed loads and stores are accessing distinct elements of the same array. Through more accurate information, the scheduler has greater freedom to safely move loads of some array elements ahead of stores to other elements of that array.

Another improvement made to help the scheduler when it is ordering code involved tuning the heuristics used to take into account some of the unique features of implementations of PA-RISC 1.1. These heuristics are aimed at avoiding cache stalls (stores immediately followed by loads or other stores), and modeling the floating-point latencies of the new PA-RISC 1.1 implementation more closely.

Finally, the instruction scheduler has also been enhanced to identify cases in which the new five-operand instructions available in PA-RISC 1.1 can be formed. The scheduler, running before register allocation, identifies floating-point multiplication (FMPY) instructions and independent floating-point addition (FADD) or subtraction (FSUB) instructions that can be combined to form a single five-operand FMPYADD or FMPYSUB instruction.

When five-operand instructions are formulated during the scheduler pass, global data flow information is used to ensure that one of the registers used as an operand of the FADD or FSUB can be used to hold the result of the FADD or FSUB. This will be true if the data flow information shows that the register containing the original operand has no further use in the instructions that follow. For example, in the instruction sequence:

```
FMPY fr1, fr2, fr3    ;fr3 = fr1 * fr2
....
....
FADD fr4, fr5, fr6    ;fr6 = fr4 + fr5
```

if fr5 has no further uses in the instructions that follow the FADD, it can be used to replace register fr6 as the result of the addition. Any instructions that follow the FADD that use the result of the addition would be modified to use register fr5 instead of register fr6.

Another problem confronting the instruction scheduler is instructions that occur between two instructions targeted to be joined. For example, take a simple case of using fr3 between the FMPY and the FADD:

```
FMPY fr1, fr2, fr3          ; fr3 = fr1 * fr2, fr5 = fr4 + fr5
FSTDS fr3, memory           ; store fr3 in memory
....
FADD fr4, fr5, fr6          ; fr6 = fr4 + fr5
```

The five-operand FMPYADD cannot be placed in the position of the FADD without moving the use of fr3 below the new five-operand instruction because the wrong value may be stored to memory. When the scheduler is satisfied that the necessary criteria have been met, it will produce the five-operand instruction:

```
....
....
FMPYADD fr1, fr2, fr3, fr4, fr5  ; fr3 = fr1 * fr2, fr5 = fr4 + fr5
FSTDS fr3, memory
```

where register fr5 serves as both an operand and the result of the addition.

These five-operand instructions allow the compiler to reduce significantly the number of instructions generated for some applications. In addition, they allow the floating-point coprocessor to dispatch two operations in a single cycle.

Software Pipelining

Software pipelining is an advanced transformation that attempts to interleave instructions from multiple iterations of a program loop to produce a greater amount of instruction-level parallelism and minimize pipeline stalls. Software pipelining is a new component that has been added to the global optimizer for both PA-RISC 1.0 and PA-RISC 1.1. The article on page 39 provides more information on this loop transformation scheme.

Register Reassociation

Register reassociation is a code improving transformation that supplements loop-invariant code motion and strength reduction. The main objective of this optimization is to eliminate integer arithmetic operations found in loops. It is particularly useful when applied to multidimensional array address computations. The article on page 33 provides more information on this transformation technique.

Linker Optimizations

The `ADDIL` instruction is used by the compilers in conjunction with load or store instructions to generate the virtual addresses of global or static data items. The compiler must produce these large addresses because the compiler has no knowledge of where the desired data will be mapped with regards to the global data base register. The `ADDIL` instruction is unnecessary if the short displacement field in the load or store instruction is adequate to specify the offset of the data from the base register. The actual displacement of data items is finalized at link time. The PA-RISC 1.0 and 1.1 compilers now arrange for small global variables to be allocated as close to the base of the data area as possible. The HP-UX linker has been enhanced to remove any unnecessary `ADDIL` instructions when the short displacement field in load and store instructions is found to be adequate.

As optimization strategies become more sophisticated, the use of run-time profiling data can be very useful in guiding various transformations. In the first of many stages to come, the PA-RISC optimizing linker now uses profiling information to reorder the procedures of an application to reduce cache contention and to minimize the number of dynamic long branches needed to transfer control between heavily called routines. This repositioning technique is currently known as profile-based procedure repositioning.[10]

These two linker-based optimizations are enabled through special compiler and linker options. See "Link-Time Optimizations" on page 22 for more information on these two transformations.

FORTRAN Vectorizing Preprocessor

The Hewlett-Packard FORTRAN optimizing preprocessor is a major addition to the FORTRAN compiler for the HP 9000 Series 700 implementation of PA-RISC 1.1. The preprocessor was a joint effort of Hewlett-Packard and an outside vendor. Using advanced program and data flow analysis techniques, and with specific details covering the implementation of the underlying architecture, FORTRAN source code is transformed to be more efficient and to take advantage of a highly tuned vector library. The preprocessor has boosted benchmark performance and real customer application performance by as much as 30%. The Series 700 FORTRAN optimizing preprocessor is described in the article on page 24.

Compatibility

An important design goal in evolving the architecture to PA-RISC 1.1 was to allow a smooth transition from existing PA-RISC 1.0 implementations. With the exception of FORTRAN, the compilers on the Series 700 implementation of PA-RISC 1.1 are based on the compilers used in the existing Series 800 implementations of PA-RISC 1.0. Because the same compilers are used on Series 800 and 700 systems, maximum portability of source code is achieved.

Another system design goal was to provide software compatibility at the source code level with the HP 9000 Series 300 and Series 400 workstations, which are based on the Motorola MC680x0 architecture.

Special efforts have been made for the C and FORTRAN languages to provide this compatibility. The PA-RISC C compiler has been enhanced with compiler directives to provide Series 300 and 400 compatible data alignment, which is the one area of potential incompatibility with PA-RISC. In the case of FORTRAN, a larger number of compatibility issues exist. The first release of system software included a version of the FORTRAN compiler from the Series 800 and a separate version from the Series 300 and 400 workstations. The latest releases now contain a single FORTRAN compiler based on the Series 300 and 400 workstation compiler that has an option that allows users to compile their FORTRAN applications with semantics identical to either the Series 800 compiler or the Series 300 compiler.

Given that the PA-RISC 1.1 architecture is a strict superset of PA-RISC 1.0, all HP-UX object code is completely forward compatible from PA-RISC 1.0 based implementations to the new PA-RISC 1.1 workstations. Portability includes object modules, libraries, and relocatable programs. Programs compiled and linked on PA-RISC 1.0 implementations can run unchanged on PA-RISC 1.1 implementations, and any combination of object modules and libraries from the two systems can be linked together. Recompilation is necessary only if the programmer wishes to take advantage of the architecture and optimization enhancements. This forward compatibility of object modules allowed many vendors to port their products to the Series 700 with little or no effort.

Although object files can be ported from PA-RISC 1.0 implementations to PA-RISC 1.1 implementations, the reverse may not always be true if the object file on a PA-RISC 1.1 machine was generated by a compiler that exploits the extensions to the PA-RISC 1.1 architecture. The HP-UX loader detects such situations and refuses to execute a program on a PA-RISC 1.0 implementation that has been compiled with PA-RISC 1.1 extensions. To assist the user in generating the most portable object files, a compiler option has been added to specify the destination architecture (DA) for the code generated by the compiler. For example,

```
% cc +DA1.0 my_prog.c
```

would generate an object file based on the PA-RISC 1.0 architecture definition. The object file could also be ported directly to a PA-RISC 1.1 implementation without the need for recompilation. A user can also use this option explicitly to ask for PA-RISC 1.1 extensions, or for cross compiling while on a PA-RISC 1.0 implementation with the command-line sequence:

```
% cc +DA1.1 my_prog.c
```

Of course, the object file produced could no longer be executed on a PA-RISC 1.0 implementation.

If the destination architecture is not specified, the default for the compilers is to generate code based on the architecture implementation on which the compiler is executing.

Performance

Through a combination of clock rate, instruction set extensions, compiler optimization enhancements, and processor implementation, the HP 9000 Series 700 workstations are currently producing industry leading performance. Although much of this performance improvement comes from an increase in clock rate, as seen in the tables below, the compilers play a significant role in increasing the overall performance.

Table I compares the raw speed of the HP 9000 Series 720 workstation based on PA-RISC 1.1 architecture with the HP 9000 Series 835 workstation based on PA-RISC 1.0. The SPEC benchmarks for the Series 835 were compiled with the HP-UX 7.0 compilers using full optimization. For the Series 720, the HP-UX 8.07 compilers containing the latest enhancements to support PA-RISC 1.1 were used. For the SPEC benchmark suite, higher SPECmarks imply higher throughput.

Table I
Performance Comparison of PA-RISC Implementations

Processor/ Implementation	Clock MHz	Cache Size in K bytes	SPECmarks		
			Integer	Float	Overall
Model 835/ PA-RISC 1.0	15	128/ 128*	9.7	9.1	9.4
Model 720/ PA-RISC 1.1	50	256/ 256*	39.5	80.0	60.4

*Instruction Cache/Data Cache

The data in Table II compares the relative efficiency of the HP 9000 Series 835 and the HP 9000 Series 720 by normalizing the benchmark performance. To normalize the numbers, the performance numbers are divided by the clock frequency. The normalized SPECmark performance of the Model 720 is 92% higher than the normalized performance of the Model 835. Floating-point performance, which is 162% higher, is primarily because of the optimizing preprocessor, better compiler optimizations, architecture extensions, implementation of separate floating-point multiplication and arithmetic functional units, faster floating-point operations, and larger caches. The gains in the integer SPEC benchmark (22%) are primarily because of enhancements to traditional optimizations that are architecture independent.

Table II
Normalized Performance Comparison of PA-RISC Implementations

Processor/ Implementation	Clock MHz	Normalized Performance			Improvement over Series 835		
		Integer	Float	Overall	Integer	Float	Overall
Model 835/ PA-RISC 1.0	15	0.65	0.61	0.63	1.00	1.00	1.00
Model 720/ PA-RISC 1.1	50	0.79	1.60	1.21	1.22	2.62	1.92

To see exactly how much performance was gained through enhancements to the traditional compiler optimizations (not architecture-specific), we compiled the SPEC benchmarks using the HP-UX 8.07 compilers with level 2 optimization and the destination architecture PA-RISC 1.0. This disables the use of the added instructions and floating-point registers. We also disabled use of the FORTRAN optimizing preprocessor. Table III shows how the HP-UX 7.0 SPEC benchmarks compare to the HP-UX 8.05 benchmarks while running on an HP 9000 Model 720.

From Table III, we can see that the enhancements made to the traditional compiler optimizations performed by the compilers produced gains of 1 to 24 percent.

It is also interesting to see how much the architecture itself contributed to performance improvement. To do this, we used the same HP-UX 8.05 compilers (with the -0 option, which indicates to compile without the FORTRAN optimizing preprocessor) to produce SPEC benchmarks compiled for PA-RISC 1.0 and PA-RISC 1.1. Table IV shows that all floating-point benchmarks except Spice show a significant improvement. This improvement comes directly from the larger register file and the added instructions in the PA-RISC 1.1 instruction set. The integer SPEC benchmarks are absent from this table because the architecture enhancements do little for integer code.

Table III
Comparison between Benchmarks Compiled With HP-UX 7.0 and HP-UX 8.07 Compilers Running on an HP 9000 Model 720 Workstation

Benchmarks	HP-UX 7.0	HP-UX 8.05	Compiler Improvement
001.gcc1.35	31.2	35.0	1.12
008.espresso	37.4	43.7	1.17
022.li	38.1	38.6	1.01
023.eqntott	36.8	41.2	1.12
012.spice2g6	37.1	46.2	1.24
015.doduc	37.6	44.2	1.18
020.nasa7	36.8	42.3	1.15
030.matrix300	25.8	27.0	1.05
042.fpppp	52.5	59.6	1.14
047.tomcatv	38.8	40.8	1.05
SPECint	35.8	39.5	1.10
SPECfp	37.3	42.3	1.13
SPECmark	36.7	41.1	1.12

Table IV
Performance Improvement Resulting from Architecture Enhancements on the HP 9000 Model 720 Workstation

Benchmark	PA-RISC 1.0	PA-RISC 1.1	Architecture Improvement
012.spice 2g6	46.2	46.1	1.00
015.doduc	44.2	50.6	1.14
020.nasa7	42.3	44.2	1.04
030.matrix300	27.0	28.2	1.04
042.fpppp	59.6	82.8	1.39
047.tomcatv	40.8	50.4	1.24
SPECfp	42.3	47.8	1.13
SPECmark	41.1	44.4	1.08

Finally, we wanted to see how much the optimizing preprocessor contributed to the SPEC benchmark improvement. To do this, we used the HP-UX 8.05 FORTRAN compiler to produce two sets of the FORTRAN SPEC benchmarks. Both sets were compiled with full optmization but only one was compiled with full optimization and the addition of the preprocessor. While benchmarks nasa7 and tomcatv showed fairly large improvements with the optimizing preprocessor, the gains for matrix300 were dramatic. All these benchmarks are known to suffer from cache and TLB (translation lookaside buffer) miss penalties, but the preprocessor was able to improve their performance through its memory hierarchy optimizations. Table V shows a comparison between the benchmarks created on an HP-UX 8.05 operating system running on an HP 9000 Model 720 and compiled with and without the optimizing preprocessor enhancements. Excluded benchmarks showed little or no gain. See the article on page 24 for more about the FORTRAN optimizing preprocessor.

Table V
Performance Gains With the FORTRTAN Optimizing Preprocessor

Benchmarks	Without Preprocessor	With Preprocessor	Improvement
020.nasa7	44.2	62.9	1.42
030.matrix300	28.2	320.9	11.38
047.tomcatv	50.4	67.1	1.33
SPECfp	47.8	80.0	1.67
SPECmark	44.4	60.4	1.36

Conclusions

To remain competitive in the workstation market, the PA-RISC architecture has been extended to better meet the performance demands of workstation applications. With these changes to the architecture, Hewlett-Packard's compiler products have evolved to exploit the extensions made. Most important, the compilers successfully exploit the increase in the number of floating-point register files and the new instructions including the integer multiply and the five-operand instructions.

Besides being enhanced to exploit these new architectural features, additional code improving transformations have been introduced that are independent of the underlying architecture and substantially boost the performance of applications. These include a new vectorizing preprocessor for FORTRAN, software pipelining, register reassociation, link-time optimizations, and better instruction scheduling. The combined result of the architecture extensions, compiler enhancements, and a high-speed CMOS processor implementation is a workstation system that compares favorably with the most advanced workstations presently available.

Acknowledgements

There are too many names to list whose ideas and contributions have helped produce an industry leading workstation offering with highly tuned optimizing compilers. This includes all of the members of the HP California Language Laboratory optimization technology section, the HP Apollo Massachusetts Language Laboratory, and the HP Colorado Language Laboratory who shared many optimization ideas with us.

I would like to extend special thanks to Manoj Dadoo for his help in providing the latest SPEC benchmark results.

References

1. J. S. Birnbaum and W. S. Worley, Jr., "Beyond RISC: High-Precision Architecture," *Hewlett-Packard Journal*, Vol. 36, no. 8, August 1985.
2. M. Mahon, et al, "Hewlett-Packard Precision Architecture: The Processor," *Hewlett-Packard Journal*, Vol. 37, no. 8, August 1986, pp. 4-21.
3. Ruby B. Lee, "Precision Architecture," *IEEE Computer*, Vol. 22, January 1989, pp. 78-91.

(continued on page 23)

Link-Time Optimizations

There are some optimizations that can be performed only when the linker produces an executable file. For the PA-RISC systems these optimizations include removing unnecessary instructions by changing the location of certain data segments, and locating procedures that call each other frequently close together.

Elimination of Unnecessary ADDIL Instructions

Compilers generally do not know whether their data will be close to the base register for the data segment. Therefore, references to global or static variables on PA-RISC machines require two instructions to form the address of a variable or to load (or store) the contents of the variable. For example the instructions:

```
ADDIL LR'var-$global$,dp
LDW  RR'var-$global$(r1),r10
```

load the contents of contents of a global variable into register 10.

The ADDIL instruction constructs the left side of the 32-bit virtual address. In most cases, however, the data is within reach of the load or store instructions, and an unnecessary ADDIL instruction is present in the code. Since ADDILs account for about 2% of the generated code, significant run-time savings result from their removal.

If the location for the variable turns out to be close to the global data pointer dp, then the offset of the ADDIL is zero and the ADDIL is like a COPY of global base register 27 (the location of dp) to register 1. In such a case, it is more efficient to eliminate the ADDIL and use register 27 as the base register in the LDW instruction. This elimination can be performed at link time once the linker lays out all the global data and computes the value that will be assigned to dp.

The –O linker option turns on linker optimizations. Link-time optimizations include removing the unnecessary ADDILs. Data is also rearranged to increase the number of data items that can be accessed without ADDILs. The –O option is passed to the linker by the compilers when the +O3 compiler option is selected. The +O3 option also advises the compiler not to optimize by moving ADDILs out of loops, in the expectation that they will be removed at link time. This can be very effective in reducing register pressure for some procedures. For example, to optimize a C program at link time as well as compile time, use cc +O3 foo.c.

Because shared libraries on HP-UX use position independent code that is referenced from register 19 as a base register, ADDIL elimination is not done when building an HP-UX shared library. It is also in conflict with the –A (dynamic linking) option, the –r (relocatable link) option, and the –g (symbolic debugging) option. All conflicts are resolved by disabling this optimization. Shared libraries and position independent code are described on page 46.

The linker rearranges data to maximize the number of variables that can be placed at the beginning of the data area, increasing the probability that ADDILs referencing these variables can be removed. Nonstandard, conforming programs that rely on specific positioning of global or static variables may not work correctly after this optimization.

ADDIL elimination is appropriate for programs that access global or static variables frequently. Programs not doing so may not show noticeable improvement. Link-time optimization increases linking time significantly (approximately 20%) because of the additional processing and memory required.

Profile-Based Procedure Repositioning at Link Time

Research has consistently shown that programs tend to keep using the instructions and data that were used recently. One of the corollaries of this principle is that programs have large amounts of code (and to a lesser extent data) that is used to handle things that very seldom happen, and therefore are only in the way when running normal cases.

This observation is exploited by a new optimization in the HP-UX 8.05 linkers called profile-based procedure repositioning (sometimes referred to as feedback-directed positioning).[1] This three-step optimization first instruments the program to count how often procedures call each other at run time. The instrumented program is run on sample input data to collect a profile of the calls executed by the program. The linker then uses that profile information in the final link of the production program to place procedures that call each other frequently close together.

A more important case is the inverse—things that are infrequently or never called are grouped together far away from the heavily used code. This increases instruction-cache locality and in large applications decreases paging, since only the code that will be used is demand-loaded into main memory or cache, not a mixture of useful and unneeded code that happens to be allocated to the same page or cache line.

This optimization is invoked by two new linker options:

- –I: Instrument the code to collect procedure call counts during execution. This option is used in conjunction with the –P option.
- –P: Examine the data file produced with the –I option and reposition the procedures according to a "closest is best" strategy.

These options are often passed to the linker via the compiler driver program's –W option. For instance, a C program can be optimized with profile-driven procedure positioning by:

```
cc –c –O foo.c          # compile with optimizations
cc –Wl,–I foo.o         # link with profiling instrumentation code added
a.out < data.in         # run program to generate profile information
                        # in the "flow.data" file
cc –Wl,–P foo.o         # link with procedures positioned according to
                        # profile
```

The first link of the executable produces an executable with extra code added to produce a file of profile information with counts of all the calls between each pair of procedures executed. The final link uses the profile data file information to determine the order of procedures in the final executable file, overriding the normal positioning by the order of the input files seen. This order will optimize use of the virtual memory system for the program's code segment. A secondary effect is to reduce the number of long branch stubs (code inserted to complete calls longer than 256K bytes). While the total number of long branch stubs may actually increase, the number of long branches executed at run time will decrease.

Carl Burch
Software Engineer
California Language Laboratory

Reference

1. *Programming on HP-UX*, The HP-UX 8.05 Technical Addendum, HP part number B2355A, option OBF.

4. D.J. Magenheimer, L. Peters, K.W. Pettis, and D. Zuras, "Integer Multiplication and Division on the HP Precision Architecture," *IEEE Transactions on Computers*, Vol. 37, August 1988, pp. 980-990.
5. Mark S. Johnson and Terrence C. Miller, "Effectiveness of a Machine-Level, Global Optimizer," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices*, Vol. 20, no. 7, July 1986.
6. K. W. Pettis and W. B. Buzbee, "Hewlett-Packard Precision Architecture Compiler Performance," *Hewlett-Packard Journal*, Vol. 38, no. 3, March 1987.
7. D. S. Coutant, C. L. Hammond, and J. W. Kelly, "Compilers for the New Generation of Hewlett-Packard Computers," *Hewlett-Packard Journal*, Vol. 37, no. 1, January 1986.
8. G. J. Chaitin, "Register Allocation and Spilling via Graph Coloring," *Proceedings of the SIGPLAN '82 Symposium On Compiler Construction, SIGPLAN Notices*, Vol. 17, no. 6, June 1982, pp. 98-105.
9. P. Gibbons and S. Muchnick, "Efficient Instruction Scheduling for a Pipelined Architecture," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices*, Vol. 20, no. 7, July 1986.
10. K. Pettis and R.C. Hansen, "Profile Guided Code Positioning," *Proceedings of the SIGPLAN '90 Symposium on Programming Language Design and Implementation, SIGPLAN Notices*, Vol. 25, no. 6, June 1990.

Bibliography

1. M. Forsyth, S. Mangelsdorf, E. DeLano, C. Gleason, and D. Steiss, "CMOS PA-RISC Processor for a New Family of Workstations," *IEEE COMPCON Spring '91 Digest of Papers*, February 1991.
2. R. Horning, L. Johnson, L. Thayer, D. Li, V. Meier, C. Dowdell, and D. Roberts, "System Design for a Low Cost PA-RISC Desktop Workstation," *IEEE COMPCON Spring '91 Digest of Papers*, February 1991.
3. D. Odnert, R. Hansen, M. Dadoo, and M. Laventhal, "Architecture and Compiler Enhancements for PA-RISC Workstations," *IEEE COMPCON Spring '91 Digest of Papers*, February 1991.

HP 9000 Series 700 FORTRAN Optimizing Preprocessor

By combining HP design engineering and quality assurance capabilities with a well-established third party product, the performance of Series 700 FORTRAN programs, as measured by key workstation benchmarks, was improved by more than 30%.

by Robert A. Gottlieb, Daniel J. Magenheimer, Sue A. Meloy, and Alan C. Meyer

An optimizing preprocessor is responsible for modifying source code in a way that allows an optimizing compiler to produce object code that makes the best use of the architecture of the target machine. The executable code resulting from this optimization process is able to make efficient use of storage and execute in minimum time.

The HP 9000 Series 700 FORTRAN optimizing preprocessor uses advanced program and data flow analysis techniques and a keen understanding of the underlying machine implementation to transform FORTRAN source code into code that is more efficient and makes calls to a highly tuned vector library. The vector library is a group of routines written in assembly language that are tuned to run very fast (see "Vector Library" on page 29). Fig. 1 shows the data flow involved in using the optimizing preprocessor to transform FORTRAN source code into an optimized executable file.

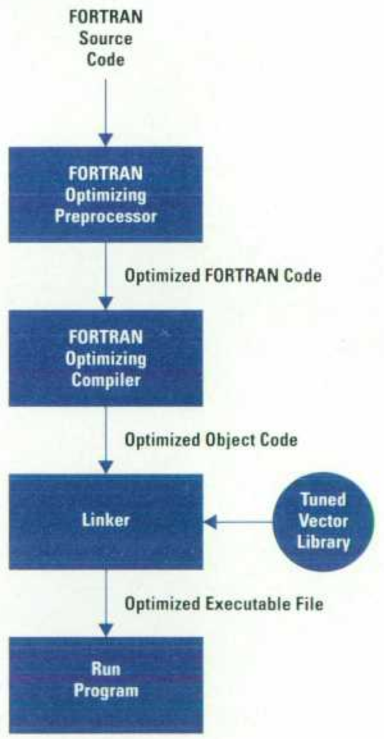


Fig. 1. Data flow for compiling FORTRAN source code using the optimizing preprocessor.

A slightly different version of this product serves as the preprocessor for HP Concurrent FORTRAN, which is now running on HP Apollo DN10000 computers. HP Apollo engineers responsible for this product identified opportunities for substantial improvements to the preprocessor and concluded that these improvements were also applicable to the Series 700 FORTRAN. Performance analysis confirmed these conclusions, and after marketing analysis, an extended multisite, cross-functional team was formed to incorporate the preprocessor into the FORTRAN compiler for the HP 9000 Series 700 computer systems. Because of this effort, as of the HP-UX* 8.05 release, the preprocessor is bundled with every FORTRAN compiler.

The preprocessor is based on a third-party product. HP's contribution included:

- Tying the preprocessor into the HP FORTRAN product (This included user interface changes and extensive documentation changes.)
- Identifying modifications required to allow the preprocessor to recognize HP's extended FORTRAN dialect
- Assembly coding a vector library that incorporates knowledge of CPU pipelining details and implementation dependent instructions to allow the Series 700 to work at peak performance
- Performing extensive quality assurance processes that uncovered numerous defects, ensuring that the product meets HP's high-quality standards.

These contributions are discussed in detail in this article. Examples of specific transformations and performance improvements on key industry benchmarks are also described.

Preprocessor Overview

Although the preprocessor is bundled with every Series 700 FORTRAN compiler as of the HP-UX 8.05 release, the preprocessor is not automatically enabled whenever a user compiles a FORTRAN program. To invoke the preprocessor, the `+OP` option must be specified on the command line invoking the FORTRAN compiler. For example,

```
f77 +OP file.f
```

will cause the file file.f to be preprocessed and then compiled by the FORTRAN compiler. In addition, an integer between 0 and 4 can be appended following +OP. This integer selects the settings of certain preprocessor options. For example, to make the preprocessor optimize as aggressively as possible, the following could be used:

```
f77 +OP4 file.f
```

By default, the +OP option also automatically invokes the standard optimizer at the optimization level defined by the -O option, which typically indicates full optimization (-O2).

Advanced Options. The preprocessor can be invoked with many options by using the -WP option. For example,

```
f77 +OP -WP, -novectorize file.f
```

precludes the preprocessor from generating calls to the vector library. Some other classes of options include:

- Inlining Options. These options instruct the preprocessor to replace subroutine or function calls with the actual text of the subroutine or function. This removes the overhead of a procedure call and exposes additional opportunities for optimizations. These options allow the user not only to instruct the preprocessor whether or not to inline, but also to provide the maximum level of subprogram nesting and lists of files to examine for inlining. The user can exercise manual control over inlining with directives, and impose restrictions on inlining in nested loops.
- Optimization Options. Using optimization options, the user can adjust parameters that control loop unrolling, transformations that may affect arithmetic roundoff, and the aggressiveness of the optimizations that are attempted.
- Vectorization Options. These options tell the preprocessor whether or not to generate calls to the vector library and adjust the minimum vector length that will cause such a call to be generated.
- Listing Options. The user can obtain detailed information about the program and the optimizations performed by the preprocessor with listing options. Also, the user can adjust the format and level of detail of the information in the listings.
- Other Options. Some options specify whether certain directives (described below) are to be recognized by the preprocessor and what global assumptions can be made about the behavior of the user program. There are also options that allow the user to designate special inline comment characters to be recognized and whether to save program variables in static memory.

Directives. The preprocessor provides an extensive set of directives. These directives can be inserted directly in the FORTRAN application and appear to the compiler as comments except when enabled by certain command-line options. Placement of these directives in the code allows the user to vary control of the optimizations performed by the preprocessor in each subprogram. This control can have the granularity of a single line in a subprogram.

Some of the features provided by directives include:

- Optimization Control. Optimization directives provide control of inlining, roundoff, and optimization aggressiveness.
- Vector Call Control. Vector call translation directives control substitutions that result in calls to the vector library from the preprocessor.
- Compatibility. Certain directive formats used by competitive products are recognized to allow correct optimizations to be performed on supercomputer applications.
- Assertions. Assertions can be inserted in an application to allow the user to provide additional program information that will allow the preprocessor to make informed decisions about enabling or disabling certain optimizations. For example, many FORTRAN applications violate array subscript bounds. If the user does not inform the preprocessor of this language standard violation, transformations may be performed that result in incorrect execution of the program.

Transformations

The HP FORTRAN optimizing preprocessor supports a number of different transformations (changes to the source code) that are intended to improve the performance of the code. These transformations include the following categories:

- Scalar transformations
- Interprocedural transformations
- Vector transformations
- Data locality (blocking) and memory access transformations.

Scalar Transformations. Many of these transformations are "enabling" optimizations. That is, they are necessary to expose or enable opportunities for the other optimizations. Some of these transformations include:

- Loop Unrolling. This transformation attempts to compress together several iterations of a loop, with the intent of lowering the cost of the loop overhead and exposing more opportunity for more efficiently using the functional units of the PA-RISC architecture. The article on page 39 provides some examples of loop unrolling.
- Loop Rerolling. This transformation is the exact opposite of loop unrolling in that it is used when a loop has been explicitly unrolled by the user. The transformation recognizes that the code has been unrolled, and rerolls it into a smaller loop. This may be beneficial in cases where the code can be transformed to a call to the vector library.
- Dead Code Elimination. This transformation removes code that cannot be executed. This can improve performance by revealing opportunities for other transformations.
- Forward Substitution. The preprocessor replaces references to variables with the appropriate constants or expressions to expose opportunities for additional transformations.
- Induction Variable Analysis. The preprocessor recognizes variables that are incremented by a loop-invariant amount within a loop, and may replace expressions using one induction variable with an expression based on another induction variable. For example, in the following code fragment the preprocessor identifies that K is an induction variable:

```
DO I = 1, N
  A(I) = B(K)
  K = K - 1
ENDDO
```

The code generated by the preprocessor would be:

```
DO I = 1,N
  A(I) = B(K-I+1)
ENDDO
```

- Lifetime Analysis. The preprocessor analyzes the use of variables within a routine, and determines when the value of a variable can be discarded because it will not be needed again.

Interprocedural Transformations. The preprocessor is capable of performing subroutine and function inline substitution. This optimization allows the preprocessor, either by explicit user control or heuristically, to replace a call to a routine with the code in the routine. This transformation improves performance by:

- Reducing call overhead, which is useful for very small routines
- Replacing expressions in inlined subroutines or functions with constants because some arguments to these routines might be constants
- Exposing other performance improvement opportunities such as data locality.

Vector Transformations. The preprocessor replaces code sequences with calls to the vector library where appropriate. Some classes of these calls include:

- Loop Vectorization. This refers to cases in which the user's code refers to one or several sequences of inputs, producing a sequence of outputs. These sequences would be references to arrays. For example,

```
   DO 10 I = 1, N
10 A(I) = B(I) + C(I)
```

would become:

```
CALL vec_$dadd_vector(B(1),C(1),N,A(1))
```

Not all seemingly appropriate places would be vectorized because in some cases multiple references to the same subscripted variable might be more efficiently done by inline code rather than by a call to a vector library routine.

- Reduction Recognition. The preprocessor will recognize some cases in which the results are accumulated for use as an aggregate, such as in summing all the elements in an array or finding the maximum value in an array. For example,

```
   DO 10 I = 1, N
10 X = X + A(I) * B(I)
```

would become:

```
X = X + vec_$ddot(A(1),B(1),N)
```

This transform improves performance in part by knowing that while a Series 700 computer can add one stream of numbers in three machine cycles per element, it can also add two streams of numbers in four machine cycles per two elements.

There is one problem with this transform. When using two streams to compute the result (which is what the routine does) in floating-point calculations, changing the order in which numbers are added can change the result. This is called roundoff error. Because of this problem, the reduction recognition transformation can be inhibited by using the roundoff switch.

- Linear Recurrence Recognition. This transformation is used in cases in which the results of a previous iteration of a loop are used in the current iteration. This is called a recurrence.
Example:

```
DO 10 I = 2, N
10 A(I) = B (I) + C*A(I-1)
```

In this case the Ith element of A is dependent on the result of the calculation of the (I-1)th element of A. This code becomes:

```
CALL vec_$rec1cr(B(2),N-1,C,A(1))
```

Data Locality and Memory Access Transformations. Memory side effects such as cache misses can have a significant impact on the performance of the Series 700 machine. As a result, a number of transformations have been developed to reduce the likelihood of cache misses and other problems.

- Stride-1 Inner Loop Selection. This transformation examines a nested series of loops, and attempts to determine if the loops can be rearranged so that a different loop can run as the inner loop. This is done if the new inner loop will have more sequential walks of arrays through memory. This type of access is advantageous because it reduces cache misses. For example,

```
DO 10 I = 1, N
   DO 10 J = 1, N
10 A(I,J) = B(I,J) + C(I,J)
```

accesses the arrays A, B, and C. However, it accesses them in the sequences:

```
A(1,1), A(1,2), A(1,3), ...
B(1,1), B(1,2), B(1,3), ...
C(1,1), C(1,2), C(1,3), ...
```

which will result in nonsequential access to memory, making cache misses far more likely. The following legal transform will reduce the likely number of cache misses.

```
DO 10 J = 1, N          } These loops have
  DO 10 I = 1, N        } been exchanged
10   A(I,J) = B(I,J) + C(I,J)
```

- Data Locality Transformations. For situations in which there is significant reuse of array data, and there is opportunity to restructure, or "block," the code to reduce cache misses, the preprocessor will create multiple nested loops that will localize the data in the cache at a cost of more loop overhead.
- Matrix Multiply Recognition. The preprocessor will recognize many forms of classic matrix multiply and replace them with calls to a highly tuned matrix multiply routine.

Example of a Transformation. The following code fragments taken from the Matrix300 benchmark of the SPEC benchmark tests show how some of the transformations described above are incorporated into a FORTRAN program.

```
      REAL*8 A(300,300), B(301,301), C(302,302)
      DATA M, N, L /300,300,300/
      IA = 300
      IB = 301
      IC = 302
      CALL SGEMM(M, N, L, A, IA, B, IB, C, IC, 0, +1)
      END

      SUBROUTINE SGEMM(M, N, L, A, IA, B, IB, C, IC, JTRPOS, JOB)
      REAL*8 A(IA,N), B(IB,L), C(IC,L)
      JB = ISIGN(IABS(JOB)+2*(JTRPOS/4),JOB)
      JUMP = JTRPOS + 1
      GO TO (10, 30), JUMP
10    CONTINUE
      DO 20 J = 1, L
20      CALL SGEMV(M, N, A, IA, B(1,J), 1, C(1,J), 1, JB)
      RETURN
30    CONTINUE
      DO 40 J = 1, L
40      CALL SGEMV(M, N, A, IA, B(1,J), 1, C(J,1), IC, JB)
      RETURN
      END

      SUBROUTINE SGEMV(M, N, A, IA, X, IX, Y, IY, JOB)
      REAL*8 A(IA,N), X(IX,N), Y(IY,N)
      IF (N.LE.0) RETURN
      II = 1  IJ = IA
      IF (((IABS(JOB)-1)/2).EQ.0) GO TO 210
      IJ = 1
      II = IA
210   CONTINUE
      IF (MOD(IABS(JOB)-1,2).NE.0) GO TO 230
      DO 220 J = 1, M
220     Y(1,J) = 0.0D0
230   CONTINUE
      IF (JOB.LT.0) GO TO 250
      DO 240 J = 1, N
        K = 1 + (J-1)*IJ
240     CALL SAXPY(M, X(1,J), A(K,1), II, Y(1,1), IY)
      RETURN
250   CONTINUE
      DO 260 J = 1, N
        L = 1 + (J-1)*IJ
260     CALL SAXPY(M, -X(1,J), A(L,1), II, Y(1,1), IY)
      RETURN
      END

      SUBROUTINE SAXPY(N, A, X, INCX, Y, INCY)
      REAL X(INCX,N), Y(INCY,N), A
      IF (N.LE.0) RETURN
      DO 310 I = 1, N
310     Y(1,I) = Y(1,I) + A*X(1,I)
      RETURN
      END
```

First, routine SGEMM is inlined into the main routine, and the scalar forward substitution transformation is applied to propagate arguments.

```
      REAL*8 A(300,300), B(301,301), C(302,302)
      DATA M, N, L /300,300,300/
      IA = 300
      IB = 301
      IC = 302
      JB = 1
      JUMP = 1
      GO TO (10, 30), 1
10    CONTINUE
      DO 20 J = 1, L
20      CALL SGEMV(M, N, A, IA, B(1,J), 1, C(1,J), 1, JB)
      RETURN
30    CONTINUE
      DO 40 J = 1, L
40      CALL SGEMV(M, N, A, IA, B(1,J), 1, C(J,1), IC, JB)
      RETURN
      END
```

Second, dead code elimination is applied. The computed GO TO turns into a simple GO TO, and the unreachable code is removed.

```
      REAL*8 A(300,300), B(301,301), C(302,302)
      DATA M, N, L /300,300,300/
      IA = 300
      IB = 301
      IC = 302
      JB = 1
      JUMP = 1
      DO I = 1, L
        CALL SGEMV(M,N,A,IA,B(1,I),1,C(1,I),1,JB)
        END DO
      RETURN
      END
```

Next, lifetime analysis is applied to the code, and it is seen that with the current code configuration the variables L, IB, IC, and JUMP are never modified after the initial assignment.

```
      REAL*8 A(300,300), B(301,301), C(302,302)
      DATA M, N /300,300/
      IA = 300
      JB = 1
      DO I = 1, 300
        CALL SGEMV(M,N,A,IA,B(1,I),1,C(1,I),1,JB)
        END DO
      END
```

Notice that the large body of conditional code has been removed. This is significant as far as the capability to perform further optimizations is concerned. The reason that M, N, and IA were not replaced with the value 300 is that at this point it is not known that the corresponding arguments to SGEMV are not modified.

Next, the routine SGEMV is inlined, and once again, a number of transformations are applied:

```
      REAL*8 A(300,300), B(301,301), C(302,302)
      DATA M/300/
      DO I = 1, 300
        I3 = 1
        DO J = 1, M
          C(J,I) = 0.0D0
          END DO
        DO J = 1, 300
          CALL SAXPY(M,B(J,I),A(1,J),I3,C(1,I),1)
          END DO
        END DO
      END
```
Now, we inline SAXPY to get:

```
      REAL*8 A(300,300), B(301,301), C(302,302)
      DO I = 1, 300
        DO J = 1, 300
          C(J,I) = 0.D0
          END DO
        DO J = 1, 300
          DO K = 1, 300
            C(K,I) = C(K,I) + B(J,I) * A(K,J)
            END DO
          END DO
        END DO
```

Finally, we see that this is a matrix multiply and transform it into a call to a fast matrix multiply routine:

```
      CALL BLAS_$DGEMM('N','N',300,300,300,1.D0,
     X  A(1,1),300,B(1,1),301,0.D0,C(1,1),302)
      END
```

This set of transformations results in an 11× performance improvement because of the ability to transform the original code to a form that can use blocking efficiently via a coded matrix multiply routine.

Matching the HP FORTRAN Dialect

Although a primary motivation for using the preprocessor was the significant performance gains, it was also very important for the preprocessor to work as an integrated component of the FORTRAN compile path. One key aspect to this integration was for the preprocessor to recognize and correctly process the dialect extensions supported by the HP Series 700 FORTRAN compiler.

Three dialect areas were addressed: language extensions, compiler directives, and command-line options. For each of these areas, there were some items that the preprocessor could just ignore, while others required certain actions. Another aspect of the dialect issue is that the transformed FORTRAN code generated by the preprocessor must conform to HP's FORTRAN dialect.

The first task was to define the list of HP dialect extensions the preprocessor had to recognize. The initial pass at this was done by gathering all known extensions in HP FORTRAN including the military extensions (MIL-STD-1753), VAX FORTRAN 4.0 features, Apollo Domain DN10000 features, and other HP extensions. This list was given to the third-party developers as a starting point for implementing HP dialect recognition in the preprocessor.

The next step in defining the dialect extensions was to push the preprocessor through our extensive FORTRAN test suites. These suites contain over 8500 tests, ranging from very simple programs to large FORTRAN applications. The method we used was to run each positive test (no expected failure) with the preprocessor, and compare the results with the expected answers. In this manner, we were able to collect additional dialect items that needed to be added to the preprocessor. The final set of dialect items came as we entered a beta program later in the release, exposing the preprocessor to sets of customer codes.

There were a large number of language extensions the preprocessor did not originally recognize, but they were generally relatively minor features. One example is the ON statement, an HP extension that allows specification of exception handling. The preprocessor merely had to recognize the syntax of this statement and echo it back to the transformed file. Another example was allowing octal and hexadecimal constants to appear as actual arguments to a statement function.

The HP compiler directives also needed to be recognized, sometimes requiring semantic actions from the preprocessor. As an example, consider the code segment:

```
      INTEGER A(10), B(10), C(10), D
      DO I = 1, 10
        A(I) = B(I) * C(I) + D
      ENDDO
```
The preprocessor will transform this to the following vector call:

```
      CALL vec_$imult_add_constant (B(1),C(1),10,D,A(1))
```

However, if the $SHORT directive is present in the file, the preprocessor will instead generate a call to the short integer version of this vector routine:

```
      CALL vec_$imult_add_constant16 (B(1),C(1),10,D,A(1))
```

Most of the directives, such as $WARNINGS, are ignored by the preprocessor.

There were also a number of FORTRAN command-line options that the preprocessor needed to be aware of. For example, the -I2 option specifies that short integers will be the default, which should cause the same effect as the $SHORT directive in the example above. For each of these options, the information was passed via preprocessor command-line options. In the case of the -I2 option, the FORTRAN driver will invoke the preprocessor with the -int=2 option.

Another interesting command-line option is +DA1.0, which indicates that the resulting executable program can be run on a PA-RISC 1.0 machine. Since the vector library contains PA-RISC 1.1-specific instructions, the preprocessor is informed that no vector calls should be generated in the transformed source by passing it the -novectorize flag.

In addition to having the preprocessor recognize the HP Series 700 FORTRAN dialect, there was a need to ensure that the resulting transformed source from the preprocessor would be acceptable to the standard FORTRAN compiler. This situation actually occurred in several different cases. In one case, the preprocessor generated FORTRAN source code in which DATA statements appear amid executable statements, something the compiler allows only if the -K command-line option is present. The solution was to have the preprocessor change the order in which it emits the DATA statements.

(continued on page 30)

Vector Library

The vector library is a collection of routines written in assembly language that are tuned to achieve optimal performance in a PA-RISC 1.1 machine.

At the time we decided to port the preprocessor to the HP-UX operating system, the HP Concurrent FORTRAN compiler team had already been working with the third party to generate calls to the vector library available in the Domain operating system. To have a single interface for both products, we decided to provide the Domain library interface on HP-UX.

The Domain library consists of 57 basic functions, with variations for different types and variable or unit stride,* for a total of 380 routines. However, not all of these routines are generated by the preprocessor. Table I lists some of the 39 basic routines that are generated by the Series 700 preprocessor.

Table I
Some of the Basic Vector Library Routines

Routines	Operations
	Unary
vec_$abs(U, Count, R)	R(i) = |U(i)|
vec_$neg(U, Count, R)	R(i) = –U(i)
	Scalar-vector
vec_$add_constant(U, Count, a, R)	R(i) = a + U(i)
vec_$mult_constant(U, Count, a, R)	R(i) = a × U(i)
	Vector-vector
vec_$sub_constant(U, Count, a, R)	R(i) = a – U(i)
vec_$add_vector(U, V, Count, R)	R(i) = U(i) + V(i)
vec_$mult_vector(U, V, Count, R)	R(i) = U(i) × V(i)
vec_$sub_vector(U, V, Count, R)	R(i) = U(i) – V(i)
	Vector-vector-vector
vec_$add_mult_vector(U, V, W, Count, R)	R(i) = (U(i) + V(i)) × W(i)
vec_$mult_add_vector(U, V, W, Count, R)	R(i) = (U(i) × V(i)) + W(i)
vec_$mult_rsub_vector(U, V, W, Count, R)	R(i) = –(U(i) × V(i)) + W(i)
vec_$mult_sub_vector(U, V, W, Count, R)	R(i) = (U(i) × V(i)) – W(i)
	Scalar-vector-vector
vec_$add_mult(U, V, Count, a, R)	R(i) = (a + V(i)) × U(i)
vec_$mult_add(U, V, Count, a, R)	R(i) = (a × V(i)) + U(i)
vec_$mult_sub(U, V, Count, a, R)	R(i) = (a × V(i))
	Vector-vector-scalar
vec_$add_mult_constant(U, V, Count, a, R)	R(i) = (U(i) + V(i)) × a
vec_$mult_add_constant(U, V, Count, a, R)	R(i) = (U(i) × V(i)) + a
	Summation and dot product
vec_$asum(U, Count)	result = SUM(|U(i)|)
vec_$sum(U, Count)	result = SUM(U(i))
vec_$dot(U, V, Count)	result = SUM(U(i) × V(i))
	Linear recurrences
vec_$rec1(U, V, Count, R)	R(i+1) = U(i) + V(i) × R(i)
vec_$rec1c(U, Count, a, R)	R(i+1) = U(i) + a × (a × R(i))
	Copy and initialization
vec_$copy(U, V, Count)	V(i) = U (i)
vec_$init(U, V, Count,)	U(i) = a

For most of these basic routines there are eight versions for handling the variations in type and stride. Table II lists the eight versions for vec_$abs, the routine that computes an absolute value.

* Stride is the number of array elements that must be skipped over when a subscript's value is changed by 1.

Table II
Different Versions of vec_$abs

Routine	Characteristics
vec_$abs(U, Count, R)	Single-precision floating-point, unit stride
vec_$dabs(U, Count, R)	Double-precision floating-point, unit stride
vec_$iabs(U, Count, R)	32-bit integer, unit stride
vec_$iabs16(U, Count, R)	16-bit integer, unit stride
vec_$abs_i(U, Stride1, Count, R, Stride 2)	Single-precision floating-point, variable stride
vec_$dabs_i(U, Stride1, Count, R, Stride 2)	Double-precision floating-point, variable stride
vec_$iabs_i(U, Stride1, Count, R, Stride 2)	32-bit integer, variable stride
vec_$iabs16_i(U, Stride1, Count, R, Stride 2)	16-bit integer, variable stride

Because of time constraints, we could not hand-tune every routine, so we chose to concentrate on those that would derive the most benefit from tuning. For the rest, we used FORTRAN versions of the routines. Some of those routines were run through the preprocessor to unroll the loops and/or run through the software pipelining optimizer to get the best possible code with minimal effort.

Machine-Specific Features. Two features of PA-RISC 1.1 that hand-tuning was able to take particular advantage of are the FMPYADD and FMPYSUB combined operation instructions, and the ability to use double-word loads into 32-bit floating-point register pairs. In addition, floating-point instruction latencies provide the greatest opportunities for scheduling.

Because of these factors, we felt that floating-point routines would benefit more from hand-tuning than integer routines. In particular, 32-bit floating-point routines can exploit the double-word load and store feature, which is currently beyond the capabilities of the optimizer.

For some of the most critical routines, we used a nonarchitected instruction available in this particular implementation of PA-RISC 1.1 to do quad-word stores. This instruction requires longer store interlocks so it isn't always worthwhile to use it, but it was able to improve some routines by about 10%.

Double-Word Load and Stores. To use double-word loads and stores for single-precision vectors, care must be taken to ensure that the addresses are properly aligned. PA-RISC 1.1 enforces strict alignment requirements on data accesses. Thus, a single-word load must reference a word-aligned address, and a double-word load must reference a double-word-aligned address. For example, take two single-precision vectors:

```
    REAL*4 A(4),B(4)
```

The elements of arrays A and B might be laid out in memory as shown in Fig. 1.

Suppose we want to copy vector A to vector B. If we use single-word loads and stores, each element will be accessed by a separate load or store. There is no

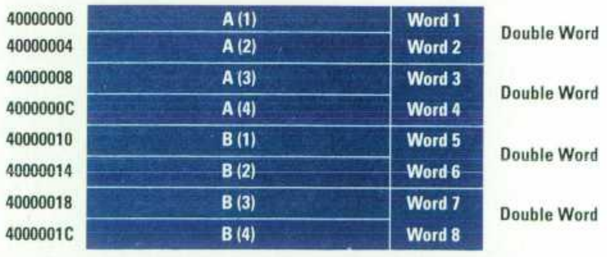


Fig. 1. The arrangement of vectors A and B in memory. All the elements are double-word-aligned.

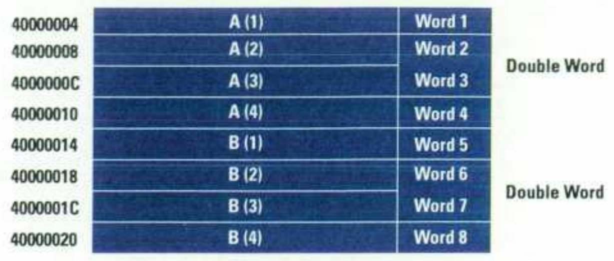


Fig. 2. The arrangement of vectors A and B in memory when the some of the elements are not double-word-aligned.

problem with alignment because each element is aligned on a word boundary. This method requires four loads and four stores.

Since vectors A and B are both double-word-aligned (the starting address is a multiple of eight), we can use double-word loads and stores, cutting the number of memory accesses in half. The first load will load **A(1)** and **A(2)** into a double floating-point register. The first store will store that register into **B(1)** and **B(2)**. This method requires only two loads and two stores.

In Fig. 2 we have a case in which the starting addresses of the vectors are not double-word-aligned. In this case only elements 2 and 3 can be copied using double-word loads and stores. The first and last elements must use single-word accesses because of alignment restrictions.

Special code is required to handle all the possible alignment combinations for the different vectors in a library routine. For example, there are 16 different possible alignment combinations for **vec_$mult_sub_vector**.

We reduced the amount of code needed to handle all these combinations by performing a single iteration for some combinations, then jumping to the code for the opposite combination. For example, if vectors 2 and 4 are double-word aligned and vectors 1 and 3 are not, we can perform the operation on one element, which effectively inverts the alignment combination. Vectors 1 and 3 will now be double-word aligned, and vectors 2 and 4 will not. We can then go to the code for the vector combination unaligned-aligned-unaligned-aligned, which takes advantage of double-word load and store instructions for that particular alignment combination.

We also took advantage of commutative operations to reduce the amount of code we had to write. Again, for **vec_$mult_sub_vector**, the multiplication is commutative, so we can swap the pointers to vectors 1 and 2, then jump to the code for the commuted combination.

Using these techniques, we reduced the number of different combinations that had to be tuned for the routine **vec_$mult_sub_vector** from 16 to six.

Instruction Scheduling

The instruction scheduling for the vector library is tuned for this particular implementation of PA-RISC 1.1. The characteristics of other implementations could very well be different.

There are requirements for minimum distance between certain types of instructions to avoid interlocks. To make the most efficient use of the procesor, the instruction sequences for independent operations can be interleaved. This is known as software pipelining, which is discussed in the article on page 39.

Another aspect of this issue is that the transformed FORTRAN source code is often structured differently from what a human programmer would write, exposing the FORTRAN compiler to code it had not seen before. The result is that we uncovered (and fixed) several minor compiler defects both in the FORTRAN compiler front end and the optimizer.

In Pursuit of HP Quality

Early in the HP-UX 8.05 release cycle, as potential performance benefits were identified, a commitment was made to use the preprocessor and to deliver specific performance on key benchmarks and applications. The subsequent development effort involved a geographically distributed HP team working together with the third party—all on a very tight schedule. In this situation, close attention to the quality assurance process was required. Three general areas of quality were addressed:

- Performance testing for both industry benchmarks and general applications
- Correctness of preprocessor source transformations
- Preprocessor acceptance of the HP FORTRAN dialect.

To address these quality issues, the following steps were taken:

- Identification of a test space to use for testing purposes
- Initiation of a beta test program
- Choice of a method for tracking and prioritizing outstanding problems
- Development of a regular testing and feedback cycle.

Identifying the Test Space. For performance related testing, standard benchmarks such as the SPEC benchmark programs and Linpack were used. Since we had committed to specific performance numbers with these benchmarks, it was crucial to monitor their progress. As the release progressed, performance related issues also came to our attention through runs of an internally maintained application test suite as well as from HP factory support personnel and from a beta program.

While some of the performance tests did help test the correctness and dialect issues of quality, we wanted to identify a set of programs or program fragments specifically for these purposes. White box testing was provided by the third party. For HP's testing process, we viewed the preprocessor as a black box, concentrating on its functionality to the FORTRAN user. To this end, we chose to concentrate on the same test bed that we use for quality assurance on the Series 700 FORTRAN compiler. In addition, to get further exposure to typical FORTRAN programs, we also developed a beta program.

This choice of a testing space did not test the complete functionality of the preprocessor. For example, procedure inlining was performed when the preprocessor was run on our test suites, but for this release we did not develop a set of tests specifically to test the inlining capabilities.

Another issue in choosing the test space was to identify the command-line option combinations to test. In the case of the preprocessor, over 30 individual options are supported, and when the different option combinations were considered, the complete set of option configurations became unreasonably large to test fully under our tight development schedule.

To handle the option situation, we concentrated on configurations most likely to be used. In most situations, we anticipated that the preprocessor would be invoked through the FORTRAN compiler driver by using one of the +OP options. These five options (+OP0 ... +OP4) were designed to be most useful for general use, and they exercise many of the main preprocessor features. For this reason, we restricted the majority of our test runs to using the +OP options.

Although not initially considered for testing purposes, the examples given in the FORTRAN manual also turned out to be important for tracking the quality of the preprocessor. Many tests were written for the manual to help explain each of the new features provided by the preprocessor. Running these tests during the release uncovered a few regressions (defects) in the preprocessor. These regressions were fixed, adding to the overall quality of the product.

Beta Program. An important source of quality issues was the beta program initiated specifically to gain additional exposure for the preprocessor. Since this was a new component of the FORTRAN compile path, it was especially important to expose the preprocessor to existing FORTRAN applications.

The results of the beta program were quite successful. Since some of the sites involved applications heavily reliant on the HP FORTRAN dialect, we uncovered a number of preprocessor problems concerning dialect acceptance. Performance issues were also raised from the beta program. In some cases significant performance gains were reported; in others, less success was achieved.

As problems reported were fixed, updated versions of the preprocessor were provided to the beta sites. In this manner, the beta program provided another source of improvement and regression tracking.

Problem Tracking. With the tight schedule and multisite team, it was important to have a mechanism for tracking problems that arose with the preprocessor. The purpose was to make sure that all problems were properly recorded, to have a common repository for problems, and to have a basis for prioritizing the outstanding problems.

Although many people could submit problems, a single team member was responsible for monitoring a list that described reported preprocessor problems and closing out problems when they were resolved. As part of this process, a test suite was developed that contained example code from each of the submitted problems. This test suite provided us with a quick method of checking recurrence of old problems as development progressed.

Since this list was used to prioritize preprocessor problems, the team developed a common set of guidelines for assigning priority levels to each submitted problem. For example, any item that caused a significant performance problem (e.g., slowdown on a key benchmark) would be assigned a very high priority, while problems with an infrequently used feature in HP FORTRAN dialect processing were given a lower priority.

During team conferences, the problem list was a regular agenda item. The team would review all outstanding problems, adjusting priority levels as considered appropriate. In this manner, we had an ongoing list representing the problems that needed to be fixed in the preprocessor.

Testing and the Feedback Cycle. As part of any quality process, it is important to develop a regular set of activities to monitor improvements and regressions in the product. As the preprocessor release entered its later stages, we developed a regular weekly cycle that coincided with the program-wide integration cycle. The activities we performed during each week of this period included:

- Review of the list of outstanding problems, identifying the next items to be addressed by the third party.
- Weekly phone conference with the third party. These meetings provided close tracking of the problems fixed the previous week as well as a discussion of any new problems to be fixed the following week.
- A regression test of the latest version of the preprocessor. Each week we received a new version of the preprocessor containing the latest fixes. The testing involved running our test suites and checking for any regressions.
- The resolution of any fixed problems and updating the outstanding problem list.
- A decision about allowing the latest version of the preprocessor to be submitted to system integration. Based on the results of test runs, the new preprocessor would be submitted if it was superior to the previous version.

The fast, regular feedback of this process towards the end of the product release cycle maximized the quality of the product within very tight delivery constraints.

Performance Analysis

The FORTRAN optimizing preprocessor has had a significant impact on the performance of FORTRAN applications. While the performance improvement seems to vary significantly based on the specifics of the code, we have seen more than a 10× speedup in some programs because of improvements in data locality, which significantly reduces cache miss rates. Array manipulation also tends to show improvement.

The Livermore Loops are a collection of kernel loops collected by the staff at Argonne Laboratories, and are frequently used as benchmarks for scientific calculations. Table I shows the performance results for these loops executing after being compiled with the optimizing preprocessor.

The improvements in loops 3, 6, 11, 12, and 18 were because of vectorization. Loop 13 benefited from loop splitting, while loop 14 benefited from loop merging. Loop 15 gained from transforming "spaghetti code" to structured code. Loop 21 gained significantly from recognition of a matrix multiply and a call to a tuned and blocked matrix multiply routine. Note that because of either the options selected or the heuristics of the optimizer, loops 5 and 8 degraded in performance.

Matrix300 is a well-known benchmark in the SPEC benchmark suite. The code in this routine performs eight matrix multiplies on two 300-by-300 matrices. In this case, the application of blocking to the matrix multiply algorithm had a significant impact on the performance of the benchmark. Table II compares the results of running the

Matrix300 benchmark with and without the optimizing preprocessor.

Note the significant reduction in cache miss rate because of the blocking techniques. This technique is applicable to a number of multidimensional matrix algorithms beyond matrix multiply.

Besides benchmarks, we have seen some significant performance improvements in other applications when the preprocessor is used. Although we had one case in which there was a 211% improvement, most FORTRAN programs have exhibited performance improvements in the 15% to 20% range. Also, as in the Livermore Loops benchmarks, we have found a few cases in which there was either no improvement in performance, or a degradation in performance. We continue to investigate these cases.

Table I
Performance of Livermore Loops Using the Preprocessor

Loop	Performance Results
1	1.5%
3	233.3%
4	1.0%
5	-23.6%
6	166.6%
7	1.8%
8	-6.7%
11	81.6%
12	63.9%
13	5.5%
14	9.0%
15	10.3%
16	0.9%
17	1.3%
18	4.1%
21	194.7%
22	8.4%
23	0.7%
24	15.0%

Loops 2, 9, 10, 19, and 20 yielded 0.0%
Harmonic Mean = 15.7%
Median Rate = 20.6%
Average Rate = 12.7%

Table II
Performance of Matrix300 Benchmark

	Without Prepro-cessor*	With Prepro-cessor*
Number of floating-point multiplies	54	19
Number of floating-point adds	54	19
Number of FMPYADDs	162	197
Number of floating-point loads	432	234
Number of floating-point stores	217	15
Number of miscellaneous instructions	576	33
Number of cache misses	64 (7.29%)	2.8 (1.1%)
HP 9000 Model 720 SPECmark	27.9	330.0

* Millions of operations

Acknowledgments
Kathy Harris, Maureen Hoffert, Manoj Dadoo, Don Gourley, and David Boundy all had significant roles in turning the product into reality as did managers Peter Morris, Marl Godfrey, Jeff Rees, and Mike Kolesar. Norm Rubin was a principal engineer and Joyce Spencer was the project manager for HP Concurrent FORTRAN, the sister product currently shipping on the Apollo DN10000. Both were an important influence on pursuing this technology for the Series 700. We are also indebted to the many HP field support engineers and factory support personnel who helped improve the quality of the preprocessor, especially Bob Montgomery and Felix Krasovec. Kuck and Associates Inc. has invested many engineer years in FORTRAN preprocessor technology. HP is pleased to be able to provide an industry leading FORTRAN product through the combination of Kuck and Associates' preprocessor and HP's FORTRAN compiler technology.

Register Reassociation in PA-RISC Compilers

Optimization techniques added to PA-RISC compilers result in the use of fewer machine instructions to handle program loops.

by Vatsa Santhanam

Register reassociation is a code improving transformation that is applicable to program loops. The basic idea is to rearrange expressions found within loops to increase optimization opportunities, while preserving the results computed. In particular, register reassociation can expose loop-invariant partial expressions in which intermediate results can be computed outside the loop body and reused within the loop. For instance, suppose the following expression is computed within a loop:

```
(loop_variant + loop_constant_1) + loop_constant_2
```

where loop_variant is a loop-varying quantity, and loop_constant_1 and loop_constant_2 are loop-invariant quantities (e.g., literal constants or variables with no definitions within a loop). In the form given above, the entire expression has to be computed within the loop. However, if this expression is reassociated as:

```
(loop_constant_1 + loop_constant_2) + loop_variant
```

the sum of the two loop-invariant terms can be computed outside the loop and added to the loop-varying quantity within the loop. This transformation effectively eliminates an add operation from the body of the loop. Given that the execution time of applications can be dominated by code executed within loops, reassociating expressions in the manner illustrated above can have a very favorable performance impact.

The term "register reassociation" is used to describe this type of optimization because the expressions that are transformed typically involve integer values maintained in registers. The transformation exploits not just the associative laws of arithmetic but also the distributive and commutative laws. Register reassociation has also been described in the literature as subscript commutation.[1]

Opportunities to apply register reassociation occur frequently in code that computes the effective address of multidimensional array elements that are accessed within loops. For example, consider the following FORTRAN code fragment, which initializes a three-dimensional array:

```
      DO 100 i = 1, DIM1
        DO 100 j = 1, DIM2
          DO 100 k = 1, DIM3
100         A(i,j,k) = 0.0
```

Arrays in FORTRAN are stored in column-major order, and by default, indexes for each array dimension start at one. Fig. 1 illustrates how array A would be stored in

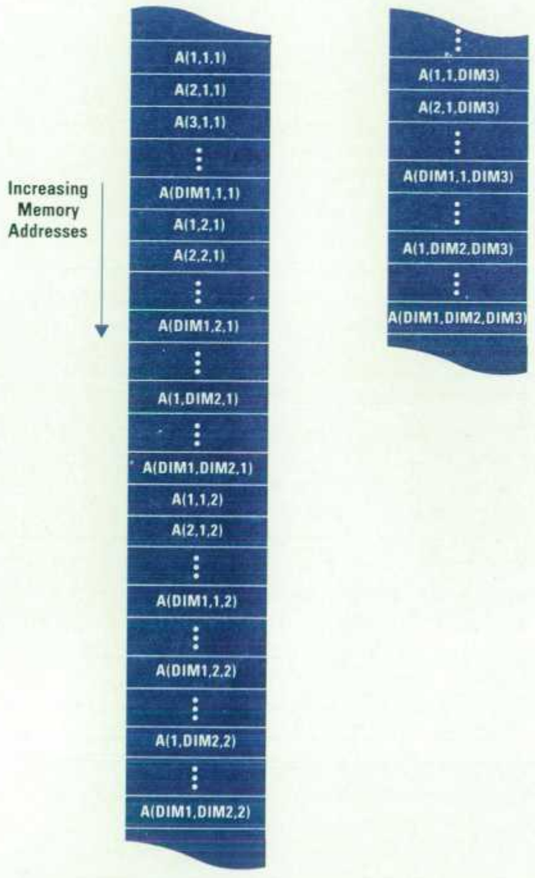


Fig. 1. Column-major storage layout for array A.

memory. Given such a storage layout, the address of the array element A(i, j, k) is given by:

$$\begin{aligned} \text{ADDR}(A(1,1,1)) &+ (k-1) \times \text{DIM2} \times \text{DIM1} \times \text{element\_size} + \\ &(j-1) \times \text{DIM1} \times \text{element\_size} + \\ &(i-1) \times \text{element\_size} \end{aligned}$$

where ADDR (A(1,1,1)) is the base address of the first element of array A, DIMn is the size of the nth dimension, and element_size is the size of each array element. Since the individual array dimensions are often simple integer constants, a compiler might generate code to evaluate the above expression as follows:

$$[((((k \times \text{DIM2}) + j) \times \text{DIM1}) + i) - ((1 + \text{DIM2}) \times \text{DIM1} + 1)] \times \text{element\_size} + (\text{ADDR}(A(1,1,1)) \quad (1)$$

Since the variable k assumes a different value for each iteration of the innermost loop in the above example, the entire expression is loop-variant.

With suitable reassociation, the address computation can be expressed as:

$$\text{ADDR}(A(i,j,k)) = \alpha \times k + \beta \qquad (2)$$

where α and β are loop-invariant values that can be computed outside the loop, effectively eliminating some code from the innermost loop. From expression 1:

$$\alpha = \text{DIM1} \times \text{DIM2} \times \text{element\_size}$$

and

$$\beta = [(j \times \text{DIM1} + i) - ((1 + \text{DIM2}) \times \text{DIM1} + 1)] \times \text{element\_size} + \text{ADDR}(A(1,1,1)).$$

The simplified expression $(\alpha \times k + \beta)$ evaluates a linear arithmetic progression through each iteration of the innermost loop. This exposes an opportunity for another closely related loop optimization known as *strength reduction*.[2,3] The basic idea behind this optimization is to maintain a separate temporary variable that tracks the values of the arithmetic progression. By incrementing the temporary variable appropriately in each iteration, the multiplication operation can be eliminated from the loop. For our simple example, the temporary variable would be initialized to $\alpha + \beta$ outside the innermost loop and incremented by α each time through the loop. This concept is illustrated in Fig. 2.

Note that this can be particularly beneficial for an architecture such as PA-RISC in which integer multiplication is usually translated into a sequence of one or more instructions possibly involving a millicode library call.†

On some architectures, such as PA-RISC and the IBM RISC System/6000, register reassociation and strength reduction can be taken one step further. In particular, if the target architecture has an autoincrement addressing mode, incrementing the temporary variable that maintains the arithmetic progression can be accomplished automatically as a side effect of a memory reference. Through this additional transformation, array references in loops can essentially be converted into equivalent, but cheaper, autoincrementing pointer dereferences.

An Example

To clarify the concepts discussed so far, let us compare the PA-RISC assembly code for the above example with and without register reassociation. Assume that the source code fragment for the example is contained in a subroutine in which the array A is a formal parameter declared as:

```
REAL *4     A(10,20,30)
```

The loop limits DIM1, DIM2, and DIM3 take on the constant values 10, 20, and 30 respectively. The following assembly code was produced by the HP 9000 Series 800 HP-UX* 8.0 FORTRAN compiler at level 2 optimization with loop unrolling and reassociation completely disabled.††

```
 1: LDI      1,%r31                     ; i <- 1
 2: FCPY,SGL %fr0L,%fr4L                ; fr4 <- 0.0
 3: LDI      20,%r24                    ; r24 <- 20
 4: LDI      600,%r29                   ; r29 <- 600
 5: i_loop_start
 6: LDI      1,%r23                     ; j <- 1
 7: j_loop_start
 8: LDI      20,%r25                    ; t<k*20> <- 20
 9: k_loop_start
10: ADD      %r25,%r23,%r19             ; r19 <- t<k*20> + j
11: SH2ADD %r19,%r19,%r20               ; r20 <- r19*5
12: SH1ADD %r20,%r31,%r21               ; r21 <- r20*2 + i
13: LDO      -211(%r21),%r22            ; r22 <- r21 - 211
14: LDO      20(%r25),%r25              ; t<k*20> <- t<k*20> + 20
15: COMB, <= %r25,%r29,k_loop_start     ; if t<k*20> <= r29
                                        ; go to k_loop_start
16: FSTWX,S %fr4L,%r22(0,%r26)          ; *[r22*4 + ADDR(A(1,1,1))] <- fr4
17: LDO      1(%r23),%r23               ; j <- j + 1
18: COMB,<=,N %r23,%r24,k_loop_start    ; if j <= r24
                                        ; go to k_loop_start
19: LDI      20,%r25                    ;
20: LDO      1(%r31),%r31               ; i <- i + 1
21: COMIBF,<,N 10,%r31,j_loop_start     ; if i <= 10 go to j_loop_start
22: LDI      1,%r23                     ;
23: BV,N     %r0(%r2)                   ; return
```

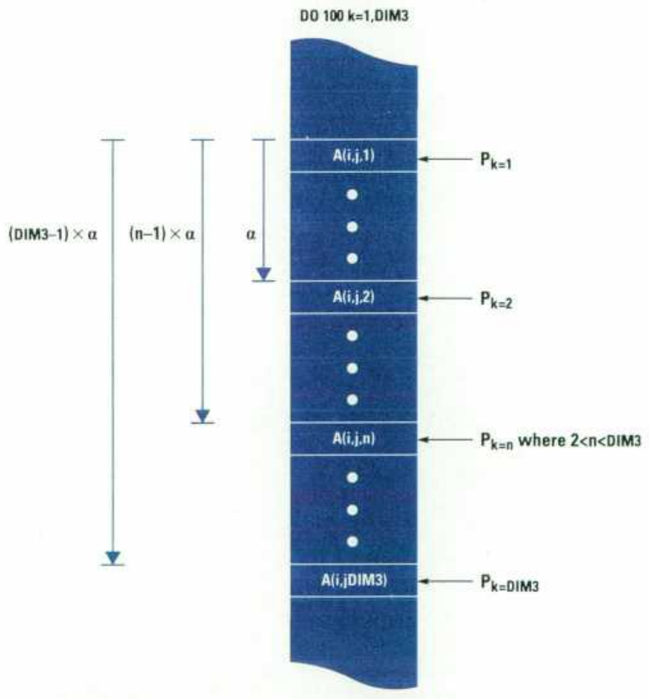


Fig. 2. An illustration of using a temporary variable P to track the address expression $\alpha k+\beta$ to stride through array A.

† Removing multiplications from loops is beneficial even on version 1.1 of PA-RISC which defines a fixed-point multiply instruction that operates on the floating-point register file. To exploit this instruction, it may be necessary to transfer data values from general registers to floating-point registers and then back again.

†† The Series 800 HP-UX 8.0 FORTRAN compiler does not include the code generation and optimization enhancements implemented for the Series 700 FORTRAN compiler. Examining the code produced by the Series 800 FORTRAN compiler instead of the Series 700 FORTRAN compiler helps better isolate and highlight the full impact of register reassociation.

The labels i_loop_start, j_loop_start, and k_loop_start that mark a start of a loop body and the annotations are not generated by the compiler, but were added for clarity.

The following sections describe the optimizations performed in the above code segment.

Constant Folding. Given that DIM2 = 20 and DIM1 = 10, the partial expression – ((1 + DIM2) × DIM1 + 1) has been evaluated to be –211 at compile time.

Loop Invariants. Loop-invariant code motion has positioned instructions that compute loop-invariant values as far out of the nest of loops as possible. For instance, the definition of the floating-point value 0.0 has been moved from the innermost loop to outside the body of all three loops where it will be executed exactly once (line 2). (On PA-RISC systems, when floating-point register 0 is used as a source operand in an instruction other than a floating-point store, its value is defined to be 0.0.)

Index Shifting. The innermost k-loop (lines 10 to 16) contains code that computes the partial expression ((((k × DIM2) + j) × DIM1) + i) whose value is added to –211 and stored in register 22 (line 13) before being used in the instruction that stores 0.0 in the array element. Register 22 is used as the index register; it is scaled by four (to achieve the multiplication by the element size) and then added to base register 26, which contains ADDR (A(1,1,1)). These operations produce the effective address for the store instruction.

The compiler has strength-reduced the multiplication of k by DIM2 in line 14. A temporary variable that tracks the value of k × 20 (referred to as t<k*20> in the annotations) has been assigned to register 25 in line 8. This temporary variable is used in the calculation of the address of A(i, j, k). By incrementing the temporary variable by 20 on each iteration of the innermost loop, the multiplication of k by 20 is rendered useless, and therefore removed.

Linear Function Test Replacement. After strength-reducing k × 20, the only other real use of the variable k is to check for the loop termination condition (line 15). Through an optimization known as *linear function test replacement*,[2,3] the use of the variable k in the innermost loop termination check is replaced by a comparison of the temporary variable t<k*20> against 600, which is the value of DIM3 scaled by a factor of 20. This optimization makes the variable k superfluous, thus enabling the compiler to eliminate the instructions that initialize and increment its value.

Branch Scheduling. A final point to note is that the loop termination checks for the i-loop and j-loop are performed using the nullifying backward conditional branch instructions in lines 18 and 21. In PA-RISC, the semantics of backward nullifying conditional branches are that the delay slot instruction (the one immediately following the branch instruction) is executed only if the branch is taken and suppressed otherwise. This nullification feature allows the compiler to schedule the original target of a backward conditional branch into its delay slot and redirect the branch to the instruction following the original target.

In contrast to the i-loop and j-loop, the innermost loop termination check is a non-nullifying backward conditional branch whose delay slot instruction is always executed, regardless of whether the branch is taken.

Applying Reassociation

The most important point to note about the assembly code given above is that the innermost loop, where much of the execution time will be spent, consists of the seven instructions in lines 10 to 16. By applying register reassociation to the innermost loop, and making use of the base-register modification feature available on certain PA-RISC load and store instructions, the innermost loop can be reduced to three instructions.†

The following code fragment shows the three instructions for the innermost loop. Registers r19 to r22 and fr4 are assumed to be initialized with values indicated by the annotations.

```
 ; Initialize general registers r19 through r22 and floating-point
 ; register fr4
                                       ; fr4 <– 0.0
                                       ; r19 <– 1
                                       ; r20 <– 30
                                       ; r21 <– ADDR(A(i,j,1))
                                       ; r22 <– 800
 ; The three innermost loop instructions
k_loop_start
 LDO 1(%r19),%r19                      ; k <– k + 1
 COMB,<= %r19,%r20,k_loop_start ; if k <= r20 go to k_loop_start
 FSTWX,M %fr4L,%r22(0,%r21)     ; *[r21] <– fr4, r21 <– r21 + r22
```

This assembly code strides through the the elements of array A with a compiler-generated temporary pointer variable that is maintained in register 21. This register pointer variable is initialized to the address of A(i,j,1) before entry to the innermost loop and postincremented by 800 bytes after the value 0.0 is stored in A(i,j,k).

This code also reflects the real intent of the initialization loop, which is to initialize the array A by striding through the elements in row-major order. It does this in fewer instructions by exploiting PA-RISC's base-register modification feature. The equivalent semantics for the above inner-loop code expressed in C syntax is:

```
for (p = &a(i,j,1), k = 1; k <= 30; k++)
{
    *p++ = 0.0;
}
```

The code sequence for the k-loop in the assembly code fragment can be improved even further. Note that as a result of register reassociation, the loop variable k, which is maintained in register 19, is now only used to control the iteration count of the loop. Using the linear function test replacement optimization mentioned earlier, the loop variable can be eliminated. Specifically, the loop termination check in which the variable k in register 19 is compared against the value 30 in register 20 can be replaced by an equivalent comparison of the compiler-generated temporary pointer variable (register 21) against

†Base-register modification of loads and stores effectively provides the autoincrement addressing mode described earlier.

the address of the array element A(i, j,30). This can reduce the innermost loop to just the following two instructions.

```
  FSTWX,M %fr4L,%r22(0,%r21)      ;*[r21] <– fr4,r21 <– r21 + r22
k_loop_start                      ;The two innermost loop instructions
  COMB,<=,N %r21,%r20,k_loop_start;If r21 <= ADDR (A(i,j,30)) go to
                                  ;k_loop_start
  FSTWX,M %fr4L,%r22(0,%r21)      ;*[r21] <– fr4,r21 <– r21+r22
```

Register r20 would have to be initialized to the address of A(i, j,30).

On PA-RISC machines, if a loop variable is only needed to control the loop iteration count, the loop variable can often be eliminated using a simpler technique. Specifically, the PA-RISC instruction set includes the ADDB (add and branch) and ADDIB (add immediate and branch) conditional branch instructions. These instructions first add a register or an immediate value to another register value and then compare the result against zero to determine the branch condition.

If a loop variable is incremented by the same amount on each iteration of the loop and if it is needed solely to check the loop termination condition, the instructions that increment the loop variable can be eliminated from the body of the loop by replacing the loop termination check with an ADDB or ADDIB instruction using an appropriate increment value and a suitably initialized general-purpose count register.

For our small example, the innermost loop can be transformed into a two-instruction countdown loop using the ADDIB instruction as shown in the following code.

```
  LDI   –29, %r19                 ; initialize count register
k_loop_start
  ADDIB,<= 1,%r19, k_loop_start   ; r19 <– r19 + 1, if r19 <= 0 go to
                                  ; k_loop_start
  FSTWX,M %fr4L,%r22(0,%r21)      ;*[r21] <– fr4, r21 <– r22 + r21
```

The j-loop and i-loop of our example can be similarly transformed. The increment and branch facility is not unique to PA-RISC, but unlike some other architectures, the general-purpose count register is not a dedicated register, allowing multiple loops in a nest of loops to be transformed conveniently.

Note that even though reassociation has helped reduce the innermost loop from seven instructions to two instructions, one cannot directly extrapolate from this a commensurate improvement in the run-time performance of this code fragment. In particular, the execution time for this example can be dominated by memory subsystem overhead (e.g., cache miss penalties) because of poor data locality associated with data assignments to array A.

Compiler Implementation

Register reassociation and other ideas presented in this article were described in the literature several years ago.[3,4,5,6] Compilers that perform this optimization include the DN10000 HP Apollo compilers and the IBM compilers for the System 370 and RISC System/6000 architectures.[5,6,7]

Strength reduction and linear function test replacement have been implemented in PA-RISC compilers from their very inception. The implementation of these optimizations is closely based on the algorithm described by Allen Cocke, and Kennedy.[2] Register reassociation, on the other hand, has been implemented in the PA-RISC compilers very recently. The first implementation was added to the HP 9000 Series 700 compilers in the 8.05 release of the HP-UX operating system. Register reassociation is enabled through the use of the +OS compiler option, which is supported by both the FORTRAN and C compilers in release 8.05 of the HP-UX operating system.

The implementation of register reassociation offered in HP-UX 8.05 is somewhat limited in scope. Register reassociation is performed only on address expressions found in innermost straight-line loops. The scope of register reassociation has been greatly extended in the compilers available with release 8.3 of the HP-UX operating system, which runs on the Series 700 PA-RISC workstations. Register reassociation, which is now performed by default in the C, C++, FORTRAN, and Pascal compilers at level 2 optimization, is attempted for all loops (with or without internal control flow) and not limited merely to straight-line innermost loops. Furthermore, in close conjunction with register reassociation, these compilers make aggressive use of the PA-RISC ADDIB and ADDB instructions and the base-register modification feature of load and store instructions to eliminate additional instructions from loops as described earlier.

Using the example given earlier, the following code is generated by the Series 700 HP-UX 8.3 FORTRAN compiler at optimization level 2 (without specifying the +OP FORTRAN preprocessor option).

```
 1: FCPY,SGL    %fr0L,%fr4L         ; fr4 <– 0.0
 2: LDI         800,%r31            ; Pijk_inc <– 800
 3: LDI         –9,%r23             ; i_cnt <–  –9

 4: i_loop_start
 5: COPY        %r26,%r24           ; Pij1 <– Pi11
 6: LDI         –19,%r25            ; j_cnt <–  –19
 7: j_loop_start
 8: COPY        %r24,%r29           ; Pijk <– Pij1
 9: LDI         –29,%r19            ; k_cnt <–  –29
10: k_loop_start
11: ADDIB,<=    1,%r19,k_loop_start; k_cnt <– k_cnt + 1
                                    ; if k_cnt <= 0 go to k_loop_start
12: FSTWX,M %fr4L,%r31(0,%r29)      ; *(Pijk) <– 0.0; Pijk <– Pikj + 800
13: ADDIB,<=    1,j_loop_start      ; j_cnt <– j_cnt + 1
                                    ; if j_cnt <= 0 go to j_loop_start
14: LDO         40(%r24),%r24       ; Pij1 <– Pij1 + 40
15: ADDIB,<=    1,%r23,i_loop_start ; i_cnt <– i_cnt + 1;
                                    ; if i_cnt <= 0 go to i_loop_start
16: LDO         4(%r26),%r26        ; Pi11 <– Pi11 + 4
17: BV,N        %r0(%r2)            ; return
```

This code shows that register reassociation and strength reduction have been applied to all three loop nests.

k-Loop Optimization. For the k-loop, a compiler-generated temporary pointer variable Pijk, which is maintained in register 29, is used to track the address of the array element A(i,j,k). This temporary variable is incremented by 800 through base-register modification (line 12) instead of the original loop variable k incrementing by one. The 800 comes from DIM1 × DIM2 × element_size (10 × 20 × 4), which represents the invariant quantity α in expression 2.

Before entering the k-loop, Pijk has to be initialized to the address of A(i,j,1) because the variable k was originally initialized to one before entering the k-loop. For this example, the address of A(i,j,1) can be computed as:

$$[(((1 \times 20) + j) \times 10) + i) - ((1 + 20) \times 10 + 1)] \times 4 + \text{ADDR}(A(1,1,1))$$

which can be simplified to

$$40 \times j + ((4 \times i) - 44 + \text{ADDR}(A(1,1,1))).$$

which for the j-loop is a linear function of the loop variable j of the form:

$$\alpha j + \beta$$

where:

$$\alpha = 40$$

and

$$\beta = ((4 \times i) - 44 + \text{ADDR}(A(1,1,1))).$$

j-Loop Optimization. To strength-reduce the address expression for A(i,j,1), the compiler has created a temporary variable Pij1, which is maintained in register 24. Wherever the original loop variable j was incremented by one, this temporary variable is incremented by 40 (line 14). Before entering the j-loop, Pij1 has to be initialized to the address of A(i,1,1) since the variable j was originally initialized to one before entering the j-loop. For this example, the address of A(i,1,1) can be computed as

$$[(((1 \times 20) + 1) \times 10) + i) - ((1 + 20) \times 10 + 1)] \times 4 + \text{ADDR}(A(1,1,1))$$

which can be simplified to

$$4 \times i + (\text{ADDR}(A(1,1,1)) - 4).$$

which for the i-loop is a linear function of the loop-varying quantity i of the form:

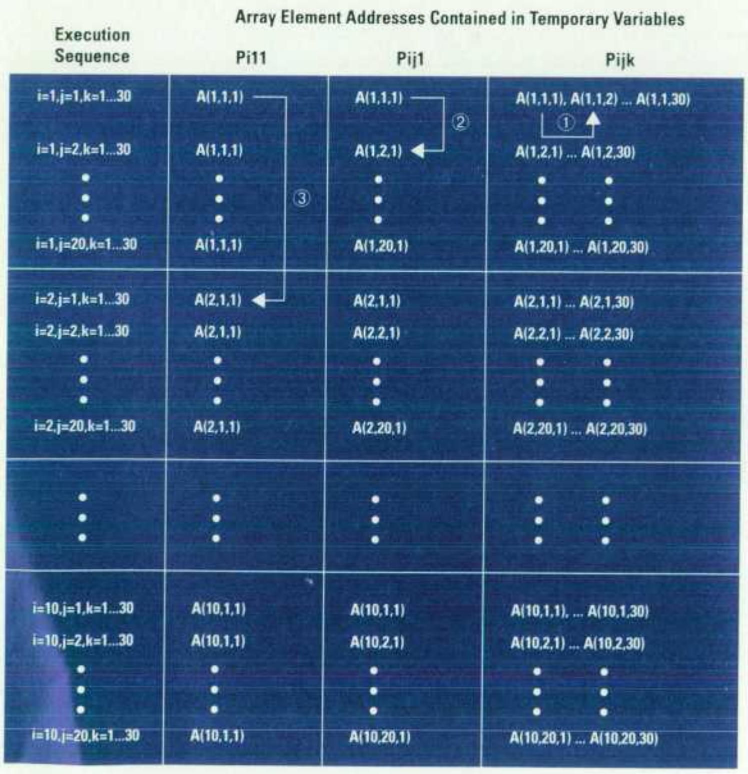
$$\alpha i + \beta$$

where:

$$\alpha = 4$$

and

$$\beta = (\text{ADDR}(A(1,1,1)) - 4).$$



Array Element Addresses Contained in Temporary Variables

Execution Sequence	Pi11	Pij1	Pijk
i=1,j=1,k=1...30	A(1,1,1)	A(1,1,1) ②	A(1,1,1), A(1,1,2) ... A(1,1,30) ①
i=1,j=2,k=1...30	A(1,1,1)	A(1,2,1)	A(1,2,1) ... A(1,2,30)
• • •	• • • ③	• • •	• • •
i=1,j=20,k=1...30	A(1,1,1)	A(1,20,1)	A(1,20,1) ... A(1,20,30)
i=2,j=1,k=1...30	A(2,1,1)	A(2,1,1)	A(2,1,1) ... A(2,1,30)
i=2,j=2,k=1...30	A(2,1,1)	A(2,2,1)	A(2,2,1) ... A(2,2,30)
• • •	• • •	• • •	• • •
i=2,j=20,k=1...30	A(2,1,1)	A(2,20,1)	A(2,20,1) ... A(2,20,30)
• • •	• • •	• • •	• • •
i=10,j=1,k=1...30	A(10,1,1)	A(10,1,1)	A(10,1,1), ... A(10,1,30)
i=10,j=2,k=1...30	A(10,1,1)	A(10,2,1)	A(10,2,1) ... A(10,2,30)
• • •	• • •	• • •	• • •
i=10,j=20,k=1...30	A(10,1,1)	A(10,20,1)	A(10,20,1) ... A(10,20,30)

Initially Pi1 1 = Pij1 = Pijk = ADDR(A(1,1,1))

① ADDR (A(1,1,2)) = ADDR(A(1,1,1)) + 800 Bytes

② ADDR (A(1,2,1)) = ADDR(A(1,1,1)) + 40 Bytes

③ ADDR (A(2,1,1)) = ADDR(A(1,1,1)) + 4 Bytes

Fig. 3. The array element addresses contained in each of the temporary variables during different iterations of the i, j, and k loops of the example code fragment.

i-Loop Optimization. To strength-reduce the address expression for A(i,1,1), the compiler has created a temporary variable Pi11, which is maintained in register 26. Wherever the original loop variable i was incremented by one, this temporary variable is incremented by four (line 16). Before entering the i-loop, Pi11 has to be initialized to the address of A(1,1,1) since the variable i was originally initialized to one before entering the i-loop.

The address of A(1,1,1) is passed as a formal parameter to the subroutine containing our code fragment since by default, parameters are passed by reference in FORTRAN (which implies that the first argument to our subroutine is the address of the very first element of array A). On PA-RISC machines, the first integer parameter is passed by software convention in register 26, and since Pi11 is maintained in register 26, no explicit initialization of the compiler-generated temporary Pi11 is needed.

The values assumed by all three compiler-generated temporary variables during the execution of the example code fragment given above are illustrated in Fig. 3. Note that the elements of array A are initialized in row-major order. Because FORTRAN arrays are stored in column-major order, elements of the array are not accessed contiguously. This could result in cache misses and thus some performance degradation. For FORTRAN this problem can be remedied by using the stride-1 inner loop selection transformation described on page 26. This transformation examines nested loops to determine if the loops can be rearranged so that a different loop can run as the inner loop.

Loop Termination. The compiler has managed to eliminate all three of the original loop variables by replacing all loop termination checks with equivalent ADDIB instructions (lines 11, 13, and 15). Coupled with the use of base-register modification, the innermost loop has been reduced to just two instructions (compared to seven instructions without reassociation).

In implementing register reassociation (and exploiting architectural features such as base-register modification) in a compiler, several factors need to be taken into account. These include the number of extra machine registers required by the transformed instruction sequence, the number of instructions eliminated from the loop, the number of instructions to be executed outside the loop (which can be important if the loop iterates only a few times), and the impact on instruction scheduling. These and other heuristics are used by the HP-UX 8.3 compilers in determining whether and how to transform integer address expressions found in loops.

Finally, the register reassociation phase of the compiler shares information about innermost loops (particularly base-register modification patterns) with the software pipelining phase. The pipelining phase uses this information to facilitate the overlapped execution of multiple iterations of these innermost loops (see "Software Pipelining in PA-RISC Compilers" on page 39).

Conclusion

Register reassociation is a very effective optimization technique and one that makes synergistic use of key PA-RISC architectural features. For loop-intensive numeric applications whose execution times are not dominated by memory subsystem overhead, register reassociation can improve run-time performance considerably, particularly on hardware implementations with relatively low floating-point operation latencies.

Acknowledgments

I would like to thank members of HP's California Language Laboratory and HP's Massachusetts Language Laboratory who contributed ideas to the design and development of register reassociation optimization in PA-RISC compilers. Special thanks go to Bob Gottlieb and Richard Schooler for sharing their insights on register reassociation and strength reduction algorithms used in the Apollo DN10000 compilers.

References

1. R.G. Scarborough and H.G. Kolsky, "Improved Optimization of FORTRAN Object Programs," *IBM Journal of Research and Development*, Vol. 24, no. 6, November 1980, pp. 660-676.
2. F.E. Allen, J. Cocke, and K. Kennedy, "Reduction of Operator Strength," *Program Flow Analysis*, Prentice Hall, 1981, pp. 79-101.
3. A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986, pp. 643-648.
4. R. L. Sites, "The Compilation of Loop Induction Expressions," *ACM Transactions on Programming Languages and Systems*, Vol. no. 1, 1977, pp. 50-57.
5. M. Auslander and M. Hopkins, "An Overview of the PL.8 Compiler," *Proceedings of the SIGPLAN Symposium on Compiler Construction*, 1982, pp. 22-31.
6. J. Cocke and P. Markstein, "Measurement of Program Improvement Algorithms," *Proceedings of the IFIP Congress*, 1980, pp. 221-228.
7. K. O'Brien, et al, "Advanced Compiler Technology for the RISC System/6000 Architecture," *IBM RISC System/6000 Technology*, 1990, pp. 154-161.

Software Pipelining in PA-RISC Compilers

The performance of programs with loops can be improved by having the compiler generate code that overlaps instructions from multiple iterations to exploit the available instruction-level parallelism.

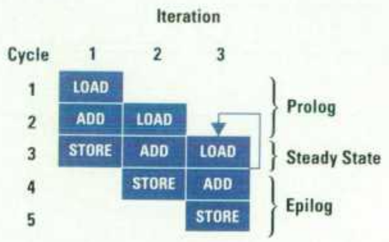
by Sridhar Ramakrishnan

In the hardware environment, pipelining is the partitioning of instructions into simpler computational steps that can be executed independently in different functional units like adders, multipliers, shifters, and so on. Software pipelining is a technique that organizes instructions to take advantage of the parallelism provided by independent functional units. This reorganization results in the instructions of a loop being simultaneously overlapped during execution—that is, new iterations are initiated before the previous iteration completes.

The concept of software pipelining is illustrated in Fig. 1. Fig. 1a shows the sequence of instructions that loads a variable, adds a constant, and stores the result. We assume that the machine supports a degree of parallelism so that for multiple iterations of the instructions shown in Fig. 1a, the instructions can be pipelined so that a new iteration can begin every cycle as shown in Fig. 1b. Fig. 1b also shows the parts of the diagram used to illustrate a software pipeline. The prolog is the code necessary to set up the steady-state condition of the loop. In steady state one iteration is finishing every cycle. In our example three iterations are in progress at the same time. The epilog code finishes executing all the operations that were started in the steady state but have not yet completed.

1. LOAD
2. ADD
3. STORE

(a)



(b)

Fig. 1. A software pipeline example. (a) The sequence of instructions in a one-stage pipeline (or one iteration). (b) Multiple iterations of the instructions shown in (a) pipelined over parallel execution components.

This example also illustrates the performance improvement that can be realized with software pipelining. In the example in Fig. 1 the pipelined implementation completes three iterations in five cycles. To complete the same number of iterations without pipelining would have taken nine cycles.

Loop Scheduling

As instructions proceed through the hardware pipeline in a PA-RISC machine, a hardware feature called pipeline interlock detects when an instruction needs a result that has yet to be produced by some previously executing instruction or functional unit. This situation results in a pipeline stall. It is the job of the instruction scheduler in the compiler to attempt to minimize such pipeline stalls by reordering the instructions. The following example shows how a pipeline stall can occur.

For the simple loop

```
for ( I = 0; I < N; I = I + 1 ) {
  A[I] = A[I] + C*B[I]
}
```

the compiled code for this loop (using simple pseudo instructions) might look like:

```
for ( I = 0; I < N; I = I + 1 ) {
  LOAD  B[I],  R2
  LOAD  A[I],  R1
  MULT  C, R2, R3  ; R3 = C*R2
                                  ; pipeline stalls R3 needed
  ADD  R1, R3, R4  ; R4 = R1 + R3
                                  ; pipeline stalls R4 needed
  STORE R4,   A[I]
}
```

Assume that for this hypothetical machine the MULT, ADD, STORE, and LOAD instructions take two cycles each. We will also assume in this example and throughout this paper that no memory access suffers a cache miss. Fig. 2 illustrates how the pipeline stalls when the ADD and STORE instructions must delay execution until the values of R3 and R4 become available. Clearly, this becomes a serious

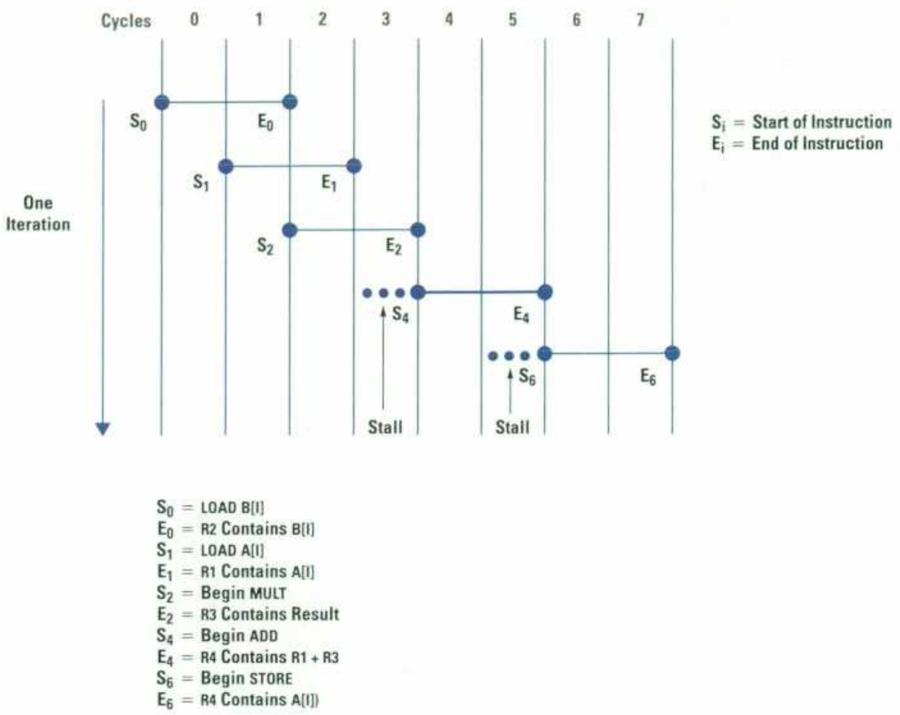


Fig. 2. An illustration of a pipeline stall.

problem if the MULT and ADD instructions have multiple-cycle latencies (as in floating-point operations). If we ignore the costs associated with the branch instruction in the loop, each iteration of the above loop would take eight cycles. Put differently, the processor is stalled for approximately 25% of the time (two cycles out of every eight).

One way of avoiding the interlocks in this example is to insert useful and independent operations after each of the MULT and ADD instructions. For the example above there are none. This problem can be solved by unrolling the loop a certain number of times as follows:

```
for ( I = 0; I < N; I = I + 2 ) {
  A[I]   = A[I]   + C*B[I]
  A[I+1] = A[I+1] + C*B[I+1]
}
```

Notice that the loop increment is changed from one to two to take into account the fact that each time the loop is entered we now perform two iterations of the original loop. There is additional compensation code* that is not shown here for the sake of simplicity.

The best schedule for this loop is as follows:

```
for ( I = 0; I < N; I = I + 2 ) {
  LOAD B[I],  R2
  LOAD A[I],  R1
  MULT C, R2, R3
  LOAD B[I+1], R6
  ADD  R1, R3, R4
  LOAD A[I+1], R5
  MULT C, R6, R7
  STORE R4, A[I]
  ADD  R5, R7, R8
  ;stall R8 needed
  STORE R8, A[I+1]
}
```

*In the context of software pipelining, compensation code refers to the code that sets up the steady state and the code that executes after completion of the steady state. Compensation code is discussed later in this article.

If we assume perfect memory access on the LOADs and STOREs, this schedule will execute two iterations every 11 cycles (again, ignoring the costs associated with the branch instruction). Fig. 3 shows what happens during each cycle for one iteration of the unrolled loop.

Despite the improvement made with loop unrolling, there are three problems with this technique. First, and perhaps most important is that the schedule derived for a single iteration of an unrolled loop does not take into account the sequence of instructions that appears at the beginning of the next iteration. Second, we have a code size expansion that is proportional to the size of the original loop and unroll factor. Third, the unroll factor is determined arbitrarily. In our example, the choice of two for the unroll factor was fortuitous since the resulting schedule eliminated one stall. A larger unroll factor would have generated more code than was necessary.

Software pipelining attempts to remedy some of the drawbacks of loop unrolling. The following code is a software pipelined version of the above example (again, we do not show the compensation code):

```
for ( I = 0; I < N; I = I + 4 ) {
  LOAD B[I+3], R14     ; start the fourth iteration
  LOAD A[I+3], R13     ; start the fourth iteration
  MULT C, R10, R11     ; R11 = C * B[I+2] (third iteration)
  ADD  R5, R7, R8      ; A[I+1] = A[I+1] + C * B[I+1] (second
                       ; iteration)
  STORE R4, A[I]       ; finish the first iteration.
}
```

Fig. 4 shows the pipeline diagram for this example. Notice that in a single iteration of the pipelined loop, we are performing operations from four different iterations.

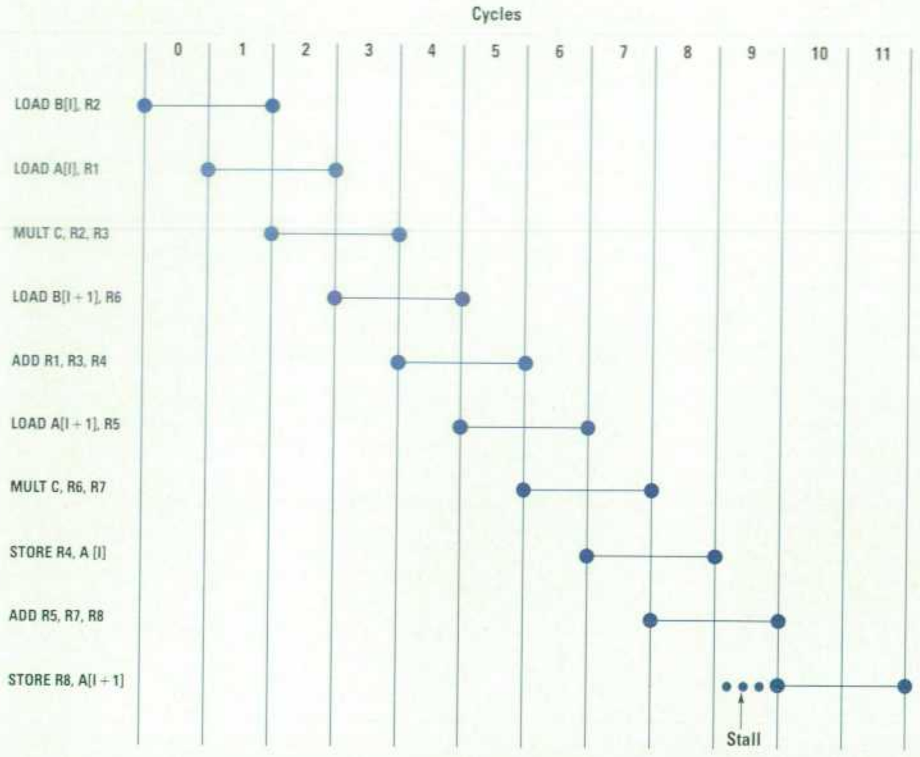


Fig. 3. One iteration of the example in Fig. 2 after the code is unrolled to execute two iterations of the previous code in one iteration.

However, each successive iteration of the pipelined loop would destroy the values stored in the registers that are needed three iterations later. On some machines, such as the Cydra-5, this problem is solved by hardware register renaming.[1] In the PA-RISC compilers, this problem is solved by unrolling the steady-state code u times, where u is the number of iterations simultaneously in flight. In Fig. 4, u is four, since there are four iterations executing simultaneously during cycles 6 and 7.

Software pipelining is not without cost. First, like loop unrolling, it has the code size expansion problem. Second, there is an increased use of registers because each new iteration uses new registers to store results. If this increased register use cannot be met by the available supply of registers, the compiler is forced to generate "spill" code, in which results are loaded and stored into memory. The compiler tries to ensure that this does not happen. Third, the PA-RISC compilers will not handle loops that have control flow in them (for example, a loop containing an if-then statement).

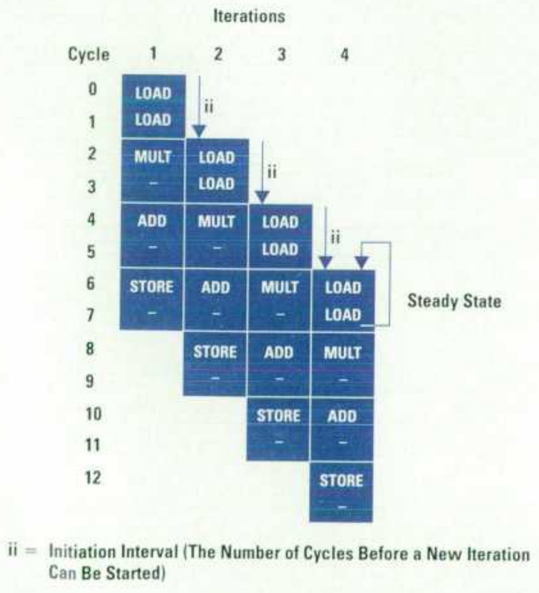


Fig. 4. A software pipeline diagram of the example code fragment.

However, unlike unrolling in which the unrolling factor is arbitrarily determined, the factor by which the steady-state code is unrolled in software pipelining is determined algorithmically (this will be explained in detail later). A key advantage of software pipelining is that the instruction pipeline filling and draining process occurs once outside the loop during the prolog and epilog section of the code, respectively. During this period, the loop does not run with the maximum degree of overlap among the iterations.

Pipeline Scheduling

To pipeline a loop consisting of N instructions, the following questions must be answered:

- What is the order of the N instructions?
- How frequently (in cycles) should the new iteration be initiated? (This quantity is called the *initiation interval*, or *ii*.)

Conventional scheduling techniques address just the first question.

The goal of pipelining is to arrive at a minimum value of ii because we would like to initiate iterations as frequently as possible. This section will provide a brief discussion about how the value of ii is determined in the PA-RISC compilers. More information on this subject is provided in reference 2.

The scheduling process is governed by two kinds of constraints: resource constraints and precedence

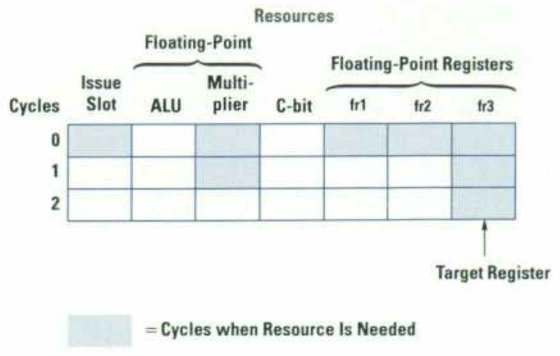

Resources
Floating-Point
Floating-Point Registers
Cycles
Issue Slot
ALU
Multi-plier
C-bit
fr1
fr2
fr3
0
1
2
Target Register
= Cycles when Resource Is Needed

Fig. 5. An example of a resource reservation table for the FMPY instruction.

constraints. Resource constraints stem from the fact that a given machine has a finite number of resources and the resource requirements of a single iteration should not exceed the available resources. If an instruction is scheduled at cycle x, we know that the same instruction will also execute at cycle x + ii, cycle x + (2 × ii), and so on because iterations are initiated every ii cycles. For the example shown in Fig. 4 ii is two.

In PA-RISC compilers we build a resource reservation table associated with each instruction. For example, the instruction:

```
FMPY,DBL fr1, fr2, fr3    ; fr3 = fr1 * fr2
```

would have the resource reservation table shown in Fig. 5 for the HP 9000 Series 700 family of processors. The reservation table defines the resources used by an instruction for each cycle of execution. For example, the FMPY instruction modeled in Fig. 5 requires the floating-point multiplier and the target register (fr3) during its second cycle of execution. The length of the table is dependent on the latency of the associated instruction.

Precedence constraints are constraints that arise because of dependences in the program. For example, for the instructions:

```
FMPY fr1, fr2, fr3
FADD fr3, fr4, fr2
```

there is a dependence from the FMPY instruction to the FADD instruction. Also, there is a dependence that goes from the FADD to the FMPY instruction because the FMPY from the next iteration cannot start until the FADD from the preceding iteration completes. Such dependencies can be represented as a graph in which the nodes represent machine instructions and the edges represent the direction of dependence (see Fig. 6). The attributes on the edges represent:

- d: a delay value (in cycles) from node u to node v. This value implies that to avoid a stall node v can start no earlier than d cycles after node u starts executing.
- p: a value that represents the number of iterations before the dependence surfaces (i.e., minimum iteration distance).* This is necessary because we are overlapping multiple iterations. A dependence that exists in the same iteration will have p = 0 (FADD depends on fr3 in Fig. 6). Values of p are always positive because a node cannot depend on a value from a future iteration. Edges that have p = 0 are said to represent intra-iteration dependences, while nonzero p values represent inter-iteration dependences.

Given an initiation interval, ii, and an edge with values <p,d> between two nodes u and v, if the function S(x) gives the cycle at which node x is scheduled with respect to the start of each iteration, we can write:

$$S(v) - S(u) \geq d(u,v) - ii \times p(u,v) \qquad (1)$$

If p(u, v) = 0 then:

$$S(v) - S(u) \geq d(u, v).$$

Equation 1 is depicted in Fig. 7.

The goal of scheduling the N instructions in the loop is to arrive at the schedule function S and a value for ii. This is done in the following steps:

1. Build a graph representing the precedence constraints between the instructions in the loop and construct the resource reservation table associated with each of the nodes of the graph.

2. Determine the minimum value of the initiation interval, (MII) based on the resource requirements of the N instructions in the loop. For example, if the floating-point multiply unit is used for 10 cycles in an iteration, and there is only one such functional unit, MII can be no smaller than 10 cycles.

3. Determine the recurrence minimum initiation interval RMII. This value takes into account the cycles associated

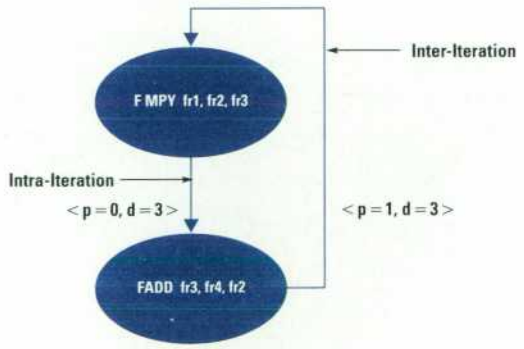
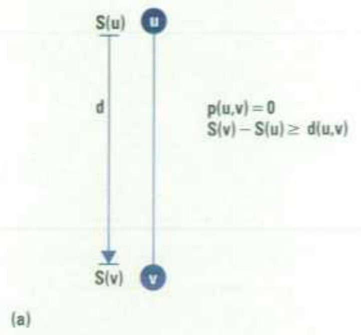


Fig. 6. A dependency graph.

* The p values are sometimes called omega values in the literature on software pipelining.



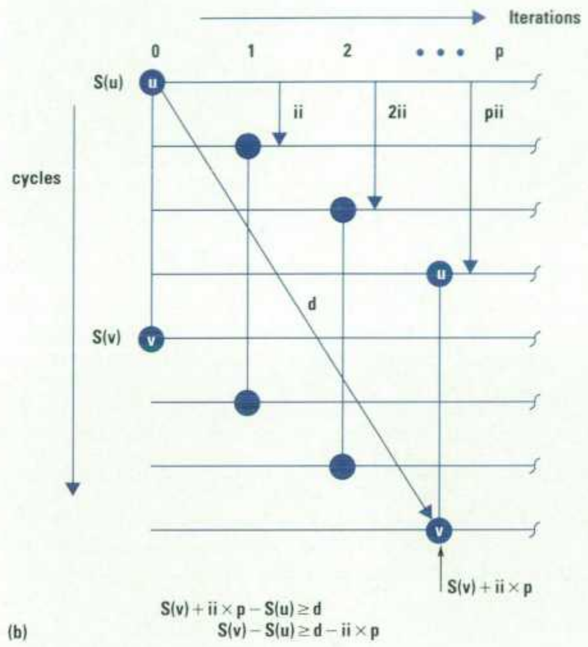


Fig. 7. (a) A dependency graph showing the schedule function when p(u,v)=0. (b) An illustration of the schedule function when p(u,v)>0.

with inter-iteration dependencies. If c is such a cycle in a graph then equation 1 becomes:

$$S(v) - S(u) \geq d(c) - ii \times p(c)$$
$$d(c) - ii \times p(c) \leq 0$$

which gives*

$$ii \geq \left\lceil \frac{d(c)}{p(c)} \right\rceil$$

RMII is the maximum value of ii for all cycles c in a graph.

4. Determine the minimum initiation interval min_ii, which is given by: min_ii = max(MII, RMII).

5. Determine the maximum initiation interval, max_ii. This is obtained by scheduling the loop without worrying about the inter-iteration dependences. The length of such a schedule gives max_ii. If we go back to our first example, we can see that we had a schedule length of eight cycles. There is no advantage if we initiate new iterations eight or more cycles apart. Therefore, eight would be a proper upper bound for the initiation interval in that example.

6. Determine the value of ii by iterating in the range [min_ii, max_ii].

6.1 For each value, ii, in the range min_ii to max_ii do the following:

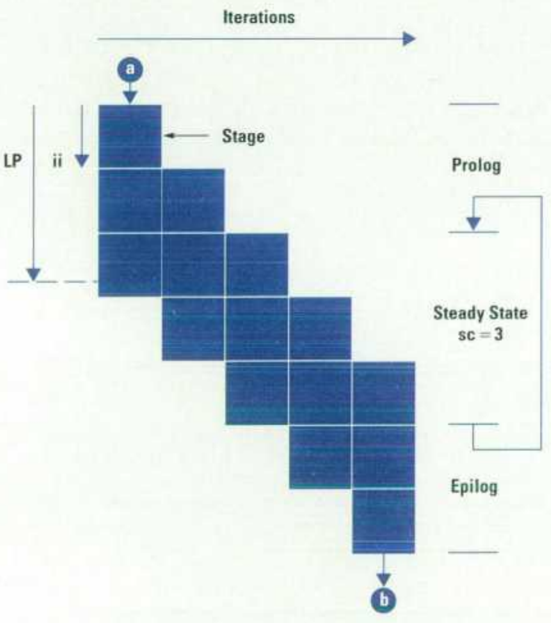
6.2 Pick an unscheduled node x from the dependency graph and attempt to schedule it by honoring its precedence and resource constraints. If it cannot be scheduled within the current ii, increment ii and start over.

If the unscheduled node can be scheduled at cycle m, set S(x) = m and update a global resource table with the resources consumed by node x starting at cycle m. This global resource table is maintained as a modulo ii table—that is, the table wraps around.

6.3 If all the nodes have been scheduled, a schedule assignment S(x) and a value for ii have been successfully found and the algorithm is terminated.

Otherwise, go back to step 6.2.

* $\lceil x \rceil$ represents the smallest integer that is greater than or equal to x. For example, $\lceil 5/3 \rceil = 2$ and $\lceil \sqrt{2} \rceil = 2$.



LP = Length of Pipeline Schedule

ii = Initiation Interval

a = Pipeline Entry

b = Pipeline Exit

If LP = 70 Cycles and ii = 25 cycles

Then Stage Count (sc) = $\left\lceil \frac{70}{25} \right\rceil = 3$

Fig. 8. An illustration of the parts of a pipeline used to compute the stage count.

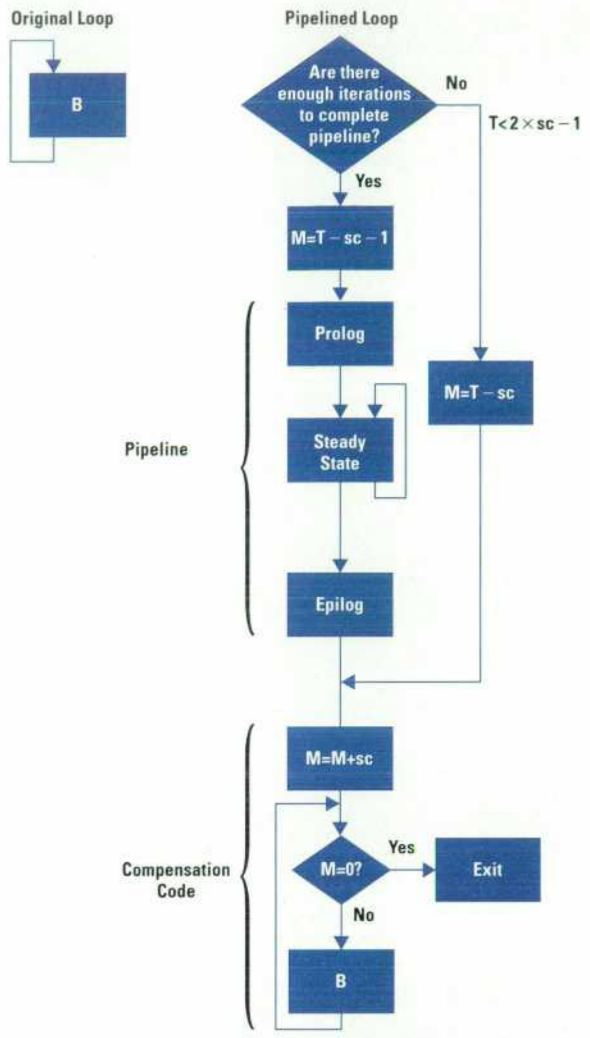


Fig. 9. Loop transformation to add compensation code to make sure the pipeline loop executes the same number of times as the original loop. Block B represents the original loop and the case in which the loop does not execute enough times to reach the pipeline.

There are three important points about this algorithm that need to be made:

- The length of a pipeline schedule (LP) may well exceed max_ii.
- Since we iterate up to max_ii, we are guaranteed that the schedule for the steady state will be no worse than a conventional schedule for the loop.
- Step 6.1 is difficult because it involves choosing the best node to schedule given a set of nodes that may be available and ready to be scheduled. The choice of a priority function that balances considerations such as the implication for register pressure if this node were to be scheduled at this cycle, the relationship of this node on the critical path, and the critical resources used by this node, is key and central to any successful scheduling strategy.

Given LP and ii, we can now determine the stage count (sc), or number of stages into which the computation in a given iteration is partitioned (see Fig. 8). This is given by:

$$sc = \left\lceil \frac{LP}{ii} \right\rceil \qquad (2)$$

The stage count also gives us the number of times we need to unroll the steady-state code. This is the unroll factor mentioned earlier.

There are two observations about equation 2. First, to guarantee execution of the prolog, steady-state, and epilog portions of the code, there must be at least $(2 \times sc) - 1$ iterations. Second, once the steady-state code is entered, there must be at least sc iterations left. In the pipeline diagram shown in Fig. 8 there must be at least five iterations to get through the pipeline (to get from a to b), and there must be three iterations left once the steady-state portion of the pipeline is reached.

Several different ways are available for generating the compensation code necessary to ensure the above conditions. For example, consider the following simple loop.

```
for ( i = 1; i <= T; i++ ) {
   B;
}
```

Here B represents some computation that does not involve conditional statements. This loop is transformed into:

```
    T = number of times loop executes
    if ( T < 2 * sc − 1) then       /*Are there enough iterations
    M =T − sc;                        *to enter  the pipeline?
    goto jump_out;                    *No.                    */
    end if;

    M = T − (sc −1)                 /*sc−1 iterations are completed
                                      *in the prolog and epilog    */

    prolog;
    steady_state;                   /*Each iteration of the steady state
                                      *decrements M by sc and the steady
                                      *state terminates when M < 0        */
    epilog;

    M = M + sc
jump_out:

    for (i = 1; i <= M; i++)     {  /*Compensation code executes
                                      * M times. If M = 0, this
                                      * loop does not execute      */
    }
```

This transformation is shown in Fig. 9. The compensation code in this code segment ensures that all the loop iterations specified in the original code are executed. For example, if T = 100 and sc = 3, the number of compensation (or cleanup) iterations is 2. As illustrated in Fig. 10, the prolog and epilog portions take care of 2 iterations, the compensation code takes care of 2 iterations, and the steady-state portion handles the remaining 96 iterations (which means that the unrolled steady state executes 32 times).

A Compiled Example

Software pipelining is supported on the HP 9000 Series 700 and 800 systems via the −O option. Currently, loops that are small (have fewer than 100 instructions) and have no control flow (no branch or call statements) are considered for pipelining.

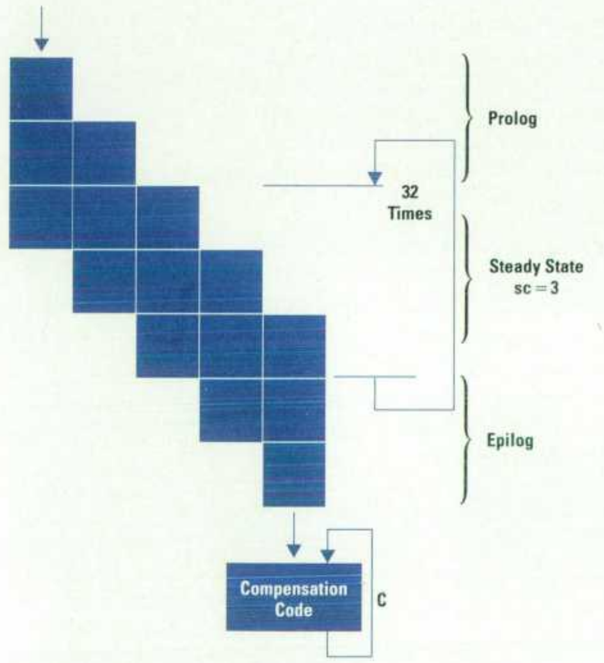
The following example was compiled by the PA-RISC FORTRAN compiler running on the HP-UX 8.0 operating system:

```
      do 10 i = 1, 1000
        z(i) = (( x(i) * y(i)) + a) * b
   10 continue
```

where x, y, z, a, and b are all double-precision quantities. The compensation code is not shown in the following example.

Without pipelining, the PA-RISC code generated is:

```
$00000015
      FLDDX,S %29(0,%23),%fr4        ; fr4 = x(i);
      FLDDX,S %29(0,%26),%fr5        ; fr5 = y(i);
;                    ## 1 cycle Interlock
      FMPY,DBL    %fr4,%fr5,%fr6     ; fr6 = x(i)*y(i);
;                    ## 2 cycle Interlock
      FADD,DBL    %fr6,%fr22,%fr7    ; fr7 = x(i)*y(i)+a;
;                    ## 2 cycle Interlock
      FMPY,DBL    %fr7,%fr23,%fr8    ; fr8 = (x(i)*y(i)+a)*b
;                  ## 1 cycle Interlock
      FSTDX,S %fr8,%29(0,%25)        ; store z(i)
      LDO    1(%29),%29              ; increment i
      COMB,<=,N    %29,%6,$00000015+4
      FLDDX,S %29(0,%23),104         ; r104 = x(i+1);
```



T = 100 Iterations
Prolog + Epilog = 2 Iterations
Stage Count = 3
Compensation Iterations C = (T + 1 − sc) Mod sc = 2 (Only If Pipeline Executes)

Iterations in Steady State = (100 − (Prolog + Epilog + C))/sc
= (100 − 4)/3 = 32

Fig. 10. An illustration of how loop iterations are divided among the portions of a pipelined loop.

If we assume perfect memory accesses, each iteration takes 14 cycles. Since there are six cycles of interlock, the CPU is stalled 43% of the time.

Using the pipelining techniques of loop unrolling and instruction scheduling to avoid pipeline stalls, the PA-RISC code generated is:

```
L$9000
      COPY  %r26,%r20                 ; r20 = i + 2;
      LDO   1(%r26),%r26              ; r26 = i + 3;
      FLDDX,S %r20(0,%r23),%fr11      ; fr11 = x(i+2);
      FLDDX,S %r20(0,%r25),%fr12      ; fr12 = y(i+2);
      FADD,DBL     %fr5,%fr22,%fr13   ; fr13 = x(i+1)*y(i+1)+a;
      FMPY,DBL     %fr11,%fr12,%fr7   ; fr7 = x(i+2)*y(i+2);
      FSTDX,S %fr4,%r29(0,%r24)       ; store z(i) Result;
      FMPY,DBL     %fr13,%fr23,%fr8   ; fr8 = (x(i+1)*y(i+1)+a)*b
      COPY  %r26,%r29                 ; r29 = i + 3;
      LDO   1(%r26),%r26              ; r26 = i + 4;
      FLDDX,S %r29(0,%r23),%fr9       ; fr9 = x(i+3);
      FLDDX,S %r29(0,%r25),%fr10      ; fr10 = y(i+3);
      FADD,DBL     %fr7,%fr22,%fr14   ; fr14 = x(i+2)*y(i+2)+a;
      FMPY,DBL     %fr9,%fr10,%fr6    ; fr6 = x(i+3)*y(i+3);
      FSTDX,S %fr8,%r19(0,%r24)       ; store z(i+1) Result;
      FMPY,DBL     %fr14,%fr23,%fr9   ; fr9 = (x(i+2)*y(i+2)+a)*b
      COPY  %r26,%r19                 ; r19 = i + 4;
      LDO   1(%r26),%r26              ; r26 = i + 5;
      FLDDX,S %r19(0,%r23),%fr8       ; fr8 = x(i+4);
      FLDDX,S %r19(0,%r25),%fr11      ; fr11 = x(i+4);
      FADD,DBL     %fr6,%fr22,%fr7    ; fr7 = x(i+3)*y(i+3)+a;
      FMPY,DBL     %fr8,%fr11,%fr5    ; fr8 = x(i+4)*y(i+4);
      FSTDX,S %fr9,%r20(0,%r24)       ; store z(i+2) Result;
      FMPY,DBL     %fr7,%fr23,%fr4    ; fr4 = (x(i+3)*y(i+3)+a)*b
      COMB,<=,N %r26,%r4,L$9000+4     ; are we done?
      COPY  %r26,%r20
```

This loop produces three results every 26 cycles which means that an iteration completes every 8.67 cycles. Since there are no interlock cycles we have 100% CPU utilization in this loop. Since it takes 14 cycles per iteration without pipelining, there is a speedup of approximately 38% in cycles per iteration with pipelining.

Another optimization technique provided in PA-RISC compilers, called register reassociation, can be used with software pipelining to generate better code because during steady state it uses different base registers for each successive iteration. See the article on page 33 for more on register reassociation.

Acknowledgements
The author would like to thank Manoj Dadoo, Monica Lam, Meng Lee, Bruce Olsen, Bob Rau, Mike Schlansker, and Carol Thompson for their assistance with this project.

References

1. B.R. Rau, et al, "The Cydra-5 Departmental Supercomputer," *Computer*, January 1989, pp. 12-35.
2. M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988, pp. 318-328.

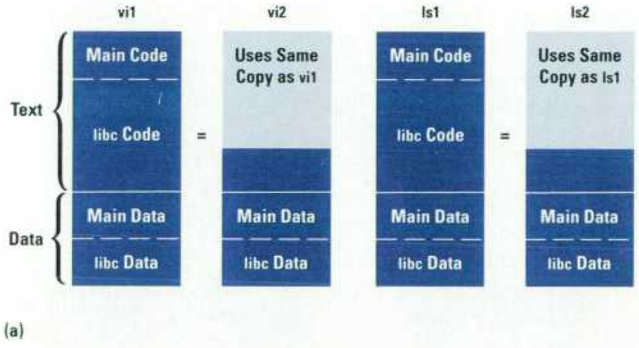
Shared Libraries for HP-UX

Transparency is the main contribution of the PA-RISC shared library implementation. Most users can begin using shared libraries without making any significant changes to their existing applications.

by Cary A. Coutant and Michelle A. Ruscetta

Multiprogramming operating systems have long had the ability to share a single copy of a program's code among several processes. This is made possible by the use of pure code, that is, code that does not modify itself. The compilers partition the program into a code segment that can be protected against modification and a data segment that is private to each process. The operating system can then allocate a new data segment to each process and share one copy of the code segment among them all.

This form of code sharing is useful when many users are each running the same program, but it is more common for many different programs to be in use at any one time. In this case, no code sharing is possible using this simple scheme. Even two vastly different programs, however, are likely to contain a significant amount of common code. Consider two FORTRAN programs, each of which may contain a substantial amount of code from the FORTRAN run-time library—code that could be shared under the right circumstances.



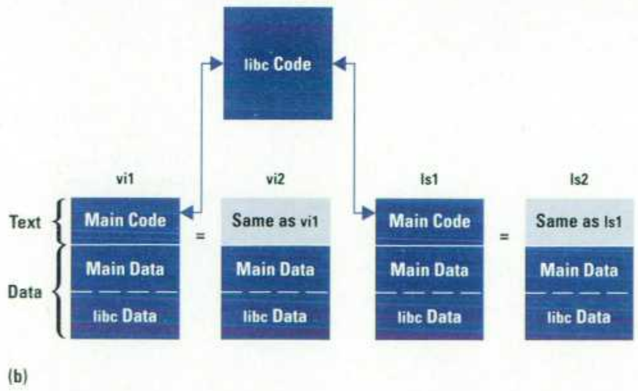


Fig. 1. Library implementations. (a) Archive library in which each program has its own copy of the library code. (b) A shared library implementation in which one copy of the library is shared between programs.

A shared library is a collection of subroutines that can be shared among many programs. Instead of containing private copies of the library routines it uses, a program refers to the shared library. With shared libraries, each program file is reduced in size by the amount of library code that it uses, and virtual memory use is decreased to one copy of each shared library's code, rather than many copies bound into every program file.

Fig. 1a shows a library scheme in which each program contains a private copy of the library code (libc). This type of library implementation is called an archive library. Note that the processes vi1 and vi2 share the same copy of the text segment, but each has its own data segment. The same is true for ls1 and ls2. Fig. 1b shows a shared library scheme in which one copy of the library is shared among several programs. As in Fig. 1a, the processes share one copy of their respective text segments, except that now the library portion is not part of the program's text segment.

Shared libraries in the HP-UX* operating system were introduced with the HP-UX 8.0 release which runs on the HP 9000 Series 300, 400, 700, and 800 workstations and systems. This feature significantly reduces disk space consumption, and allows the operating system to make better use of memory. The motivation and the design for shared libraries on the Series 700 and 800 PA-RISC workstations and systems are discussed in this article.

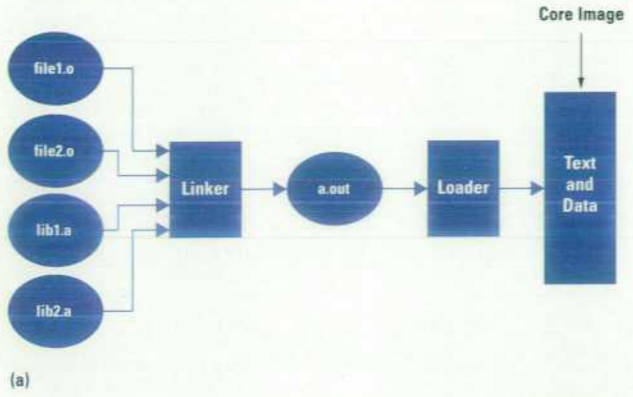
How Shared Libraries Work

Traditional libraries, now distinguished as relocatable or archive libraries, contain relocatable code, meaning that the linker can copy library routines into the program, symbolically resolve external references, and relocate the code to its final address in the program. Thus, in the final program, references from the program to library routines and data are statically bound by the linker (Fig. 2a).

A shared library, on the other hand, is bound to a program at run time (Fig. 2b). Not only must the binding preserve the purity of the library's code segment, but because the binding is done at run time, it must also be fast.

With these constraints in mind, we consider the following questions:

1. How does the program call a shared library routine?



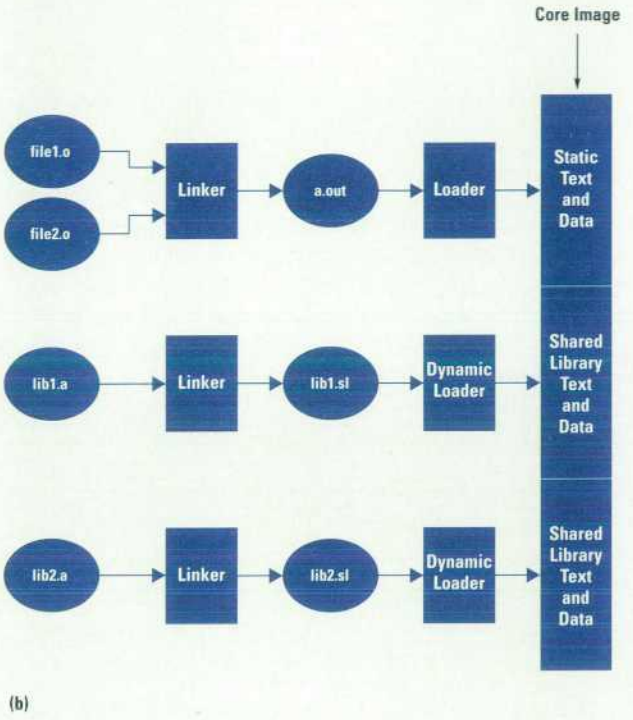


Fig. 2. Binding libraries to programs. (a) In relocatable or archive libraries, the linker binds the program's .o files and the referenced library files to create the executable a.out file. When a.out is run the loader creates the core image and runs the program. (b) For shared libraries, the linker creates an incomplete executable file (the library routines are not bound into the a.out file at link time). The shared library routines are dynamically loaded into the program's address space at run time.

2. How does the program access data in the shared library?

3. How does a shared library routine call another routine in the same library?

4. How does a shared library routine access data in the same library?

5. How does a shared library routine call a routine in another shared library (or in the program)?

6. How does a shared library routine access data in another shared library (or in the program)?

Linkage Tables. These questions can be answered several ways. The simplest technique is to bind each shared library to a unique address and to use the bound addresses of the library routines in each program that references the shared library. This achieves the speed of static binding associated with archive libraries, but it has three significant disadvantages: it is inflexible and difficult to maintain from release to release, it requires a central registry so that no two shared libraries (including third-party libraries) are assigned the same address, and it assumes infinite addressing space for each process.

Instead, we use entities called linkage tables to gather all addresses that need to be modified for each process. Collecting these addresses in a single table not only keeps the code segment pure, but also lessens the cost of the dynamic binding by minimizing the number of places that must be modified at run time.

All procedure calls into and between shared libraries (questions 1 and 5) are implemented indirectly via a procedure linkage table (PLT). In addition, procedure calls within a shared library (question 3) are done this way to allow for preemption (described later). The program and each shared library contain a procedure linkage table in their data segments. The procedure linkage table contains an entry for each procedure called by that module (Fig. 3).

Similarly, a shared library accesses its data and other libraries' data (questions 4 and 6) through a data linkage table (DLT). This indirection requires the compilers to generate indirect loads and stores when generating code

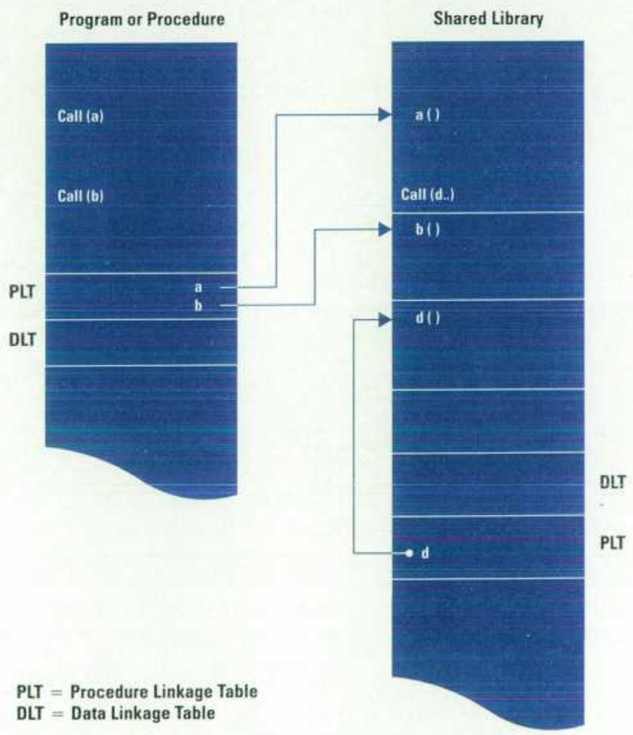


Fig. 3. Linkage tables provide the link between a program or procedure and the shared library routines. The procedure linkage table (PLT) contains pointers to routines referenced from a program, a procedure, or a shared library routine. The data linkage table (DLT) contains pointers that provide a shared library with access to its own data as well as other libraries' data.

for a shared library, which means that shared library routines must be compiled with the appropriate compiler option.

Indirect access to data is costly because it involves an extra memory reference for each load and store. We did not want to force all programs to be compiled with indirect addressing for all data, nor did we want the compilers attempting to predict whether a given data reference might be resolved within the program itself, or within a shared library.

To deal with these data access issues we chose to satisfy all data references from the program to a shared library (question 2) by importing the data definitions from the shared libraries statically (that is, at link time). Thus, some or all of a shared library's data may be allocated in the program's data segment, and the shared library's DLT will contain the appropriate address for each data item.

Binding Times. To bind a program with the shared libraries it uses, the program invokes a dynamic loader before it does anything else. The dynamic loader must do three things:

1. Load the code and data segments from the shared libraries into memory

2. Resolve all symbolic references and initialize the linkage tables

3. Modify any absolute addresses contained within any shared library data segments.

Step 1 is accomplished by mapping the shared library file into memory. Step 2 requires the dynamic loader to examine the linkage tables for each module (program and shared libraries), find a definition for each unsatisfied reference, and set the entries for both the data and procedure linkage tables to the appropriate addresses. Step 3 is necessary because a shared library's data segment may contain a pointer variable that is supposed to be initialized to the address of a procedure or variable. Because these addresses are not known until the library is loaded, they must be modified at this point. (Modification of the code segment would make it impure, so the code segment must be kept free of such constructs.)

Step 2 is likely to be the most time-consuming, since it involves many symbol table lookups. To minimize the startup time associated with programs that use shared libraries, we provide a mechanism called deferred binding. This allows the dynamic loader to initialize every procedure linkage table entry with the address of an entry point within the dynamic loader. When a shared library procedure is first called, the dynamic loader will be invoked instead, at which time it will resolve the reference, provide the actual address in the linkage table entry, and proceed with the call. This allows the cost of binding to be spread out more evenly over the total execution time of the program, so it is not noticed. An immediate binding mode is also available as an option. Deferred and immediate binding are described in more detail later in this article.

Position Independent Code. Because it is essential to keep a shared library's code segment pure, and we don't know where it will be loaded at run time, shared libraries must be compiled with position independent code. This term means that the code must not have any dependency on either its own location in memory or the location of any data that it references. Thus, we require that all branches, calls, loads, and stores be either program-counter (pc) relative or indirect via a linkage table. The compilers obey these restrictions when invoked with the +z option. However, assembly-code programmers must be aware of these restrictions.

Branches within a procedure and references to constant data in the code segment are implemented via pc-relative addressing modes. The compiler generates pc-relative code for procedure calls, but the linker then creates a special-purpose code sequence called a stub, which accesses the procedure linkage table. Loads and stores of variables in the data segment must be compiled with indirect addressing through the data linkage table.

The linkage tables themselves must also be accessible in a position independent manner. For the PA-RISC architecture, we chose to use a dedicated register to point to the current procedure and data linkage tables (which are adjacent), while on the Motorola 68000 architecture, we use pc-relative addressing to access the linkage tables.

Shared Library Trade-offs

The motivation for shared libraries is that program files are smaller, resulting in less use of disk space, and library code is shared, resulting in less memory use and better cache and paging behavior. In addition, library updates automatically apply to all programs without the need to recompile or relink.

However, these benefits are accompanied by costs that must be considered carefully. First, program startup time is increased because of the dynamic loading that must take place. Second, procedure calls to shared library routines are more costly because of the linkage table overhead. Similarly, data access within a shared library is slower because of the indirect addressing. Finally, library updates, while seeming attractive on the one hand, can be a cause for concern on the other, since a newly introduced bug in a library might cause existing applications to stop working.

Design Goals for HP-UX Shared Libraries

When we first began designing a shared library facility for the HP-UX operating system, AT&T's System V Release 3 was the only UNIX* operating system implementation of shared libraries. Sun Microsystems released an implementation in SunOS shortly afterwards.[1] We also investigated a few other models including: Multics,[2,3] VAX/VMS,[4] MPE V and MPE XL,[5] AIX,[6] and Domain/OS.[7] While AT&T's scheme requires static binding as well as a mechanism for building shared libraries, the others are all based on some combination of indirection and position independent code.

None of the existing models offered what we considered to be our most important design goal—transparency. We felt that the behavior of shared libraries should match the behavior of archive libraries as closely as possible, so

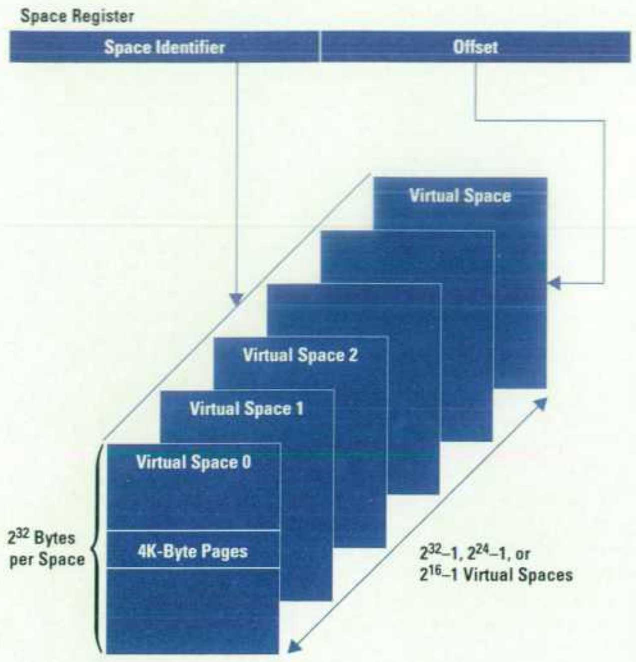


Fig. 4. The PA-RISC virtual memory architecture.

that most programmers could begin using shared libraries without changing anything. In addition, the behavior of most shared library implementations with respect to precedence of definitions differs dramatically from archive library behavior. If an entry point is defined in both the program and an archive library, only the definition from the program would be used in the program, and calls to that routine from the library would be bound to the definition in the program, not the one in the library. Such a situation is called *preemption* because the definition in the program preempts a definition of the same name in the library, and all references are bound to the first definition.

Another design goal that we followed was that the dynamic loader must be a user-level implementation. The only kernel support we added was a general memory-mapped file mechanism that we use to load shared libraries efficiently. The dynamic loader itself is a shared library that is bootstrapped into the process's address space by the startup code.

We also wanted to ease the task of building shared libraries. We explicitly avoided any design that would require the programmer to modify library source code or follow a complicated build process.

Finally, we recognized that, in the absence of an obvious standard, our shared library model should not be significantly different from other implementations based on AT&T's System V Release 4.

PA-RISC Design Issues

Although the HP-UX shared library implementation is designed to have the same external interface and behavior in the HP 9000 Series 300, 800, and 700 systems, restrictions imposed by the PA-RISC systems (Series 700 and 800 systems) posed some interesting design considerations that resulted in additional complexity in the underlying implementation. One of the main restrictions is based on the PA-RISC software architecture for virtual memory and the lack of a facility in the operating system to handle the situation.

Virtual memory in PA-RISC is structured as a set of address spaces each containing 2^{32} bytes (see Fig. 4).[8] A virtual address for a processor that supports a 64-bit address is constructed by the concatenation of the contents of a 32-bit register called a space register and a 32-bit offset. The PA-RISC software architecture divides each space into four 1G-byte quadrants, with four space registers (sr4 to sr7) assigned to identify a quadrant (see Fig. 5). This scheme requires that text and data be loaded into separate spaces and accessed with distinct space pointers. Program text is accessed using sr4, shared library text is accessed using sr6, and all data for shared libraries and program files is accessed using sr5. This architecture does not allow contiguous mapping of text and data in an executable file.* Therefore, to handle shared libraries in PA-RISC we had to have a dedicated linkage table pointer register and provide support for interspace procedure calls and returns.

Dedicated Linkage Table Pointer. Since code and data could not be mapped contiguously, the linkage tables could not be accessed with a pc-relative code sequence generated at compile time. Therefore, we chose a general register (gr19) as a place for holding the pointer for shared library linkage. All position independent code and data references within a shared library go indirectly through the gr19 linkage register. Code in a main program accesses the linkage table directly since the main program code is not required to be position independent.

Position independent code generation for shared libraries must always consider the gr19 linkage register as being live (in use), and must save and restore this register across procedure calls.

The plabel. The dedicated linkage table pointer added complexity to the design for handling procedure labels and indirect procedure calls. Two items in the PA-RISC software architecture had to be modified to include information about the linkage table pointer: a function pointer called a plabel (procedure label), which is used in archive HP-UX libraries and programs, and a millicode routine called $$dyncall, which is used when making indirect function calls. To support this new plabel definition the following changes had to be made.

- In programs that use shared libraries, a plabel value is the address of the PLT entry for the target routine, rather than a procedure address. An HP-UX PA-RISC shared library plabel is marked by setting the second-to-last low-order bit of the plabel (see Fig. 6).
- The $$dyncall routine was modified to use this PLT address to obtain the target procedure address and the target gr19 value. In the modified implementation, the $$dyncall routine and the kernel's signal-handling code check to see if the HP-UX shared library plabel bit is set, and if so, the library procedure's address and linkage table pointer values can be obtained using the plabel value.

* The HP 9000 Series 300 systems do support contiguously mapped text and data.

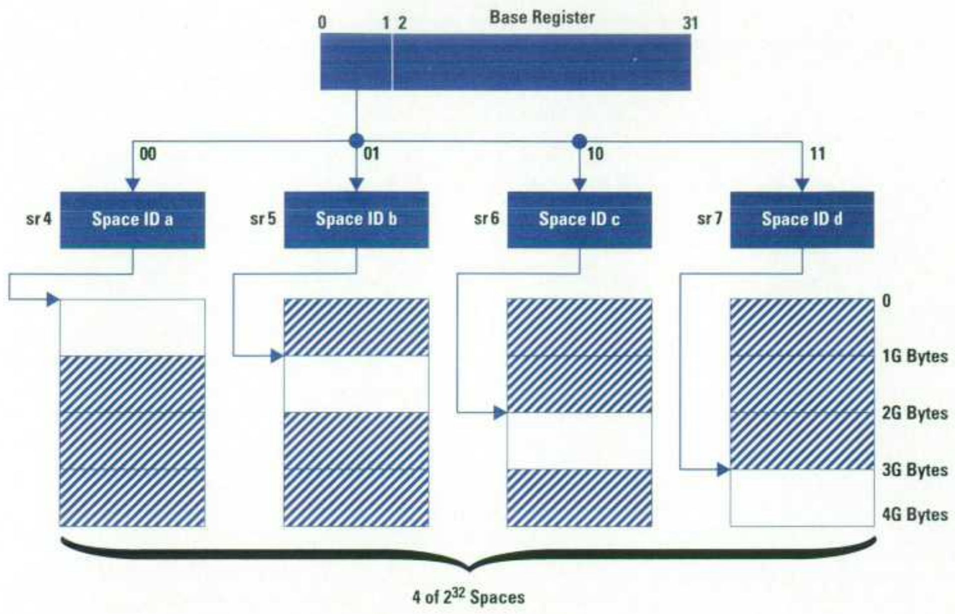
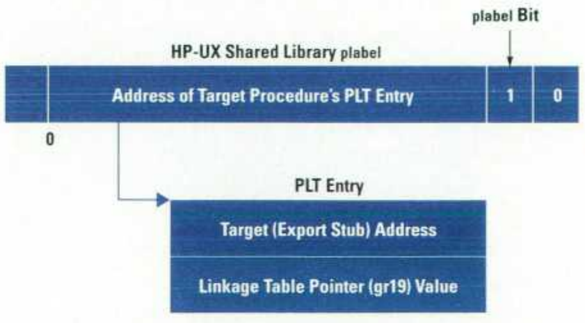


Fig. 5. The relationship of space registers sr4, sr5, sr6, and sr7 to the virtual address spaces.

Interspace Calls and Returns. The second significant impact on the shared library design was the need for a way to handle interspace calls and returns because in the PA-RISC software architecture, program text and shared memory text are mapped into separate spaces.

The default procedure call sequence generated by the HP-UX compilers consists of intraspace branches (BL instruction) and returns (BV instruction). The compilers assume that all of a program's text is in the same virtual space. To perform interspace branches, an interspace call and return sequence is required. The call and return sequence for an interspace branch is further complicated by the fact that the target space is not known at compile time, so a simple interspace branch instruction (BLE offset(srX,base)) is not sufficient. Instead, a code sequence that loads the target space into a space register and then performs an interspace branch is required.

The HP-UX memory map implementation mmap() is used for mapping shared library text. As mentioned earlier, all shared library text is mapped into the sr6 space (quad 3 addresses) and all data is mapped into the sr5 space (quad 2 addresses). This mapping, along with the need to have a dedicated position independent code linkage register, requires special code sequences to be produced for each function in the library. These code sequences are referred to as stubs. The linker places stubs into the routine making the call and in the library routines (and program files) being called to handle saving and restoring the gr19 linkage register and performing the interspace branch (see Fig. 7). As mentioned above, compilers generate an intraspace branch (BL) and an intraspace return (BV) for procedure call sequences. The linker patches the BL to jump to the import stub code (① in Fig. 7), which then performs the interspace branch to the target routine's export stub (② in Fig. 7). The export stub is used to trap the return from the call, restore the original return pointer and execute an interspace branch.



plabel Bit
HP-UX Shared Library plabel
Address of Target Procedure's PLT Entry
1
0
0
31
PLT Entry
Target (Export Stub) Address
Linkage Table Pointer (gr19) Value

Fig. 6. A shared library plabel and PLT entry.

HP-UX User Interface

The HP-UX shared library design offers various user interface routines that provide capabilities to dynamically load and unload libraries, to define symbols, and to obtain information about loaded libraries and symbol values. All of these user interface routines are designed to be used by a user-level program to control the run-time loading and binding of shared libraries.

Library Loading. Shared libraries can be loaded either programmatically (explicit loading) or via arguments in the link command line (implicit loading). Explicit loading and unloading are provided through the shl_load() and shl_unload() routines. Libraries specified for implicit loading are mapped into memory at program startup. There are two main binding modes for loading shared libraries: immediate binding and deferred binding. For implicit loading the binding modes can be specified on the link command line using the –B immediate or –B deferred linker command line options. The default mode for implicit shared libraries is deferred binding. For explicit loading the binding mode is specified by using the BIND_IMMEDIATE or BIND_DEFERRED flag in shl_load()'s argument list.

The deferred binding mode will bind a code symbol when the symbol is first referenced, and will bind all visible data symbols on program startup. The data symbols must be bound when the library is loaded since there is no mechanism for trapping a data reference in PA-RISC

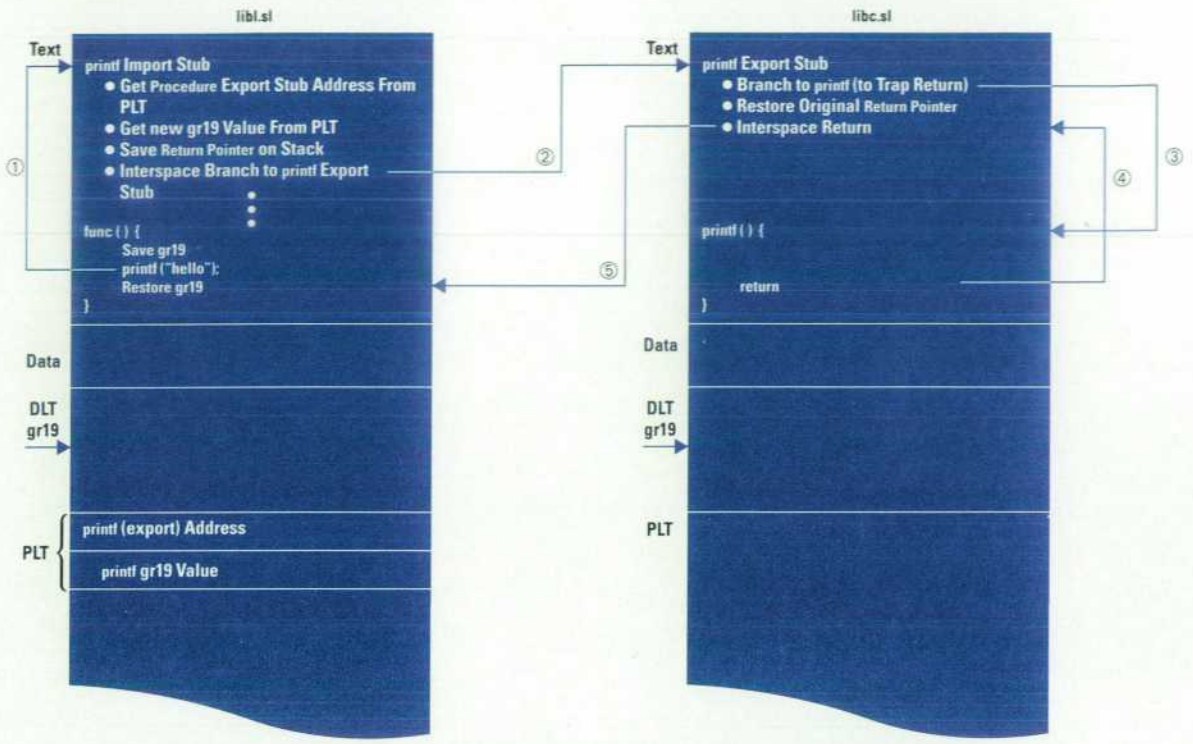


Fig. 7. Shared library procedure calls.

(see "Deferred Binding, Relocation, and Initialization of Shared Library Data" on page 52 for more details). The deferred binding mode spreads the symbol-binding time across the life of the program, and will only bind procedures that are explicitly referenced.

The immediate binding mode binds all data and code symbols at program startup. If there are any unresolved symbols, a fatal error is reported and loading is not completed. The program will abort if unresolved symbols are detected while loading any implicitly loaded libraries when immediate binding is used. The immediate binding mode forces all of the symbol binding to be done on startup, so the binding cost is paid only at startup. The immediate binding mode also has the advantage of determining unresolved symbol references at startup. (In deferred binding mode the program could be running for some time before an unresolved symbol error is detected.)

Additional flags are available for explicitly loading shared libraries that alter the behavior of the immediate and deferred binding modes. These flags provide the user with some control over the binding time and binding order of shared library symbols, and are used in conjunction with the BIND_IMMEDIATE and BIND_DEFERRED flags.

- BIND_FIRST. This option specifies that the library should be placed at the head of the library search list before the program file. The default is to have the program file at the head of the search list and to place additional shared libraries at the end of the search list. All library searching is done from left (head) to right (tail).
- BIND_NONFATAL. When used with the BIND_IMMEDIATE flag, this flag specifies that if a code symbol is not found at startup time, then the binding is deferred until that code symbol is referenced (this implies that all unresolved code symbols will be marked deferred with no error or warning given, and all unresolved data symbols will produce an error). The default immediate binding behavior is to abort if a symbol cannot be resolved. This option allows users to force all possible binding to be done at startup, while allowing the program file to reference symbols that may be undefined at startup but defined later in the execution of the program.
- BIND_NOSTART. This flag specifies that the shared library initializer routine should not be called when the library is loaded or unloaded. The initializer routine is specified using the +I linker option when the shared library is built. Default behavior is for the dynamic loader to call the initializer routine, if defined, when the shared library is loaded.

Deferred Binding, Relocation, and Initialization of Shared Library Data

In most shared library implementations, including the HP-UX implementation, a shared library can be loaded at any address at run time. While position independent code is used for library text, library data must be relocated at run time after a load address has been assigned to the library. Also, because addresses of symbols defined by shared libraries are not known until run time, references from application programs to shared libraries cannot be bound to correct virtual addresses at link time, nor can references between shared libraries or from shared libraries to application programs be resolved at library build time. Instead, all such references are statically resolved to linkage tables. Each entry in a linkage table corresponds to a specific symbol. When a program that uses shared libraries is executed, the loader must initialize each linkage table entry with the address of the corresponding symbol.

Furthermore, languages such as C++ support run-time initialization of data. A class can have a constructor, which is a function defined to initialize objects of that class. The constructor is executed when an object of that class is created. C++ mandates that the constructors for nonlocal static objects in a translation unit* be executed before the first use of any function or object defined in that module. Other languages, such as Ada, may have similar run-time initialization requirements.

The dynamic loader must therefore perform relocation, binding, and initialization at run time. Linkage table entries for function calls can be initialized to trap into the dynamic loader, so that the binding of a function reference can be deferred until the first call through the reference. On the other hand, data references cannot be trapped in this manner on most architectures. Thus, in most shared library implementations, the dynamic loader must perform all relocation, binding, and initialization of data for an entire library when that library is loaded. This normally implies a high startup cost for programs that use shared libraries.

Module Tables

The HP-UX design conceptually maintains some of the boundaries between the modules that make up a shared library. All export table entries, linkage table entries, relocation records, and constructors are grouped by translation unit into module tables. The dynamic loader defers the binding, relocation, and initialization of data for a module until the first potential access of any symbol defined in that module. This greatly reduces the startup overhead of programs that use shared libraries.

Since the Series 700 architecture does not support trapping on specific data references, the dynamic loader cannot directly detect the first access of a given data symbol. Instead, the dynamic loader considers a given data symbol to be potentially accessed on the first call to any function that references the symbol. Rather than actually keeping track of which functions reference which data symbols, the module table allows the dynamic loader to make a further approximation. On the first call to a given function, the dynamic loader considers the whole module to have been potentially accessed. It consults the module table to determine which linkage table entries to bind, which relocation records to apply, and which constructors to execute.

This algorithm is recursive, since binding linkage table entries, relocating data, and executing constructors all may reference symbols in other modules. These modules must also be considered to be potentially accessed. The dynamic loader must therefore bind, relocate, and initialize data in those modules as well. If libraries typically contain long chains of data references between modules, then this algorithm will be processing data for many modules on the first call to a given library function. If the library is completely connected by such references, this algorithm degenerates into binding, relocating, and initializing all data for an entire library the first time any function in that library is called. However, our experience shows that typical libraries seldom have chains more than three or four modules long, and many programs access only a fraction of the total number of modules in a library. Deferring the binding, relocation, and initialization of data on a module basis has shown that the time spent performing these tasks can be reduced by 50% to 80%, depending on the program and libraries involved.

Further C++ Considerations

The C++ definition of static destructors adds another complication to the design. A destructor for an object is executed when the object is destroyed. Static objects are considered destroyed when the program terminates. C++ mandates that destructors for static objects be called in reverse order from the constructors. Other languages may have different semantics. Therefore, the dynamic loader employs a more general technique. Rather than execute constructors directly when processing data for a module, the dynamic loader executes a function called an elaborator, which is defined by the C++ run-time support code. The C++ elaborator executes all static constructors for the module and also inserts any corresponding destructors at the head of a linked list. On program termination, the C++ run-time support code traverses this list and executes all destructors.

The HP-UX shared library design also supports explicit loading and unloading of shared libraries from within a program via the shl_load and shl_unload functions described in the accompanying article on page 50. While C++ does not define any particular semantics for dynamic loading and unloading of libraries, it seems natural to execute static destructors for objects defined in an explicitly loaded library when the library is unloaded. Since the destructors for objects defined in a library are often defined in the library itself, the dynamic loader clearly cannot wait until program termination to execute destructors for objects in libraries that have already been unloaded. Therefore, the dynamic loader invokes a library termination function when a library is unloaded. This function, also defined by the C++ run-time support system, traverses the linked list of destructors and executes all destructors for the library being unloaded. It then removes those destructors from the list. For symmetry, the dynamic loader also invokes an initialization function when a library is loaded, implicitly or explicitly, but this capability is not used by the C++ implementation.

Marc Sabatella
Software Development Engineer
Systems Technology Division

* A static object is an object that lives throughout the life of the program, and a translation unit is the source file produced after going through the C++ preprocessor.

- BIND_VERBOSE. This flag causes messages to be emitted when unresolved symbols are discovered. Default behavior in the immediate bind mode performs the library load and bind silently and returns error status through the return value and errno variable.

Other user interface routines are provided for obtaining information about libraries that have already been loaded.

- shl_get(). This routine returns information about currently loaded libraries, including those loaded implicitly at startup time. The library is specified by the index, or ordinal position of the shared library in the shared library search list. The information returned includes the library handle, pathname, initializer address, text start address, text end address, data start address, and data end address.
- shl_get_handle(). This routine returns the same information as the shl_get() routine, but the user specifies the library of interest by the library handle rather than the search-order index. Typically, the shl_get() routine would be used when a user wants to traverse through the list of libraries in search order, and the shl_get_handle() routine can be used to get information about a specific library for which the library handle is known (i.e., explicitly loaded libraries).

Dynamic Symbol Management. User interface routines provided for dynamic symbol management include shl_findsym() and shl_definesym(). The shl_findsym() routine is used to obtain the addresses of dynamically loaded symbols so that they can be called. The shl_findsym() interface is the only supported way of calling dynamically loaded routines and obtaining addresses for dynamically loaded data items. The shl_definesym() routine allows the user to dynamically define a symbol that is to be used in future symbol resolutions. The user provides the symbol name, type, and value. If the value of the symbol falls within range of a library that has previously been loaded, then the newly defined symbol is associated with that library and will be removed when the associated library is unloaded.

Other Features

Other features that are specific to the HP-UX shared library implementation include a module-level approach to version control, a method of symbol binding that reduces the shared library startup cost for programs that use a small percentage of the library routines, and special C++ support. The special C++ support is described in the short article on the previous page.

Version control. One of the advantages of shared libraries is that when a change (e.g., a defect repair) is made to the library, all users can take immediate advantage of the change without rebuilding their program files. This can also be a disadvantage if the changes are not applied carefully, or if the change makes a routine incompatible with previous versions of that routine. To protect users of shared libraries, some type of version control must be provided.

The HP-UX shared library approach to version control is provided at the compilation unit (module) level, which is unlike most existing implementations that provide version control only at the library level. Our version control scheme is based on library marks that are used to identify incompatible changes. When an incompatible change is made to a routine, the library developer date-stamps the routine using a compiler source directive. The date is used as the version number and is associated with all symbols exported from that module. The resulting module can then be compiled and added to the shared library along with the previous versions of that module. Thus, the date stamp is used as a library mark that reflects the version of the library routine. When a user program file is built, the mark of each library linked with the program is recorded in the program file. When the program is run, the dynamic loader uses the mark recorded in the program file to determine which shared library symbol is used for binding. The dynamic loader will not accept any symbol definitions that have a mark higher than the mark recorded for the defining library in the program file.

This scheme can also be used for changes that are backwards compatible and for programs that rely on new behavior. In this case, library developers would include a dummy routine with a new date to force an increase in the library's mark. Any new programs linked with this library would have the new mark recorded, and if run on a system with an older version of the library, the dynamic loader will refuse to load the old library because the version number of the installed library would be lower than the number recorded in the program file.

Archive Symbol Binding. Typically, a shared library is treated as one complete unit, and all symbols within the library are bound when any symbol in that library is referenced. In the HP-UX scheme, the shared library file maintains module granularity similar to archive libraries. When the shared library is built, a data structure within the shared library is used to maintain the list of modules (compilation units) used to build the library. The list of defined symbols and referenced symbols is maintained for each module. During symbol resolution, the dynamic loader binds only symbols for modules that have been referenced. This symbol binding technique provides a significant performance improvement in the startup and symbol binding time for typical programs (i.e., programs that reference a relatively low percentage of the total symbols in the attached shared libraries).

Acknowledgments

We would like to recognize the contributions made in the design of the HP-UX shared library mechanism by Stacy Martelli, Kevin Wallace, and Eric Hamilton. Much of the implementation, testing, and debugging was done by Carl Burch and Jim Schumacher. In addition, we appreciate the efforts of engineers in other HP laboratories who helped in getting everything put together, especially Bob Schneider.

References

1. R.A. Gingell, et al, "Shared Libraries in SunOS," *USENIX Proceedings,* Summer 1987.
2. R.C. Daley and J.B. Dennis, "Virtual Memory, Process, and Sharing in Multics," *Communications of the ACM,* Vol. 11, no. 5, May 1968.
3. A. Bensoussan, et. al., "The Multics Virtual Memory: Concepts and Design," *Communications of the ACM,* Vol. 15, no. 5, May 1972.
4. L.J. Kenah and S.F. Bate, *VAX/VMS Internals and Data Structures*, Digital Press, 1984.
5. *HP Link Editor/XL Reference Manual,* Hewlett-Packard Co., November 1987.
6. M.A. Auslander, "Managing Programs and Libraries in AIX Version 3 for RISC System/6000 Processors," *IBM Journal of Research and Development,* Vol. 34, no. 1, January 1990.
7. *Domain/OS Programming Environment Reference,* Hewlett-Packard Co., October 1989.
8. M.J. Mahon, et al, "Hewlett-Packard Precision Architecture: The Processor," *Hewlett-Packard Journal,* Vol. 37, no. 8, August 1986, pp. 4-21.

Integrating an Electronic Dictionary into a Natural Language Processing System

This paper discusses the types of electronic dictionaries available and the trends in electronic dictionary technology, and provides detailed discussion of particular dictionaries. It describes the incorporation of one of these electronic dictionaries into Hewlett-Packard's natural language understanding system and discusses various computer applications that could use the technology now available.

by Diana C. Roberts

Computational linguistics is demonstrating its relevance to commercial concerns. During the past few years, not only have companies funded and carried out research projects in computational linguistics, but also several products based on linguistic technology have emerged on the market. Franklin Products has created a line of handheld calculator-like dictionaries which range from a spelling dictionary to a pronouncing dictionary with a speech generator attached to a full thesaurus. Franklin Products, Texas Instruments, Casio, and Seiko all produce multilingual handheld translating dictionaries. Many text editors and word processors provide spelling checkers and thesauruses, such as those used by WordPerfect. Grammatik IV and Grammatik Mac are widely available style and grammar checkers. Merriam-Webster and the Oxford University Press have recently released their dictionaries on CD-ROM.

Both the commercial success of these linguistics products and the promising nature of their underlying theoretical basis encourage more ambitious work in industrial research. Outside of the United States, particularly in Europe and Japan, there is great interest in machine translation, although products remain on the research level. The Toshiba Corporation has developed a Japanese-English typed-input translating system for conversational language. Within the United States, Unisys, SRI, and Hewlett-Packard† have developed natural language understanding systems with prospective applications of database inquiry and equipment control, among other areas. In the area of electronic dictionary development, both the Centre for Lexical Information (CELEX) in the Netherlands and Oxford University Press (publishers of the Oxford English Dictionary) in England are developing dictionary products that are sophisticated both in the linguistic data they contain and in the way the data is accessed.

† Hewlett-Packard's HP-NL (Hewlett-Packard Natural Language) system was under development from 1982 to 1991.[1]

The linguistics of computational linguistic theory is based on standard modern theories such as lexical functional grammar, or LFG,[2] and head-driven phrase-structure grammar, or HPSG.[3] Most of these theories assume the word to be the basic linguistic element in phrase formation, that is, they are "lexicalized" theories. Words, therefore, are specified in great linguistic detail, and syntactic analysis of sentences is based on the interplay of the characteristics of the component words of a sentence. Therefore, products based on linguistic theory such as grammar checkers and natural language understanding systems require dictionaries containing detailed descriptions of words. Products that do not involve sentential or phrasal analysis, such as spelling checkers and word analyzers, also require extensive dictionaries. Thus, dictionaries are very important components of most computational linguistic products.

Of course, the book dictionary has been a standard literary tool, and the widespread acceptance of the computer as a nontechnical tool is creating an emerging demand for standard dictionaries in electronic form. In fact, the importance of linguistically extensive dictionaries to computational linguistic projects and products is reflected in the emerging availability of electronic dictionaries during the past few years. Webster's Ninth on CD-ROM, a traditional type of dictionary now in electronic form, became available from Merriam-Webster in 1990. Longman House made the typesetting tape for its Longman's Dictionary of Contemporary English (LDOCE)

Notation and Conventions

In this article, italic type is used for natural language words cited in the text (e.g., *happy*).
The sans-serif font is used for programming keywords.
The asterisk (*) preceding a phrase indicates ungrammaticality.
The dagger (†) indicates a footnote.

available to the research community of linguists, lexicographers, and data access specialists in the mid-1980s. It is still a research tool, but has become very expensive to commercial clients, presumably reflecting its value to the research and commercial community. The products from CELEX and the Oxford University Press mentioned above are among the most sophisticated electronic dictionary products available today. The second edition of the Oxford English Dictionary, the OED2, is available on CD-ROM. Its data, in running text form, is marked explicitly in SGML (the Standard Generalized Markup Language, ISO 8879) for retrieval. The CELEX dictionary in English, Dutch, and German is a relational database of words and their linguistic behavior. These two latter products are sophisticated in very different ways and are designed for very different uses, but they have in common a great flexibility and specificity in data retrieval.

Related work that can support further lexicographical development in the future is being carried on by the Data Collection Initiative (DCI) through the Association for Computational Linguistics. This and similar initiatives are intended to collect data of various forms, particularly literary, for computer storage and access. Lexicographers are already making use of large corpora in determining the coverage for their dictionaries.[4] The availability of large corpora for statistical studies will certainly aid and may also revolutionize lexicographical work and therefore the nature of electronic dictionaries.

There is apparently a commercial market for linguistically based products, since such products are already being sold. Many of these current products either rely on electronic dictionaries or are themselves electronic dictionaries of some kind. Recent years have seen electronic dictionaries become more sophisticated, both in their content and in the accessibility of their data. Because many sophisticated products based on computational linguistics must rely on dictionary information, the potential scope of computational systems based on linguistics has increased with the improvements in electronic dictionaries.

My aim with this paper is to introduce the area of electronic dictionary technology, to suggest areas of research leading to product development that crucially exploit the emerging dictionary technologies, and to report on the results of one such effort at Hewlett-Packard Laboratories.

What Is a Dictionary?

Commonly, a dictionary is considered to be a listing by spelling of common natural language words, arranged alphabetically, typically with pronunciation and meaning information. There are, however, collections of words that violate one or more of these three stereotypical characteristics but are still considered dictionaries. For instance, the simplest kind of dictionary is the word list, used for checking spelling; it contains no additional word information. The Bildwörterbuch from Duden contains both pictures and words for each entry, and is arranged not alphabetically, but by topic. Stedman's Medical Dictionary contains alphabetically ordered technical terms from the domain of medicine and their definitions rather than common English words. It also contains some etymological information, but offers pronunciation information for only some entries. Similarly, symbol tables of compilers contain symbols used by software programs; their entries are not natural language words. Data dictionaries of database management systems also contain entries for non-natural-language words, as well as other nonstandard dictionary information such as computer programs.

If all three of the stereotypical characteristics can be violated, then for the purposes of this paper we need to establish what a dictionary is. As a start, we can appeal to a dictionary as an authority on itself. Webster's Ninth New Collegiate Dictionary (the electronic version of which is one of the dictionaries discussed in this paper) says that a dictionary is "1: a reference book containing words usu. alphabetically arranged along with information about their forms, pronunciations, functions, etymologies, meanings, and syntactical and idiomatic uses 2: a reference book listing alphabetically terms or names important to a particular subject or activity along with discussion of their meanings and applications 3: a reference book giving for words of one language equivalents in another 4: a list (as of phrases, synonyms, or hyphenation instructions) stored in machine-readable form (as on a disk) for reference by an automatic system (as for information retrieval or computerized typesetting)."

There are some common elements of these definitions, which together form the defining characteristics of the dictionary. First and most crucial, the dictionary is a listing of language elements, commonly words. Implied too is that the entries can be taken from any domain. These entries are arranged in some way to make retrieval either possible or easy. And finally, the dictionary also often contains other information associated with the entry. An electronic dictionary is any kind of dictionary in machine-readable form.

The electronic dictionaries available now vary greatly. This paper will only consider dictionaries whose entries come from the domain of natural language, and whose entries are words rather than phrases. I will discuss three dimensions along which electronic dictionaries differ from each other: type of additional information about the entry presented, the explicitness of the information categories (more explicit marking of the categories reducing ambiguity), and the accessibility and organization of the data. After the discussion of electronic dictionaries and their characteristics, I will discuss the possible uses of electronic dictionaries and the necessary characteristics of the dictionaries for the various possible uses. The purposes to which an electronic dictionary can be put depend on its characteristics in each of the three dimensions discussed in the following sections.

Evolution of Electronic Dictionaries

Early electronic dictionaries were word lists. They had a limited range of use because they contained limited types of information in a simple organization. Electronic dictionaries are becoming more complex and more flexible now, as they become potentially more useful in domains that did not exist before. The potential uses are shaping the ways in which electronic dictionaries are evolving.

As computers began to be used for writing and communication, the standard desk reference book, the dictionary, was ported to electronic form. When this reference tool became available in the same medium as the word processor—the computer—the lexicographical information was now machine-readable. As linguistically based software systems matured, the demand for accessible lexicographical data based on modern linguistic theory grew. This demand has brought several important pressures on electronic dictionaries.

Lexicographical Information. First, several newer electronic dictionaries provide extensive linguistic information based in some cases on modern linguistic theories. Word entries in traditional dictionaries often do not recognize the same categories that are important to generative linguistic theory. Traditional dictionaries focus on defining words, providing historical derivation information (etymologies), providing sample sentences to illustrate word use, and providing some basic linguistic information, such as spelling, syllabification, pronunciation, part of speech, idiomatic use, and semantically related words (synonyms, hyponyms, hypernyms, and antonyms).† This information, if it were unambiguously accessible, could be used for some software applications—for example, a spelling checker, a semantic net, or possibly speech generation or recognition, depending on the sophistication of the speech system. However, this information is insufficient for applications that require complex word and/or sentence analysis, such as natural language processing, which involves natural language understanding.

The recent expansion of electronic dictionaries coincided, not surprisingly, with the emergence of several book dictionaries of English that carefully detail linguistic word information based on modern linguistic theory. These "learning dictionaries," created for foreign learners of English rather than native speakers, contain only the most common words instead of attempting to be exhaustive. These dictionaries concentrate more on complete syntactic and morphological characterization of their entries than on exhaustive meaning explanations and citations, and use linguistic categories from modern generative linguistics in preference to traditional categories. Three of these dictionaries are Longman's Dictionary of Contemporary English (LDOCE), the Oxford Advanced Learner's Dictionary of Current English (OALD), and Collins COBUILD English Language Dictionary. Some of the most useful electronic dictionaries draw their lexicographical information from these sources.††

The following are some of the kinds of information found in electronic dictionaries, both traditional and modern:
- Orthography (spelling)
- Syllabification
- Phonology (pronunciation)
- Linguistic information about the word's properties, including syntax, semantics (meaning), and morphology (word structure)
- Related word(s)—related by morphology, either inflectional (*work, works*) or derivational (*happy, unhappy*)
- Synonym listings
- Semantic hierarchies
- Frequency of occurrence
- Meaning (not yet a robustly structured field)
- Etymology
- Usage (sample phrases and/or sentences, either created by the lexicographer or cited from texts).

Data Categorization. A second trend in newer electronic dictionaries is to represent the lexicographical data in such a way that it is unambiguously categorized, either tagged in the case of running text dictionaries, or stored in a database. Linguistically based software systems must be able to access lexicographical information unambiguously.

Traditional dictionaries rely on human interpretation of various typefaces which are often formally ambiguous to determine the category of information represented. In the entry for "dictionary" in Webster's Ninth, the italic type face is used to represent both the part-of-speech and foreign-language etymological information, and the part-of-speech indicator comes after the entry for "dictionary" and before the entry for the plural "-naries".

One of the earlier desk-type dictionaries in electronic form was the Longman's Dictionary of Contemporary English. The tape containing the typesetting information for the book form was stored electronically. Thus, all its lexicographical information was available electronically, but the data fields were ambiguously indicated through the typesetting commands.

The second edition of the Oxford English Dictionary (OED2) is available in an electronic edition on CD-ROM. This dictionary, like the LDOCE, is a running text dictionary rather than a regular database. Its data fields, however, are explicitly marked using the SGML tagging language. Here, data retrieval does not face the problem of ambiguity.

The CELEX electronic dictionary is in relational database form. This encourages uniformity in the classification system and in the data.

Accessibility of Data. A third trend is the increased accessibility to their data offered by some electronic dictionaries. Accessibility is affected by both data structure and data encryption. In some dictionaries, the entry point for retrieval is only by word spelling; in others, there are multiple entry points. The data of some dictionaries is designed intentionally to be inaccessible programmatically; in other cases, it is designed to be accessible.

In word lists such as spelling dictionaries, data organization is not extensive, usually consisting of only alphabetical ordering and indexing to allow fast access. The data in these dictionaries is fully accessible to at least the spelling software application, and may be designed in a way that it could be used by other software applications as well. For instance, the ispell dictionary is a word list in ASCII format that can be used for other purposes.

Other dictionary types vary more in their accessibility, both in whether the data is accessible at all programmatically, and in how flexible the access is. The data in the OED editions and in Webster's Ninth is encrypted (an

† In a pair of words, one of which has a broader meaning than the other, the word with the broader meaning is the hypernym and the word with the more narrow meaning is the hyponym. For example, for the words book and novel, book would be the hypernym and novel the hyponym.

†† Extensive and explicit phonetic, morphological, and syntactic information is useful now in computer applications, whereas neither semantic nor etymological information is yet structured enough to be useful.

```
10th
1st
2nd
...
a
A&P
a's
AAA
aardrup
aardvark
aardwolf
aardwolves
Aarhus
Aaron
ABA
Ababa
aback
abacterial
abacus
abacuses
```

Fig. 1. The first few entries in the dictionary used by the spelling checker ispell.

unencrypted version of the OED2 is available, but expensive). In both the LDOCE and the CELEX dictionaries, the data is available programmatically.

In Webster's Ninth, the user can access data only through spelling. In the OED, the user can access data through spelling, quotation year, original language, and many other attributes. In the CELEX dictionary, which is a relational database, access is completely flexible. And in LDOCE, the user can access the data through selected fields in the data entries.

Coverage. A fourth trend in modern electronic dictionaries is that, as linguistically based software systems become larger, the need grows for representing all and only those words relevant to a particular application in the electronic dictionary. All relevant words must be represented to allow for complete coverage. It is desirable also to represent only relevant words to improve storage and retrieval costs. At least one of the learning dictionaries discussed above, Collins COBUILD dictionary, chose its selection of entries not by the traditional approaches of introspection and of searching for unusual word uses, but rather by amassing a corpus of text and entering word occurrences in that corpus into its dictionary. This should result in a dictionary with a vocabulary representative of the domain in which the corpus was gathered rather than an idiosyncratically collected vocabulary. This approach yields an additional desirable characteristic: statistical studies on the corpus can indicate frequency of word use, which can be used in ordering both linguistic uses and meaning uses of the same word. This frequency information could be useful to software applications.

Sample Dictionaries

Word lists contain word spellings, sometimes accompanied by syllabification, pronunciation, or frequency information. An example is the dictionary used by the spelling checker ispell. Fig. 1 shows the first few entries in this dictionary, which contains word entries by spelling only.

Another well-known word list electronic dictionary is the Brown corpus, which was constructed from statistical studies on linguistic texts. It provides spelling and part-of-speech information. However, its great contribution is its frequency information, which records the frequency of different words with the same spelling. Frequency lists usually collapse word frequencies to occurrences of a spelling instead of accounting for homonyms.

Other electronic dictionaries contain more extensive data than do word lists. The desktop dictionary Webster's Ninth on CD-ROM, for instance, provides spelling, syllabification, pronunciation, meaning, part of speech, and some etymological information for each word.

The Oxford English Dictionary on CD-ROM desktop dictionary also provides extensive information. Its data includes spelling, etymology (parent language), part of speech, quotations to demonstrate context, year of quotation, and meaning.

The data in the machine-readable Longman's Dictionary for Contemporary English (LDOCE) and in the CELEX lexical database from the Centre for Lexical Information is also extensive and includes phonology, syllabification, part of speech, morphology, specification of the arguments that occur with the word in a phrase, such as subject, object and so on (subcategorization), and in the CELEX dictionary, frequency. Also, while the desktop type of electronic dictionary does not typically contain linguistic information that coincides with modern linguistic theories, these two electronic dictionaries contain work categorizations based on modern linguistic theory. Much of the CELEX dictionary's syntactic data is based on categories from the LDOCE and the OALD.

Fig. 2 shows examples from the CELEX electronic dictionary. In these examples, the lemma is the root word, the part of speech is the major word class to which the word belongs, the morphology is the formula for deriving the spelling of the flected word from the lemma's spelling, morphological information includes morphological characteristics of the word (singular, comparative, etc.), and flection type contains the same morphological information compressed to one column.

The HP Natural Language System

A natural language processing system is a software system that takes natural language input (typically spoken or typed input), assigns some interpretation to the input, and perhaps transforms that interpretation into some action. Examples of natural language processing systems are speech recognizers, machine translators, and natural language understanding systems.

HP's natural language understanding system, HP-NL, accepts typed English input and performs morphological, syntactic, and semantic analysis on the input. If the sentence is well-formed, HP-NL assigns a logic representation to the sentence. HP-NL can then translate this logic expression into another language, for instance a database query language such as SQL.

The linguistic coverage of HP-NL is limited by, among other factors, the size of its lexicon, or its word inventory. To increase the size of the lexicon and therefore the coverage of the software system, and to demonstrate that electronic dictionaries can be used to solve problems of computation, we integrated the CELEX lexical database into the HP-NL natural language understanding system.

entry	lemma	part of speech	syntactic information
1	a	ART	NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
2	aback	ADV	NNNNNNNNNNNNNNNNNNNYNNNNNNNNNNNNNNN
3	abacus	N	YNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
4	abandon	N	NYNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
5	abandon	V	NNNNNNNNNYNNNNNNNNNNNNNNNNNNNNNNNNN
6	abandoned	A	NNNNNNNNNNNNNNYNNNNNNNNNNNNNNNNNNNN
7	abandonment	N	NYNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
8	abase	V	NNNNNNNNNYNNNNNNNNNNNNNNNNNNNNNNNNN
9	abasement	N	NYNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
10	abash	V	NNNNNNNNNYNNNNNNNNNNNNNNNNNNNNNNNNN

entry	lemma	spelling	syllabification	morphology	frequency	flection type	morphological information
6	4	abandon	a-ban-don		143	S	YNNNNNNNNNNNN
7	5	abandon	a-ban-don		36	i	NNNNNYNNNNNNN
8	6	abandoned	a-ban-doned		321	b	NNYNNNNNNNNNN
9	5	abandoned	a-ban-doned	+ed	64	a1S	YNNNNNNNYYNNN
10	5	abandoning	a-ban-don-ing	+ing	76	pe	NNNNNNYYNNNNN
11	7	abandonment	a-ban-don-ment		88	S	YNNNNNNNNNNNN
12	5	abandons	a-ban-don-s	+s	15	e3S	YNNNNNNYNNNYN

Fig. 2. Examples of data from the CELEX electronic dictionary.

Natural Language Processing Technology

Current linguistic technology relies on detailed linguistic specification of words. Linguistic analysis is based on basic information about words, the building blocks of phrases. We will restrict our consideration to two central kinds of linguistic information: morphological and syntactic information.

Morphological information specifies which component parts of a word (morphemes) may occur together to constitute a word. Morphemes may show up on a restricted class of words. For instance, the prefix *un-* may appear on adjectives, and the suffix *-s* may occur on third-person verbs:

1a.	happy	(adjective)
b.	un+happy	(adjective)
c.	work	(verb)
d.*	un+work	

2a.	work	(base verb)
b.	work+s	(present third-person-singular verb)
c.	happy	(adjective)
d.*	happy+s	

Syntactic information specifies how words interact with other words to form phrases. Two important kinds of syntactic information are part of speech and subcategorization. Part of speech is the major word category to which the word belongs—for example, noun, verb, adjective, and so on. Part of speech is important in determining which words may occur together and where they may occur in a sentence. For instance, a verb does not typically occur as the first element in a declarative English sentence:

3a. She finished repairing the broken toy.
b.* Finished she repairing the broken toy.
c.* Finished repairing the broken toy.

Subcategorization indicates more specifically than part of speech which words or phrases may occur with the word in question. It specifies how many and which arguments may occur with a word. *Devour* must have a noun phrase following it in a sentence (*devour* subcategorizes for a postverbal noun phrase), whereas *eat* need not:

4a. The tiger devoured its kill.
b.* The tiger devoured.
c. The tiger ate its kill.
d. The tiger ate.

Subcategorization also allows us to determine which verbs may occur where in verbal clusters:

5a. They may have left the party already.
b.* They may left the party already.
c.* They may have could left the party already.

Words must be specified in sufficient detail that the natural language processing system can draw distinctions such as those indicated above.

HP-NL's Lexicon

The grammatical theory behind the HP-NL system is HPSG (head-driven phrase structure grammar).[3,5] In this theory, as in most other modern linguistic theories, full specification of linguistic information at the word level is essential.

Many words have a great deal of linguistic information in common. For instance, in example 4 above, each verb subcategorizes for the same kind of subject and object, but the object is obligatory in the case of *devour*, and optional in the case of *eat*. In example 2 above, we see that English present third-person-singular main verbs end with *-s*. HP-NL captures these and other linguistic similarities of words through a system of hierarchical word classification.[6]

HP-NL's lexicon consists of word entries and word classes arranged in a tree hierarchy (the word class hierarchy). Each nonleaf node in the word class hierarchy is a word class. A word class defines a collection of words that share some cluster of linguistically relevant properties, which are predictive or descriptive of the linguistic behavior of the words. The words may be similar morphologically and/or syntactically, and may have similar

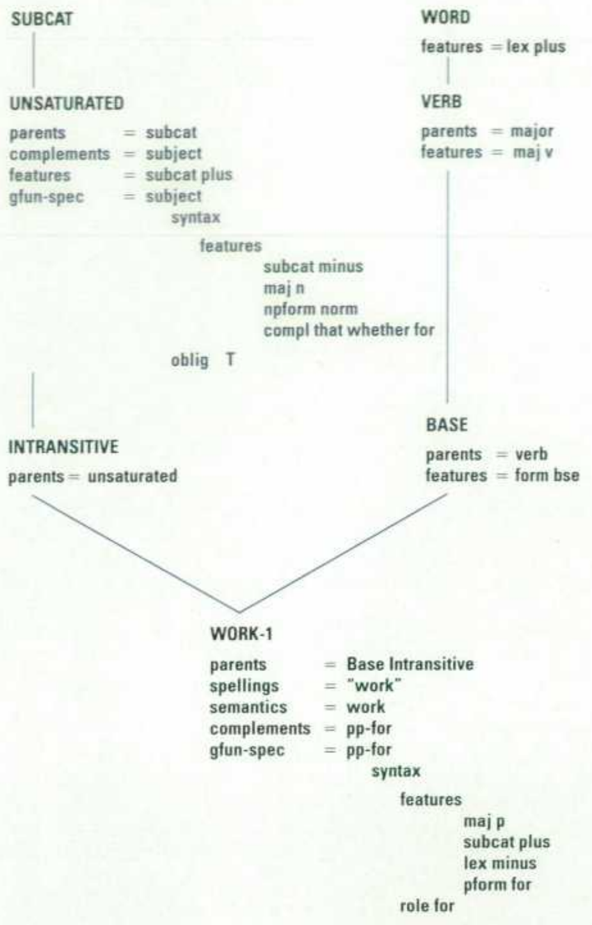


Fig. 3. An example of a word entry (subcategorization section for the word *work*), excerpted from a word class hierarchy.

subcategorization. A word class partitions the lexicon into two sets of words: those that belong to the word class and those that do not. The characteristics of a word class are defined by the characteristics that its members share. Word classes are more general closer to the root of the word class hierarchy, and are more specific closer to the leaves. Word entries are the leaves of the word class hierarchy. The word entry itself contains spelling, word class membership (parent), and idiosyncratic linguistic information (see Fig. 3).

The complete linguistic specification of a word is established through instantiation (Fig. 4). In this, the linguistic information of the word is unified with the information of all of its parent word classes. Any possible conflicting information is resolved in favor of the more specific information.

Lexicon Development

The development of this extensive word classification system, the word class hierarchy, makes the creation of HP-NL lexical entries fairly easy. Lexical development consists of determining the word class to which a word belongs and identifying the word's idiosyncratic behavior with respect to that word class and of recording the spelling and idiosyncratic linguistic behavior of the word.

Even with this powerful lexical tool, creating lexical entries is time-consuming. First, each word to be entered into the lexicon must be analyzed linguistically for word class membership and idiosyncratic behavior with respect to that word class. And second, it is difficult for the lexicon developer to know which words a user of a natural language processing system will want to use, and which should therefore be entered into the lexicon. Of course, lexical tools such as desktop dictionaries and frequency listings can help the lexicon developer, but nonautomatic lexicon development is still work-intensive.

Because of this, a hand-built lexicon must be small or labor-expensive. And because the linguistic coverage of a natural language processing system is limited by the size of its lexicon, the narrow coverage resulting from the small lexicon could result in failure of the natural language processing system caused solely by unrecognized vocabulary. Natural language processing systems used as computer interfaces are intended to allow the user maximum freedom in expression.

To address the problems of identifying the most common words of English and specifying their linguistic behavior, HP-NL's lexicon was augmented with dictionary data obtained from the CELEX electronic dictionary. The CELEX electronic dictionary was chosen for three reasons. First, the linguistic classification system is compatible with modern linguistic theory. Second, the data is fully accessible. And third, the CELEX electronic dictionary provides the frequency data needed to identify common words.

Lexical Extension Using the CELEX Dictionary

Several advantages were expected from using the lexical information in the CELEX electronic dictionary. First, the primary objective in this effort was to increase the linguistic coverage of HP-NL by increasing the size of the HP-NL lexicon with externally compiled dictionary data from the CELEX electronic dictionary. Until CELEX was integrated into HP-NL, HP-NL's basic lexicon contained approximately 700 root words (about 1500 words in all, including those derived from the root words by lexical rule). Because the only sentences that can be parsed are

```
WORK-1, instantiated
spellings     = "work"
semantics     = work
features      = subcat plus
                lex plus
                maj v
                form bse
complements   = subject pp-for
gfun-spec     = subject
                  syntax
                      features
                              subcat minus
                              maj n
                              npform norm
                              compl that whether for
                  oblig  T
                pp-for
                  syntax
                      features
                              maj p
                              subcat plus
                              lex minus
                              pform for
                      role for
```

Fig. 4. An example of an instantiated word entry.

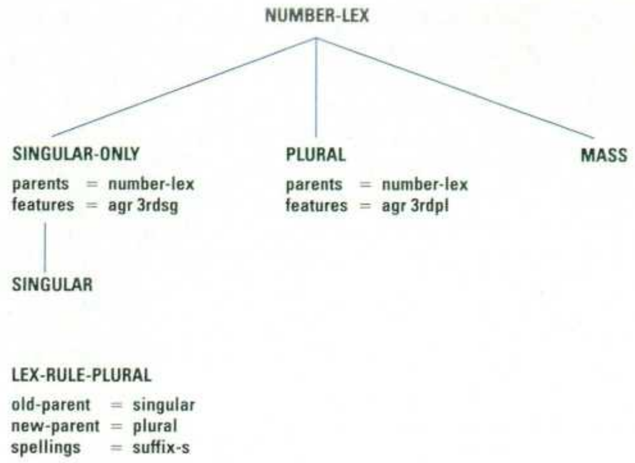


Fig. 5. An excerpt from the word class hierarchy for the Singular and Plural word classes, with lexical rule.

those whose component words have been specified explicitly in the dictionary, HP-NL's rather small dictionary also necessarily meant narrow coverage. The CELEX dictionary has both more words than HP-NL (the CELEX dictionary has 30,000 lemmas and 180,000 wordforms) and more different word uses for many of the words.

Second, because the developers of the CELEX dictionary drew on learning dictionaries of English (which attempt to cover core vocabulary) for their lexical data, the CELEX dictionary represents the most common words in English. Assuming that natural language processing users will prefer common rather than unusual words, using the CELEX data should eliminate the need on the part of HP-NL lexicon developers to guess at the words commonly used by natural language processing users.

Finally, we believed that buying a license to use the CELEX dictionary would be cheaper than creating a large lexicon ourselves.

All of these expectations were met. The CELEX dictionary was clearly the best choice among the candidate electronic dictionaries. The data is accessible, unambiguously categorized, and extensive. It recognizes many lexical classes of interest to linguists, and the fee for commercial clients is reasonable. None of the other candidate dictionaries had all of these qualities.

Procedure

In the work reported here, the orthography, language variation, phonology, inflectional morphology, and syntax data from the English CELEX database was used.

To integrate a large portion of the CELEX dictionary into the HP-NL dictionary, we transduced CELEX spelling, syntactic, and morphological information into a form compatible with the HP-NL system by mapping the CELEX dictionary's word classifications onto the (often more detailed) word classes of HP-NL's lexical hierarchy.

Several mappings between the CELEX dictionary's word classification scheme and HP-NL's word classes are straightforward. For instance, the CELEX dictionary's two classes called count nouns (C_N) and uncount nouns (Unc_N) correspond to HP-NL's three classes Singular, Plural, and Mass nouns. A count noun can be pluralized and takes a singular verb in its singular form and a plural verb in its plural form. Examples from the CELEX dictionary are *almond(s)*, *bookworm(s)*, and *chum(s)*.

6a. Her chum was waiting for her at the corner.
b. Her chums were playing tag when the cat got stuck in the tree.

The two word classes in HP-NL that together correspond to the C_N word class are the Singular word class and the Plural word class, which are related by the plural lexical rule (Fig. 5).

The CELEX uncount nouns are those nouns that occur only in the singular form with a singular verb. This includes mass and abstract nouns. Examples of Unc_N nouns from the CELEX dictionary are *bread*, *cardboard*, and *integrity*.

7a. How much cardboard is in that box?
b.* How many cardboards are in that box?

8a. The ruler has great integrity.
b.* The ruler has great integrities.

The corresponding word class in HP-NL is the Mass word class (Fig. 6).

Some nouns can be used as either count or uncount nouns and are classified as both C_N and Unc_N in the CELEX dictionary. Examples are *cake*, *hair*, and *cable*:

9a. How much cake would you like?
b. How many cakes are on the table?

These are classified as both Singular (and therefore also derived Plural) and Mass in HP-NL.

This portion of the mapping between the CELEX dictionary and HP-NL is simple:

C_N ↔ Singular (and by derivation, Plural)
Unc_N ↔ Mass

This shows an apparent one-to-one mapping. However, some of the remainder of the CELEX dictionary nouns also map onto the Singular word class.

Sing_N	for Nouns: Singular Use
Plu_N	for Nouns: Plural Use
GrC_N	for Nouns: Group Countable
GrUnc_N	for Nouns: Group Uncountable

The Sing_N and GrUnc_N classes both map onto the Singular-only HP-NL word class, and therefore these words have no

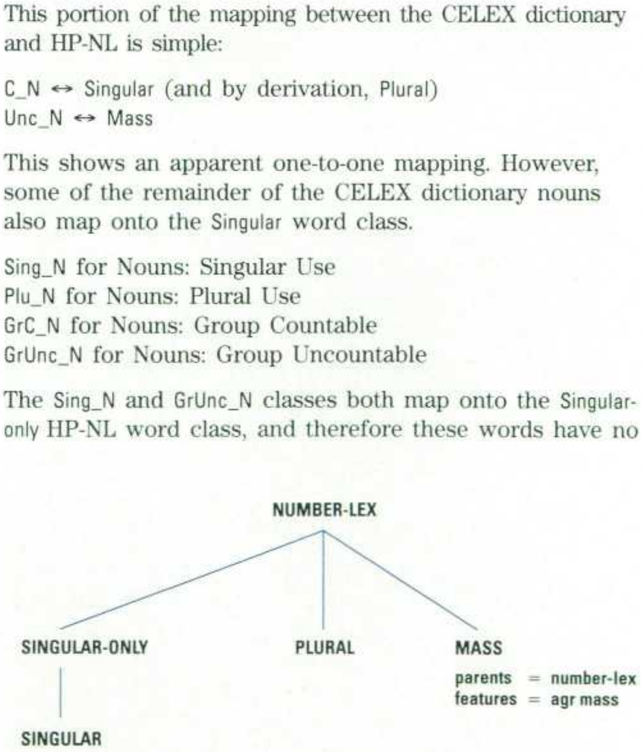


Fig. 6. An excerpt from the word class hierarchy for the word class Mass.

related Plural form. The GrC_N class maps onto the Singular word class, so these words have a related Plural form.

If all of the many-to-one mappings had been multiple CELEX word classes collapsing down to one HP-NL class, the transduction would have been perhaps difficult to untangle, but possible. Instead, as we will see in the next section, in some cases HP-NL recognizes more word classes than the CELEX dictionary. This means that information that HP-NL needs for accurate parsing is not provided by the CELEX dictionary.

Difficulties

While using the CELEX dictionary did pay off in the expected ways, there were some difficulties that the user of electronic dictionaries should be aware of.

Finding the correspondences between the CELEX dictionary's word classes and HP-NL's word classes was surprisingly complicated. Part of the problem was sketchy documentation: some of the word classes were underdescribed, and in one case, the documentation inaccurately described the data, switching two columns (this error has since been corrected). Also, some of the linguistic distinctions the CELEX dictionary recognizes are orthogonal to the distinctions HP-NL recognizes. Furthermore, some of the correspondences between the CELEX dictionary and HP-NL word classes involved one-to-many mappings in which the CELEX dictionary recognizes fewer word class distinctions than HP-NL requires.

Unclear CELEX Classification. The documentation provides a short one-to-three-word description for each of the syntactic word categories. In some cases, the description clearly describes the syntactic behavior. For instance, the example above demonstrating the mapping between the the CELEX dictionary C_N and Unc_N classes and the HP-NL Singular, Plural, and Mass word classes shows a case in which the somewhat slim documentation was adequately informative.

In the case of verb subcategorization, however, the documentation is not informative enough. The CELEX dictionary recognizes eight verb subcategorization classes: transitive, transitive plus complementation, intransitive, ditransitive, linking, prepositional, phrasal prepositional, and expressional verbs. Exactly what syntactic behavior is meant by each of these classes is unclear, however, although sample words are given in addition to a description for each word class. The following sample words occur in the CELEX documentation:

Trans_V: Transitive
  crash: *he crashed the car*
  admit: *he admitted that he was wrong*
not *cycle*: * *he cycled the bike*
TransComp_V: Transitive plus Complementation
  found: *the jury found him guilty*
  make: *they had made him chairman*
Intrans_V: Intransitive
  alight: *he got the bus and alighted at the City Hall*
  leave: *she left a will*
    she left at ten o'clock
  not *modify*: * *he modified*
Ditrans_V: Ditransitive
  envy: *he envied his colleagues*
  tell: *she told him she would keep in touch*
Link_V: Linking Verb
  be: *I am a doctor*
  look: *she looks worried*
Phrasal: Phrasal Verb
  Prepositional
    minister to
    consist of
  Phrasal prepositional
    walk away with
    cry out against
  Expression
    toe the line
    bell the cat

In English, there is a group of verbs that occur with two arguments. Sometimes the second postverbal argument must be a noun phrase, sometimes it must be a *to* prepositional phrase, and sometimes it may be either. Consider the following uses of *give*, *explain*, and *begrudge*:

10a. The girl gave a book to her younger sister.
  b. The girl gave her younger sister a book.

  c. The girl explained the story to her sister.
  d.* The girl explained her sister the story.

  e.* The girl begrudged her new ball to her sister.
  f. The girl begrudged her sister her new ball.

The linguistic behavior of these words is clear. However, the CELEX documentation does not clarify which class should contain the use of *give* in 10a and explain in 10c, which can accept either a noun phrase (NP) and a pp-to phrase as the second argument. Furthermore, it is unclear from the documentation whether the CELEX dictionary recognizes the two uses of *give* as being related to each other.

Inspecting the CELEX data also yields no clear indication whether the ditransitive class includes verbs with only two NP arguments like *begrudge*, with only one NP and one pp-to argument like *explain*, or with both like *give*. Perhaps all three verb types are ditransitive, perhaps only those that alternate, and perhaps only those that accept only two noun phrase complements. Inspecting the classification of these three words themselves yields little more insight. Many words have multiple syntactic behaviors, so that it is difficult to tell exactly which syntactic behaviors which CELEX word classification is intended to cover.

Following are some other examples that demonstrate the difficulty of ascertaining the intended meaning of the CELEX verb classes:

11a. I waited all day for you!
  b. The patriots believed that their government was right.

It is unclear whether *wait for* is a transitive verb (the pp-for argument being considered an adjunct), a ditransitive verb (the pp-for argument being considered a second complement), or a transitive plus complementation verb (the pp-for argument being considered a miscellaneous complement). Similarly, in the case of *believe*, it is unclear both from documentation and from inspecting the data whether the sentential argument of *believe* causes this verb to be transitive, transitive plus complementation, or intransitive.

Orthogonal Classification. The CELEX dictionary recognizes some categories that are orthogonal to the kinds of categories that HP-NL recognizes and requires. For instance, the CELEX dictionary recognizes linking verbs, which "link a subject *I* with a complement that describes that subject *a doctor* in a sentence like *I am a doctor.* These subject complements can take the form of a noun phrase (*She is an intelligent woman*), an adjective phrase (*She looks worried*), a prepositional phrase (*She lives in Cork*), an adverb phrase (*How did she end up there?*) or a clause (*Her main intention is to move somewhere else*)." (CELEX Users' Guide)

The distinction between linking and nonlinking verbs is a semantic one. HP-NL's word class hierarchy draws morphological and syntactic distinctions, but not semantic ones. So this information must be identified as not being useful (currently) and discarded.

One-to-Many Mapping. Some of the distinctions the CELEX dictionary draws are useful but not extensive enough for HP-NL's purposes. For instance, linguistic theories distinguish between two types of phrasal verbs: raising and equi verbs.

Raising:
12a. The student seems to be healthy.
b. There seems to be a healthy student.

Equi:
13a. The student tried to climb the tree.
b.*There tried to climb the tree the student.

The CELEX dictionary does not draw this distinction. While the verbs *seem* and *try* are indeed present in the database, not all of their syntactic behavior is documented. To use the CELEX dictionary, some of the data would have to be augmented from some other source.

One large group of words is underspecified in the CELEX dictionary with respect to the HP-NL natural language processing system: the members of the ***closed*** word classes, those classes of grammatical words to which new words are seldom added. Examples are prepositions such as *of* and *to*, determiners such as *the* and *every*, and auxiliary verbs such as *be* and *could*. These grammatical words carry a great deal of information about the linguistic characteristics of the phrase in which they appear, and must therefore be specified in detail for a natural language processing system.

Outcome

Despite the difficulties noted here, incorporating the CELEX dictionary into HP-NL turned out to be not only profitable for the system but also instructive. The vocabulary of the HP-NL system was increased from about 700 basic words to about 50,000 basic words.

The addition of the words greatly increased the number of sentences that could be parsed. This increase, however, resulted in an overall slowing of parsing, because of lexical ambiguity. This both slowed word retrieval and increased the number of possible partial parses.

Only words in the ***open*** classes (noun, verb, adjective) could be added. The HP-NL system requires a lexical specification that is too theory-specific for the very important grammatical words, as well as for some members of the open classes. However, many members of the open classes could be correctly represented in the HP-NL format.

The HP-NL project was terminated before user studies could be conducted that would have determined whether the CELEX dictionary provides the words a user would choose while using a natural language processing system.

Computational Applications of Electronic Dictionaries

This case study, done using a large electronic dictionary, suggests that electronic lexographical information can be incorporated successfully into nondictionary applications. First, we found that the CELEX data is in such a form that it can be transformed for and accessed successfully by a software application. Second, the data in the CELEX dictionary is useful in the domain of natural language processing. The areas of success and difficulty in incorporating the CELEX dictionary into the HP-NL system should indicate which kinds of software applications could successfully integrate an electronic dictionary.

The greatest gain from the CELEX dictionary was in increasing HP-NL's vocabulary dramatically. Although the vocabulary increase also resulted in slower parsing, the larger vocabulary was still seen as an improvement, because the larger vocabulary greatly extended HP-NL's natural language coverage. For an application that does not seek to have wide vocabulary coverage, a large dictionary would clearly not provide the same large advantage.

Another improvement to HP-NL is in the particular vocabulary represented. The CELEX dictionary provides common English words, which are the words HP-NL needed. An application requiring an unusual vocabulary (for instance, a vocabulary of technical terms) would not benefit from the CELEX dictionary as much as did HP-NL.

The largest problem in using the CELEX dictionary was inadequate information for some word classes. Some of the documentation was not completely clear, and some words were not represented in the detail required for successful parsing by HP-NL. This rendered some of the CELEX dictionary's information useless for HP-NL. This did not present a great difficulty; many of the problematic words had already been created for HP-NL. Of course, not all applications of dictionary technology will be in such an advanced linguistic state as HP-NL. An application of dictionary technology has the most likelihood of being successful, at least in the near term, if it does not require very fine categorization of words, particularly closed-class words.

One topic that was not addressed in the current study is the role of word meanings in a software application. The CELEX dictionary contains no definition information. Therefore, its words have no meaning with respect to a particular domain such as querying a particular database.

Mapping the meanings of the words from the CELEX dictionary to a domain must still be done painstakingly by hand. A thesaurus could potentially provide large groups of synonyms for words that are defined by hand with respect to an application. At this point, however, the most successful application of electronic dictionary technology would avoid the problem of meaning entirely and use words themselves as tokens.

In summary, the kind of software application most likely to benefit from electronic dictionaries would require a large vocabulary of common words.

Choosing Applications

Now that we have some idea of the characteristics of the software applications that might benefit from an electronic dictionary and what kinds of problems might be encountered in incorporating the dictionary into the application, we can consider what particular applications could use electronic dictionaries.

To review, software systems of the following types could benefit from the data in an electronic dictionary:

- Any software system that uses natural language in any way
- A software system that requires a large vocabulary of common words
- A software system that does not require detailed linguistic specification of grammatical words
- A software system that does not make use of word definitions
- A software system that does not require complete linguistic analysis of words unless it supplies its own categorization scheme (such as HP-NL).

Some types of software applications that match these characteristics are natural language processing, speech generation and recognition, document input, document management, and information retrieval.

The electronic dictionary in turn should possess the following characteristics:

- Data accessible to software application (not encrypted)
- Good data organization (so that access is easy and flexible)
- Appropriate vocabulary (for instance, good coverage of the core vocabulary of English)
- Appropriate additional information (for instance, modern linguistic classification for natural language processing systems).

Of the dictionaries we have surveyed, few satisfy the first requirement. The CELEX dictionary, LDOCE, and several word lists have accessible data. These electronic dictionaries vary in the degree to which they exhibit the other characteristics.

Natural Language Processing. Natural language processing systems are the software applications whose need for electronic dictionaries is most extensive and most obvious. The information necessary is spelling, morphology, part of speech, and subcategorization, at least. A more extensive discussion of the role of electronic dictionaries in natural language processing systems was presented earlier in this paper.

Speech Technology. In both speech generation and speech recognition, a vocabulary list and associated pronunciations are essential. Depending on the sophistication of the speech system, other linguistic information may also be useful.

If the speech generation system is to generate words alone, a word list with pronunciations is sufficient, but if it must generate full phrases or sentences spontaneously rather than from a script, a natural language generation system is necessary. This generation system may be based on linguistic theory or it may be based instead on template forms, but in either case, an electronic dictionary-could provide the word classification information necessary.

Speech recognition systems that recognize one-word or canned commands also need no more than a word list with pronunciations. However, if a speech recognition system must recognize spontaneously created phrases, a more sophisticated approach to recognition is necessary. After the word possibilities have been identified, there are several ways in which the speech recognizer can identify potential sentences:

- By ruling out ill-formed sentences on the basis of impossible word-type combinations
- By rating possible sentences on the basis of collocation information derived from a statistical survey of texts
- By parsing with a natural language understanding system.

Of these, the first and last possibilities would require word class information in addition to pronunciation information, which can be gained from electronic dictionaries currently available. The second possibility would require data from a statistical study, preferably performed on texts from the relevant domain.

Document Input. Examples of computer applications that facilitate document input are optical character recognition (OCR), "smart" keyboards, and dictation aids. Document input is error-prone. One of the many ways to reduce errors is to allow the computer access to linguistic information.

Such an application would need a word listing, a theory of the errors likely to be made by the system, and a theory of the relative frequency of appearance of well-formed subparts of words. A more sophisticated system might recognize multiple word blocks, requiring the linguistic module to provide either word classification information (for parsing-like ability) or word collocation information (for statistical information on word co-occurrence).

The HP-UX* operating system provides a minimal "smart" keyboard facility in the `csh` environment, with an escape feature for completing known commands. This feature could be expanded for full English, and could include not only word completion, but also partial word completion. That is, the application could have some knowledge of the frequency of substrings in spellings (or its equivalent in speech), and with this knowledge could reduce the number of keystrokes necessary for input.

When speech recognition technology advances sufficiently, the opportunity for dictation aids will arise. These tools could perform a similar function to smart keyboards, but in the realm of spoken rather than typed language.

Optical character recognition is one of the most promising areas in which electronic dictionaries could be used. At least one language-based assistance product for OCR is available commercially: OmniSpell, a spelling corrector for use with the OCR product OmniPage. It suggests likely alternate spellings for strings not recognized as words.

Document Management. Linguistic information can also aid in checking and improving the quality of a document stored on a computer. Spelling checkers, based on common typographical errors and variation of the misspelling from well-formed words, as well as on phonological characteristics of the misspelled word, are available already.

Grammar and style checkers, however, are not available in as great abundance or variety. There are grammar checkers available such as the Grammatik spelling checker, but they focus primarily on form statistics (average word length in a sentence, average syllable length of words) and on frozen style characteristics (use of idiomatic expressions and cliches, use of passive). They are notably not very good at identifying errors in grammar, such as lack of subject-verb concord with complex subjects,

14. * The teacher but not the students are happy with the football team.

choice of correct pronoun case in complex prepositional objects,

15. * Between John and I, he works more.

and similar subtle points in grammar.†

Natural language parsing technology could improve the performance of grammar checkers, and an electronic dictionary would be an important part of such a natural language processing system. Otherwise, an electronic dictionary indicating part of speech and other relevant grammatical information such as verb conjugation class, noun phrase number, and pronoun case could be useful in heuristic inspection of sentences.

Information Retrieval. Information retrieval capability could be expanded by incorporating a theory of related words into information retrieval systems. While this expansion may not be necessary for standard databases in which words have a formal meaning and are not really natural language items, it could be very useful in full-text databases. There are two kinds of information that could expand retrieval possibilities.

† One well-known grammar checker incorrectly identified both of these sentences as being grammatical.

First, a user might search on a keyword but be interested in retrieving all occurrences of that word in related morphological forms:

16a. factory / factories
b. goose / geese
c. happy / happiness

A morphological analyzer module that can recognize morphologically-related words, either through exhaustive listing or through some theory of morphological variants, might expand retrieval possibilities.

Second, a user might be interested in retrieving all information on a particular topic, but that topic might be identified by several different synonyms. For instance, the user might want to retrieve all mentions of *animal* in a text. A thesaurus would permit the user to retrieve mentions of *creature* and *beast*, and perhaps also subtopics such as *mammal*, *amphibian*, and *reptile*.

Conclusion

Electronic dictionaries have recently reached a state of development which makes them appropriate for use on the one hand as machine-readable end-user products, and on the other hand as components of larger language-based software systems. There are several domains of software applications that either are already benefitting from electronic dictionaries or could benefit from electronic dictionaries that are available now. One project at Hewlett-Packard Laboratories has successfully integrated one electronic dictionary, the CELEX lexical database, into its natural language processing system. Other software applications that could use the extensive information available in electronic dictionaries are speech generation and recognition, document input such as optical character recognition and "smart" keyboards, document management such as spelling and grammar checking, and information retrieval.

Acknowledgments

I would like to thank the following people for related discussions: Brett Kessler, Dan Flickinger, Derek Proudian, Dale Morris, and David Keppel. I also thank Brett Kessler for his collaboration.

References

1. J. Nerbonne and D. Proudian, *The HP-NL System*, STL Report STL-88-11, Hewlett-Packard Company, 1988.
2. J. Bresnan, ed., *The Mental Representation of Grammatical Relations*, The MIT Press, 1982.
3. C. Pollard and I. Sag, *Information-Based Syntax and Semantics*, CSLI Lecture Notes, no. 13, Center for the Study of Language and Information, Stanford University, 1987.
4. *Collins COBUILD English Language Dictionary*, William Collins Sons & Co. Ltd., 1987.
5. C. Pollard, *Generalized Phrase-Structure Grammars, Head Grammars, and Natural Language*, PhD Thesis, Stanford University, 1984.
6. D. Flickinger, *Lexical Rules in the Hierarchical Lexicon*, PhD Thesis, Stanford University, 1987.

Bibliography

1. K.L. Acerson, *WordPerfect 5.1 The Complete Reference*, Osborne McGraw-Hill, 1990.
2. B. Boguraev and T. Briscoe, eds., *Computational Lexicography for Natural Language Processing*, Longman, 1989.
3. G. Burnage, *CELEX—A Guide For Users*, Drukkerij SSN, 1990.
4. J. Carroll, B. Boguraev, C. Grover, and T. Briscoe, *A Development Environment for Large Natural Language Grammars*, Technical Report No. 167, University of Cambridge Computer Laboratory, University of Cambridge, 1988.
5. *CELEX News*, nos. 1-5, December 1986 to August 1990, Centre for Lexical Information, University of Nijmegen, The Netherlands.
6. B. Dorr, "Conceptual Basis of the Lexicon in Machine Translation," *Proceedings of the First International Lexical Acquisition Workshop of the International Joint Conference on Artificial Intelligence (IJCAI)*, Detroit, Michigan, 1989.
7. Duden *Bildwörterbuch*.
8. D. Flickinger and J. Nerbonne, *Inheritance and Complementation: A Case Study of EASY Adjectives and Related Nouns*, Association for Computational Linguistics, 1992, to be published.
9. D. Flickinger, C. Pollard, and T. Wasow, "Structure-Sharing in Lexical Representation," *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, Chicago, Illinois, 1985.
10. W.N. Francis and H. Kucera, *Frequency Analysis of English Usage: Lexicon and Grammar*, Houghton-Mifflin Company, 1982.
11. G. Gazdar, E. Klein, G. Pullum, and I. Sag, *Generalized Phrase Structure Grammar*, Harvard University Press, 1985.
12. C.F. Goldfarb, *The SGML Handbook*, Clarendon Press, Oxford, 1990.
13. C. Grover, T. Briscoe, J. Carroll, and B. Boguraev, *The Alvey Natural Language Tools Grammar*, Technical Report no. 162, University of Cambridge Computer Laboratory, University of Cambridge, 1989.
14. M. Iida, J. Nerbonne, D. Proudian, and D. Roberts, "Accommodating Complex Applications," *Proceedings of the First International Lexical Acquisition Workshop of the International Joint Conference on Artificial Intelligence (IJCAI)*, Detroit, Michigan, 1989.
15. K. Koskenniemi, "Two-Level Model for Morphological Analysis," *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, Germany, 1983.
16. K. Koskenniemi, *Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production*, Publication no. 11, University of Helsinki, 1983.
17. *Longman Dictionary of Contemporary English*, Longman Group UK Ltd., 1989.
18. S. Miike, K. Hasebe, H. Somers, and S. Amano, "Experiences with an On-Line Translating Dialogue System," *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics*, Buffalo, New York, 1988.
19. *Oxford Advanced Learner's Dictionary of Current English*, Oxford University Press, 1989.
20. *Oxford English Dictionary on Compact Disc*, User's Guide, Oxford University Press, 1987.
21. D. Roberts, *Linking Thematic Roles and Syntactic Arguments in HPSG*, Unpublished Master's Thesis, Cornell University, 1991.
22. S. Shieber, *An Introduction to Unification-Based Approaches to Grammar*, Center for the Study of Language and Information, Stanford University, 1985.
23. B.M. Slator, "Using Context for Sense Preference," *Proceedings of the First International Lexical Acquisition Workshop of the International Joint Conference on Artificial Intelligence (IJCAI)*, Detroit, Michigan, 1989.
24. *Stedman's Medical Dictionary*, Williams & Wilkins Company, 1938.
25. *Webster's Ninth New Collegiate Dictionary*, Merriam-Webster, 1989.

Authors

June 1992

6 HP-UX Kernel Support

Karen Kerschen

Documentation usability is one of the professional interests of learning products engineer Karen Kerschen. Karen's first project when she joined HP's Technical Workstation group in 1987 was to edit the *HP-UX Reference Manual*. Since that project she has coauthored an HP-UX manual on system security and is now responsible for an HP-UX manual for system administrators that decribes how the HP-UX operating system works. Before joining HP, she worked in the University of California's EECS department as the assistant editor for the IEEE publication *IEEE Electrotechnology Review*. To stay abreast of developments in her field, she is a member of the Society for Technical Communication. She has a BFA (1967) from the Cooper Union for the Advancement of Science and Art in New York City. Karen grew up in New York. Her interests and hobbies include photography, Latin American studies, creative writing, and Spanish translation.

Jeffrey R. Glasson

Software engineer Jeffrey Glasson was responsible for developing software to take advantage of new PA-RISC hardware features and special routines to handle floating-point traps. He has also worked on HP-UX performance tuning and developing low-level code for the HP 9000 Model 835. Jeff came to HP in 1984 after earning a BS in computer engineering from the University of San Diego that same year. He is now a senior software engineer at Apple Computer, Inc. He is married and enjoys skiing, wine tasting, and playing strategy games.

11 Kernel Functionality

Frank P. Lemmon

Product assurance engineer Frank Lemmon was a software testing specialist in the HP-UX kernel laboratory. He was responsible for providing automated regression testing for the minimum core functionality kernel. He joined HP in 1979 at the former Computer Systems Division. Some of the projects he worked on include a message I/O project, implementation of a stitch-wire breadboard system, coordination of the partner test plan for the HP-UX install and update programs, and development of a quick test facility for the HP-UX system integration process. Frank left HP in 1991 and now works as a product assurance engineer at Auspex Systems Inc. in Santa Clara, California. He has a BS degree in engineering (1973) from the University of California at Los Angeles and an MS in computer science (1976) from Santa Clara University. Before joining HP he worked as a hardware development engineer at Amdahl Corporation, and as a design engineer for Itek Corporation at the Applied Technology Division. Frank is married. He was the founder of the HP windsurfing club and is a hike leader with the Sierra Club.

Donald E. Bollinger

Don Bollinger was the project manager responsible for integration of system software for the HP 9000 Series 700 minimum core functionality release. Don joined HP in 1979 at the former Data Systems Division (DSD) after receiving a BS degree in electrical engineering and computer science that same year from the Massachusetts Institute of Technology. He also has an MBA from Santa Clara University. At DSD, Don worked as a development engineer on the HP 1000 RTE operating system. Before working on the Series 700 project, he was a project manager for development of HP-UX commands and integration of HP-UX system software. Born in Pasadena, California, Don is married and has three children. Being a concerned parent, he is very involved in the elementary school in his area.

Dawn L. Yamine

Productivity engineer Dawn Yamine worked as an inspections consultant for the software projects involved in developing the minimum core functionality release of the HP-UX kernel. Dawn joined HP's Manufacturing Productivity Division in 1984 after receiving a master's degree in computer science that same year from Bradley University in Peoria, Illinois. She also has a BS in accounting (1980) from Millikin University in Decatur, Illinois. She has worked as a quality engineer and as a technical trainer. She had the opportunity to teach structured design methods to the HP team that developed the system to monitor Biosphere II. Born in Decatur, Illinois, Dawn is married and enjoys skiing and boardsailing.

15 Optimizations in PA-RISC 1.1

Robert C. Hansen

A graduate of San Jose State University with a BS degree (1986) in computer engineering, Bob Hansen is a language engineer/scientist at HP's Systems Technology Division. Since joining HP in 1985, he has worked on a path flow analyzer (a tool for analyzing test coverage) and a native SPL/XL compiler used to port Turbo Image from MPE V to MPE XL. Before joining HP, he worked as a coop student at IBM doing real-time data acquisition. Before working on the PA-RISC optimizer project he was doing research on methods to do profile-based compiling. He is a coauthor of a paper that describes using profile information to guide code positioning. He is listed as an inventor on a pending patent that describes a technique for profile guided code positioning. Bob was born in Japan at Johnson Air Force Base. His hobbies and interests include remodeling houses and outdoor activities such as fishing, hiking, camping, and ultimate frisbee.

24 Optimizing Preprocessor

Daniel J. Magenheimer

A project manager in HP's Colorado Language Laboratory, Dan Magenheimer comanaged the Series 700 FORTRAN optimizing preprocessor project. He was a member of the team that defined the PA-RISC architecture when he joined HP Laboratories in 1982. He has a BA degree in computer science (1981) from the University of California at Santa Barbara and an MSEE (1985) from Stanford University. He has authored several technical articles and is a member of the IEEE Computer Society. His professional interests include computer architecture, performance analysis, compilers, and development environments. Born in Milwaukee, Wisconsin, Dan is married and has two children. His family keeps him busy but he does find time to play some basketball and softball.

Robert A. Gottlieb

Software engineer/scientist Bob Gottlieb joined HP's Apollo Systems Division in 1989. He has a BA in electrical engineering and mathematical sciences (1975) and a professional master of electrical engineering (MEE 1976) from Rice University. He was one of the engineers who worked on the optimizing preprocessor for the Series 700 FORTRAN compiler. He has also worked on HP Concurrent FORTRAN for the Apollo DN10000 system and the design and implementation of the induction and analysis phase for the FORTRAN compiler, also for DN10000. Before joining Apollo he worked at Aliant Computer Systems on parallelizing compiler development and at Digital Equipment Corp. on a Pascal compiler and CAD development. He is a member of the ACM and his professional interests are in compiler optimization, particularly in areas such as parallelism and interprocedural optimizations. Born in Aberdeen, Maryland, Bob is married likes traveling, skiing, and scuba diving.

Alan C. Meyer

Software development engineer Alan Meyer joined HP's Colorado Language Laboratory in 1989. Alan has a BS degree in computer science (1981) from the University of Washington and a PhD in computer science (1989) from Washington State University. He worked on the FORTRAN compiler project for the 8.05 release of the HP-UX operating system. He is a member of the ACM and has authored or coauthored two articles on geometric modelling. Born in Pullman, Washington, Alan is married and has one daughter. For relaxation he likes to hike, ski, and garden.

Sue A. Meloy

A software development engineer at HP's Systems Technology Division, Sue Meloy worked on the vector library for the Series 700 FORTRAN compiler. Sue was a coop student at HP in 1977 and became a permanent employee in 1979. Projects she worked on before the vector library include a C++ compiler code generator for the HP 9000 Series 300, an architecture neutral distribution format (ANDF) prototype, the C compiler for the Series 800, and BASIC compilers for the HP 300 and HP 1000 computer systems. Sue has a BS degree in computer science (1978) from California State University at Chico. An expert on C, she served as HP's representative on the ANSI C standards committee. Her professional interests include C, code generation, and debugging optimized code. She has also published several articles on debugging optimized code and C. Born in Redwood City, California, Sue is married. Her favorite pastimes include reading, cooking, and learning to play the piano.

33 Register Reassociation

Vatsa Santhanam

A computer language engineer/scientist at HP's California language laboratory, Vatsa Santhanam works on the design and implementation of compiler optimization techniques. He joined HP in 1984 at HP's Santa Clara Division. While there he worked as a test system software design engineer on a VLSI tester project. He has since worked on different optimization projects including an investigation of interprocedural optimizations. He also worked on a project that produced HP's response to an Open Software Foundation's request for technology for an architecture neutral software distribution format (ANDF). He received a Bachelors of Technology degree in electrical engineering (1982) from the Indian Institute of Technology in Madras, India and an MS in computer science (1984) from the University of Wisconsin at Madison. He also worked as a teaching assistant at the University of Wisconsin. He has coauthored three papers on compiler technology and is named as a coinventor on patent applications for an interprocedural register allocation technique and an architecture neutral distribution format. Vatsa was born in Madras, India and grew up in Japan and Hong Kong. He is married, and when he is not pursuing his professional interests in compilers and computer architecture, he likes to play chess, dabble in Hindu astrology, and listen to Indian classical music.

39 Software Pipelining

Sridhar Ramakrishnan

A software engineer at HP's Systems Technology Division, Sridhar Ramakrishnan joined HP in 1988. He worked on the instruction scheduling and pipelining components of the optimizer project. Before the optimizer project, he worked on the development of a programming environment for Ada based on SoftBench. He has a Bachelor of Technology degree in computer science (1985) from the Indian Institute of Technology in Bombay, India and an MS in computer science (1987) from the University of New Hampshire. Before joining HP, he worked on an Ada debugger and other tools at Tartan Laboratories in Pittsburgh, Pennsylvania. He has published a paper that describes a compiler for Prolog. Sridhar's professional interests include code generation and optimization and programming development environments.

46 Shared Library

Cary A. Coutant

Senior software engineer Cary Coutant was the project manager for the linker project for the HP 9000 Series 700 and 800 systems. Cary joined HP's Information Networks Division in 1979. Other projects he has worked on include HP Word, an HP 3000-based spooling system called Print Central, and HP-UX commands. Cary has a BS in physics (1977) from Furman University and an MS in computer science (1979) from the University of Arizona. He has coauthored one paper on an advanced tty driver for the UNIX* operating system and another paper on performance measurement. He is a member of the ACM and his main professional interest is computer languages. Born in Chicago, Illinois, Cary is married and has one daughter. Golf and tennis are among his recreational activities.

Michelle A. Ruscetta

A software development engineer at HP's Systems Technology Division, Michelle Ruscetta joined HP in 1983 after graduating from the University of Utah with a BS degree in computer science that same year. She was one of the engineers who worked on the linker and dynamic loader used in the shared library project. Projects she has worked on in the past include developing enhancements to HP's PA-RISC C compiler and developing methods of debugging optimized code. The optimized code project provided the basis for a paper that describes an approach to source-level debugging of optimized code. Michelle was born in Schenectady, New York. Her recreational interests include tennis and softball.

54 Electronic Dictionary

Diana C. Roberts

Diana Roberts received her BS degree in psychology from the Georgia Institute of Technology in 1982 and her MA degree in linguistics from Cornell University in 1991. She joined HP Laboratories in 1985 as a member of the technical staff and did development work in the lexicon of HP-NL, the HP natural language understanding system. She is a member of the Association for Computational Linguistics and has published on linguistic subjects. Besides her native English, she is fluent in German and has an active interest in the German language and German linguistics. She is conversant in French and has some facility in Italian and Spanish. From the fall of 1982 to the spring of 1984 she was an exchange fellow at the University of Hanover, Germany. During that time she served as an assistant teacher of linguistics at the university and taught English as a foreign language at a local language school. In the summer of 1991 she again taught English as a foreign language in Germany. Another of her interests is folk literature. Diana was born in Idaho Falls, Idaho and grew up in Atlanta, Georgia. Her leisure activities include camping, hiking, dancing, sewing, and reading.

68 Spatial Frequency Methods

Dale D. Russell

Now a member of the technical staff at HP's Boise Printer Division, chemist Dale Russell joined the HP Inkjet Components Operation in 1987. She has worked on ink formulations, waterfastness improvement for inkjet inks, color science, and the development of machine vision print quality evaluation methods, and is now doing R&D on electrographic toners. Two patents on waterfastness improvement, five pending patents on dye chemistry and analytical methods, and ten professional papers have resulted from her work. Her undergraduate degree is a BA in English from the University of California at Davis (1967), and she holds MS and PhD degrees in chemistry (1979 and 1985) from the University of Arizona. She served as a research chemist with the Environmental Trace Substance Research Program from 1979 to 1980 and as an assistant professor of chemistry at Northern Arizona University from 1984 to 1987, and is currently an adjunct professor of chemistry at Boise State University. She is a member of the Society for Imaging Science and Technology, the American Chemical Society, and the International Society for Optical Engineering. Born in San Diego, California, she is married and has two children. Her interests include mountaineering, snow, rock, and ice climbing, marathons and triathlons, and skiing. She's a certified scuba diver and paraglider pilot and a graduate of the National Outdoor Leadership School. She is active in her church, serves as a women's shelter volunteer, and teaches rock climbing, rappeling, and wilderness survival for the Boy Scouts of America.

76 Parallel Image Generation

Susan S. Spach

Susan Spach has been a member of the technical staff of Hewlett-Packard Laboratories in Palo Alto, California since 1985. She has done simulations of graphics algorithms to be designed in hardware and has worked on 3D graphics architectures and algorithms, concentrating on accelerating realistic rendering algorithms. A native of Durham, North Carolina, she attended the University of North Carolina at Chapel Hill, receiving her BA degree in mathematics in 1979 and her MS degree in computer science in 1981. Before coming to HP Labs, she was a software engineer at the same university, working on database applications and 3D graphics research. She is a member of the ACM and the IEEE Computer Society, and has coauthored several papers on 3D graphics algorithms, geometric modeling, rendering, and graphics architecture. She is married and has a daughter. An avid runner, she was an organizer of the HP National Running Club and serves as its treasurer.

Ronald W. Pulleyblank

A member of the technical staff of Hewlett-Packard Laboratories in Palo Alto, California, Ron Pulleyblank has been doing research on 3D computer graphics architectures and algorithms, concentrating on accelerating realistic rendering algorithms. A native of Detroit, he received BS degrees in electrical engineering and physics from the University of Michigan in 1964 and 1965, and a PhD degree in electrical engineering from the University of Pennsylvania in 1969. He taught electrical engineering at the University of Lagos, Nigeria and the University of the Pacific and worked on digital communications at Bell Telephone Laboratories before joining HP in 1980. At HP's Delcon Division, he worked on measurement techniques for transmission impairment measuring sets and on nonintrusive measurements for data transmission over telephone lines. With HP Labs since 1981, he has has worked on techniques for integrating voice and data on LANs and on hardware and algorithm design for 3D graphics. A member of the IEEE, he has published papers on integrating voice and data in broadband cable systems, optimal digital communications, and the design of a VLSI chip for rendering bicubic patches. Ron suffers from ALS, is quadriplegic, and requires a ventilator to breathe. He is married, has two daughters, and serves on the disability awareness task force of the City of Palo Alto.

Application of Spatial Frequency Methods to Evaluation of Printed Images

Contrast transfer function methods, applied in pairwise comparisons, differentiated between print algorithms, dot sizes, stroke widths, resolutions (dpi), smoothing algorithms, and toners. Machine judgments based on these methods agreed with the print quality judgments of a panel of trained human observers.

by Dale D. Russell

Certain aspects of printed images lend themselves to analysis by spatial frequency methods. The ultimate goal of this type of analysis is to derive a single figure of merit from a test pattern that is sensitive to the overall performance of the printer system.[1] The printer system includes the firmware, hardware, and software, as well as the print engine with its associated colorant/paper interaction.

The value of the modulation transfer function (MTF) in defining optical systems has been demonstrated for decades. As early as 1966, photographic resolving power was shown to be an inadequate measure of system performance and the MTF has been increasingly used.[2] Similarly, the resolution of a printer in terms of dots per inch (dpi) is not adequate to describe the performance and fidelity of the printer through the whole range of spatial frequencies that must be rendered. A consideration of resolution alone fails to take into account either the lower-frequency fidelity or the limiting effect of the human eye.[1]

The MTF generates a curve indicating the degree to which image contrast is reduced as spatial frequency is increased. Unlike resolution, MTF gives a system response with values from zero to a finite number of cycles per millimeter, thus filling in information about the low and middle ranges of the spatial frequency spectrum.

Strictly speaking, continuous methods such as the MTF and the contrast transfer function (CTF) do not apply to discrete systems such as a digital printer, and applications of these functions to discrete systems typically meet with mixed success. The MTF and CTF assume a system that is linear and space and time invariant. Any system with fixed sampling locations (such as a 300-dpi grid) is not space invariant, and sampling theory must be judiciously applied to characterize it. Printers not only digitize data, but most printers binarize it as well, making interpolations of values impossible. This introduces what is essentially a large noise component and gives rise to moire patterns on periodic data.

On the other hand, spatial frequency methods offer a great advantage in that the transfer functions for individual components of a system can be cascaded (i.e., multiplied together) to yield the transfer function of the system (with some exceptions). Provided that a system is close to linear, as it must be if the printed image is to look anything like the intended image, then multiplying component MTFs point by point adequately predicts a complete system MTF.[1] If MTF methods can be adapted to discrete systems, then the overall transfer function will exhibit a sensitivity to changes in the transfer functions of all system components. This sensitivity can be exploited, for example, to diagnose problems or to evaluate new printer designs.

The modulation transfer function is the modulus of the system optical transfer function, which is the Fourier transform of the system point-spread function.[3] While the MTF of a component or system is easier to calculate, experimental work is generally based on measurement of the CTF. This function is then compared to the theoretical performance of a system to yield a single figure of merit for that system. A printer commanded to print a 50% fill pattern consisting of lines will reach a limit of spatial frequency at which it must overprint the designated area. This results in increasing average optical density with increasing spatial frequency, as observed in the test patterns. The CTF is based on contrast, or the difference in reflectance of the printed and unprinted portions of the test pattern. As the white space is increasingly encroached upon by the printed area, or increasingly filled with spray and scatter, the contrast is degraded. This results in a loss of print fidelity and a concomitant decrease in the value of the CTF at that frequency. In the limit, contrast and CTF drop to zero.

In addition to printer limitations, the human eye, with discrete receptors, has a spatial frequency limit of sensitivity. This cutoff point sets a practical limit on the need for improved contrast in a printed image. Furthermore, the contrast sensitivity curve for the human eye, when considered as part of the total system, can be convolved with the CTF curve for the printer to assess the relative importance of improvements in contrast as perceived by the human observer.

Integrating under the CTF curve through the pertinent frequency range gives a single value of the function for that system. This figure can be compared with a standard that represents the theoretical performance to obtain a figure of merit for the system. When the CTF-derived figure of merit correlates with one or more parameters as evaluated by the human observer, then the additional advantage of being able to predict human response with a machine-graded test is realized.

Fourier transform methods are closely related to the MTF and are also applicable to print quality evaluation. Whereas the CTF experiment shows printer performance using a particular test pattern, the Fourier transform can in principle be applied to any printed image. The Fourier transform in this case takes intensity-versus-position data and transforms it to the spatial frequency domain. When a Fourier transform for an image is compared to the Fourier transform for a "standard" image, the dropout areas reveal which frequencies are not transferred by that printer. This can be used to determine limits of printer performance and to identify sources of change in printer performance.

A number of advantages are associated with the use of the Fourier transform for image quality analysis. First, the freedom to select virtually any character or image means that exactly the same image can be presented to both the human and machine observers. Greater control over the experimental variables is possible, since the very same page is evaluated, rather than a test target for the machine vision system and a page of text or graphics for the human. Printing two different patterns on two separate pages at two different times necessarily introduces uncontrolled and even unknown variables that may influence print quality measurements. Use of the Fourier transform for this analysis can eliminate at least some of these variables.

With fast Fourier transform algorithms available today, the time to transform an entire frame of image data is only a minute or so. This makes the time required for analysis by this method considerably less than that for a complete workup of the spatial frequency sweep test target. Given the freedom to select any image for the Fourier transform, attention can be focused on the most egregious visible defects in printer performance. This should further reduce the time required for analysis and ultimately for printer system improvements.

This paper discusses the development and application of various test patterns to black-and-white print quality evaluation with extension to color print quality evaluation. A trained panel of judges evaluated merged text and graphics samples, and their responses are compared with the results of the CTF method. In addition, some examples of the Fourier transform evaluation of printed images are given, and are compared to the information from the CTF method.

Experimental Methods

Three different test patterns were used to derive contrast transfer function data for the printer systems being evaluated. The simplest pattern consists of printed lines paired with unprinted spaces of equal width, in a sequence of increasing spatial frequency. The advantage of this pattern is its simplicity. The principal disadvantage is that it provides information on the printer system only in one axis. Evaluation of the contrast is done using image processing software to generate a line profile normal to the printed lines. Contrast is determined as indicated below.

The second pattern consists of 50% hatched fill patterns at five selected spatial frequencies (0.85 to 2.0 cycles/mm) chosen to lie within the range of human sensitivity (0 to ~4.7 cycles/degree at a viewing distance of 12 in) and to contain the majority of the spatial frequency information for text. Each is presented at seven different angles from the horizontal axis to the vertical axis of the page. This pattern was analyzed by measuring the average optical density and comparing it with the computed theoretical optical density of a 50% pattern, given the paper and colorant used. Patterns of this type are commercially available in print quality analysis packages.

The most complex pattern evaluated consists of concentric rings increasing in spatial frequency from the center out. This pattern includes all print angles relative to the page, but is sensitive to the algorithm used to generate the circle. It provides striking evidence of the impact of the software on the final rendering of a printed sample. In terms of CTF, it reveals the very large contribution of the software to the overall transfer function.

For uniformly sampled spaces, as in a scanner or printer, a circular spatial frequency test pattern gives a visual representation of the system response at all print angles. One effect of the continuously varied spatial frequency mapped onto a fixed grid is the appearance of moire patterns, which indicate beat frequencies between the grid and the desired print frequency.[4] The moire patterns are observed along the pattern axes at frequencies corresponding to division of the printer's capability in dots-per-inch by integer and half-integer numbers. While normally used to test the physical reproduction capability of a grid and spot point function, the circular spatial frequency test pattern also proves useful for evaluating the rendering print quality of graphics algorithms.

The concentric circle test pattern is shown in Fig. 1, which is a 2400-dpi Linotronic imagesetter output described under the heading "Effect of Print Algorithm" below. The white dots on the horizontal axis are frequency markers with a spacing of 0.5 cycles/mm. There are similar markings in the vertical axis with a spacing of 10 cycles/inch. The viewing distance determines the spatial frequency on the human retina in cycles/degree, which is the appropriate unit for dealing with human contrast sensitivity. To convert from cycles/degree to cycles/mm, the following relationship is used:

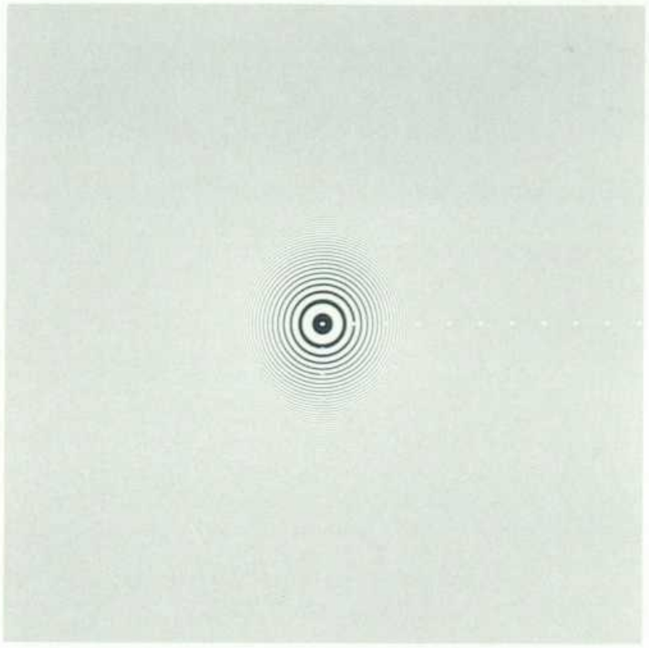


Fig. 1. Concentric circle test pattern. 2400-dpi Linotronic imagesetter output.

$$v = 57.3\frac{\mu}{H}$$

where v is the spatial frequency in cycles/mm, μ is the spatial frequency in cycles/degree, and H is the viewing distance in mm.[5]

Five different black-and-white electrographic printers, with firmware and toner variations of each, were evaluated. Two color printers, electrographic and thermal transfer, were also compared.

The color and black patterns were analyzed using a CCD (charge-coupled device) color video camera vision system and commercially available control software. Digital image processing was performed with commercially available and in-house software. Care was taken to control such experimental variables as the distance from the camera to the paper, the output power of the lamp, the angles of viewing and illumination, and the magnification of the image. Measurements were made in a thermostatically controlled room so that detector noise and dark current would be minimal and relatively constant. Every effort was made to eliminate stray light.

Optical densities of the printed lines and unprinted spaces were determined along with the spatial frequencies. This was done by evaluating a line profile taken normal to the printed line. A contrast function, C, was computed for each line-space pair according to the formula:[6]

$$C = \frac{I_{max} - I_{min}}{I_{max} + I_{min}}$$

where I_{max} and I_{min} are the reflected light intensities of the space and line, respectively, as measured by the video camera for the line profile data. These values had a range of 0 (no measurable reflected light) to 255 (maximum measurable light intensity).

Color patterns were illuminated under filtered light to increase the contrast while keeping reflected intensities within the range of 0 to 255 as measured by the video camera. Therefore, all values reported for the colored samples are relative and not absolute. Data is reported here for only one of the three color channels. A complete analysis would include all three channels. However, we found no case in this study where the inclusion of the other two channels altered a result. This data is normalized and presented as percent modulation on the plots.

By generating rays starting at the center of the test target, line profiles can be taken through as many print angles as desired, for complete analysis of the test pattern. In this work, 10 rays were taken from the center to the edge of the target, in the fourth quadrant, at 10-degree increments starting with the vertical axis and ending with the horizontal axis. The CTF data for all ten rays was computed at the desired frequencies and averaged to obtain the percent modulation as a function of spatial frequency for the sample. The data reported here was all obtained by this method and represents an average of the CTFs at the ten print angles.

Using in-house software, text and "standard" images were transformed into the spatial frequency domain. The standard images were printed on a 2400-dpi imagesetter using scaled bit maps otherwise identical to the test image. Differences between the sample and the standard Fourier transforms were computed and the dropout frequencies noted. These correspond to mathematical notch filters applied to the standard at certain frequencies.

The test image can then be reconstructed by adding the dropout frequencies one at a time to identify which frequencies are responsible for which print defects. The defect frequencies can sometimes be attributed to printer functions, such as gear noise or mechanical vibration frequencies. In this case, a new engine design or materials set will be required to correct the printed image.

Dropout frequencies associated with the sampling frequency of the print grid (i.e., dpi), cannot be corrected without changing the resolution, and thus represent a fundamental limitation. These frequencies can be filled in by various resolution enhancement techniques, or the resolution of the printer must be increased. One application of the Fourier transform method is the immediate evaluation of various resolution enhancement techniques.

Human response to print quality was determined by a committee of 14 trained observers. The committee was shown samples consisting of merged text and graphics for which they graded solid fill homogeneity, contrast, edge roughness, edge sharpness, line resolution, and character density. The committee also gave overall subjective impressions of each sample at the page level, and ranked the samples by making paired comparisons.

Results

Effect of Print Algorithm. A number of different algorithms were examined for preparing the concentric circle test target. Bresenham's algorithm generates the pattern very quickly, but snaps all radii to the nearest integer value.

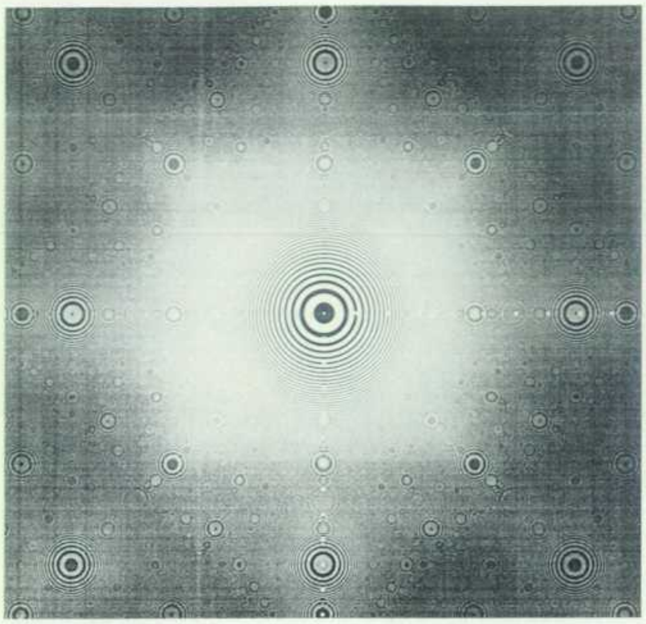


Fig. 2. Test target output from a 300-dpi electrographic printer (printer P) with a good match between dot size and resolution.

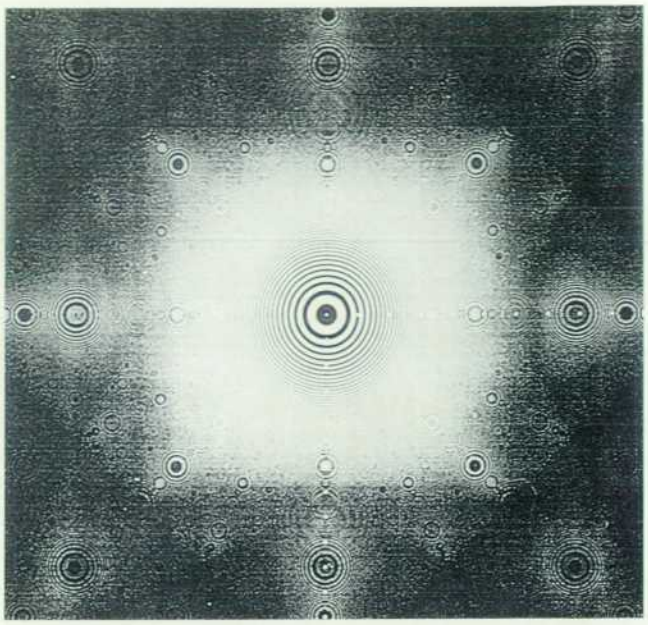


Fig. 3. Test target output from a 300-dpi electrographic printer with a larger dot size (printer R).

Several PostScript interpreters were evaluated; some have floating-point accuracy, others gave unusual renderings. There were considerable differences among them. The test target can also be calculated in polar fashion by incrementing an angle from 0 to 360 degrees and using the sine and cosine of the angle to calculate points to render directly to a bit map. This makes it possible to use the target for print engine and dpi tests.

The method chosen for the rest of this investigation generates a bit map by using a square root function to generate the circles:

$$Y = \mathrm{INT}(\sqrt{\mathrm{Radius}^2 - X^2})$$

for integer values of X. This is computed in one octant of the circle and reflected to the others. Fig. 1 is a 2400-dpi Linotronic output of the test pattern. Differences in print quality arising from the print algorithm alone could have been evaluated using the CTF method outlined in this paper. The choice of an algorithm for printing the concentric circle pattern was based on subjective and qualitative visual ranking of the geometric integrity of test patterns generated as described here.

Effect of DPI, Dot Size, and Edge Smoothing. Increasing the dpi of a printer results in improved CTF through a wider range of spatial frequencies, provided dot size is reduced accordingly. If dot size is held constant, only low-frequency response is improved. Fig. 2 is from a 300-dpi electrographic printer (coded printer P) that has a reasonable match between dot size and resolution. Features are visually distinguished out to the Nyquist frequenc` 150 cycles/mm.

A second 300-dpi printer, coded printer R, has a larger dot size than printer P. Comparison of Fig. 3 with Fig. 2 shows loss of contrast at higher spatial frequencies. It has been calculated that a severe degradation of the MTF results from even a 5% increase in dot size.[5]

Fig. 4 is from printer R with an edge smoothing algorithm applied, and shows improvement at low and middle frequencies. At high frequencies, however, there is actually loss of contrast as the white space between lines is increasingly encroached upon by the thicker, smoothed lines. The main advantage of this particular edge smoothing technique lies in the low to middle frequency regions where most text information is located. When the print

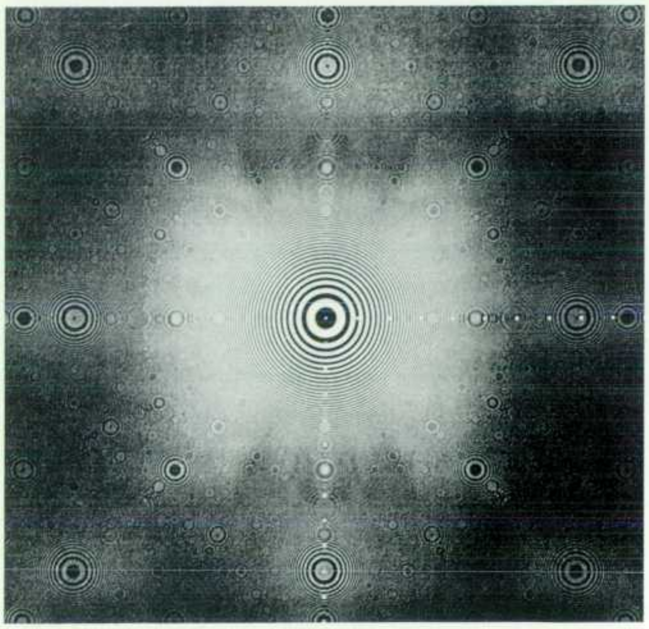


Fig. 4. Test target from printer R with an edge smoothing algorithm applied.

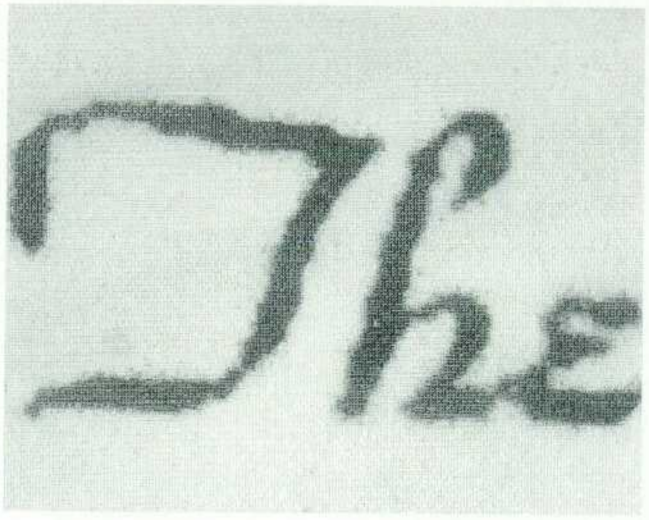


Fig. 5. Enlargement of 300-dpi text (4-point) from printer R without edge smoothing.

quality jury evaluated text only, they unanimously preferred the edge smoothing algorithm. However, when fine lines, 50% fill, and other graphics patterns were included in the samples, the overall preference was in favor of the unsmoothed samples. Figs. 5 and 6 are enlargements of text samples shown to the jury of smoothed and unsmoothed print.

Figs. 7 and 8 are from 400-dpi and 600-dpi printers, respectively. The moire centers are observed to occur at locations on the vertical and horizontal axes corresponding to the dpi value in dots per inch divided by integers and half-integers. As resolution in dpi is improved, the moire patterns have fewer discernible rings and appear at higher frequencies. Print fidelity is therefore better through a broader range of spatial frequencies. Fig. 9 is a plot of the normalized contrast, as percent modulation, for three printers with 300-dpi, 600-dpi, and 1200-dpi resolution. In general, the moire patterns are evidence of

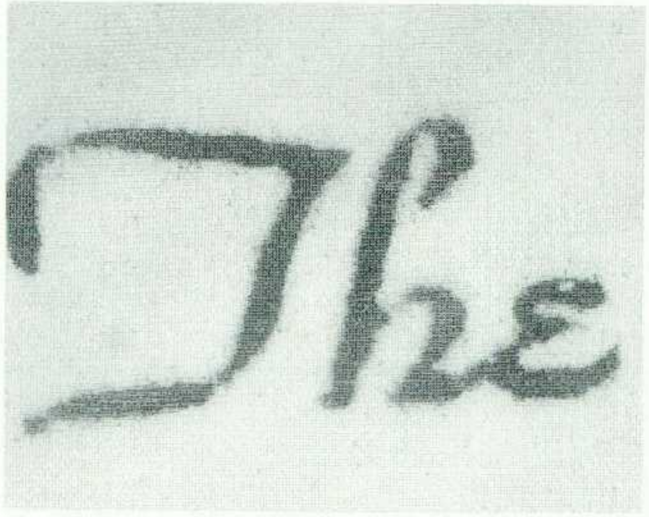


Fig. 6. Enlargement of 300-dpi text (4-point) from printer R with an edge smoothing algorithm applied.

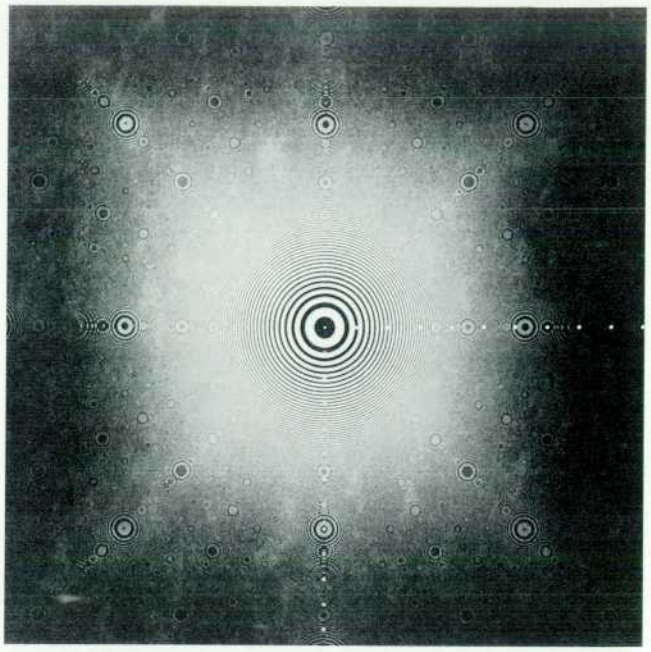


Fig. 7. Test target output from a 400-dpi electrographic printer.

print defects, and measures taken to reduce their visibility in the test target will result in improved fidelity for both text and graphics.[4]

Effect of Toner. Toner particle size can have a measurable impact on print quality,[7] and the CTF method can be used to evaluate this effect. Two special toners were compared with a standard commercially available toner. The special toners were characterized by having smaller average particle size and a narrower particle size distribution. A comparison of Figs. 10 and 2 shows the impact of this on the concentric circle test pattern. The special

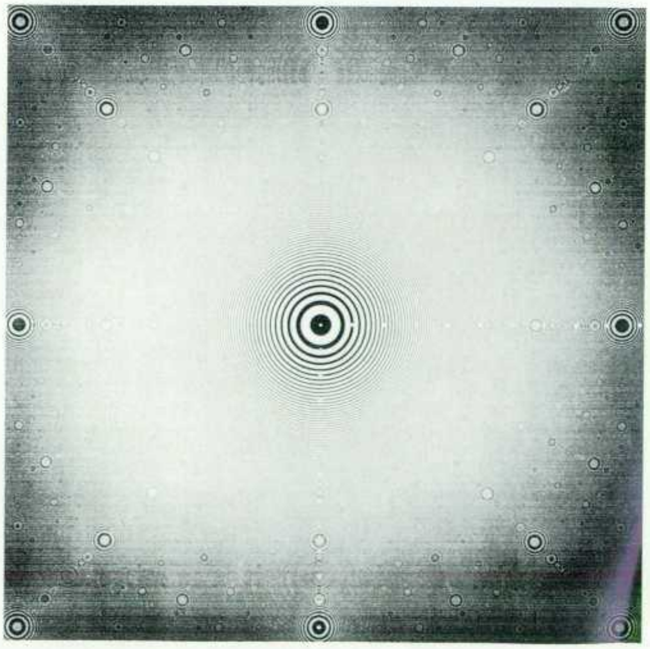


Fig. 8. Test target ouput from a 600-dpi electrographic printer.

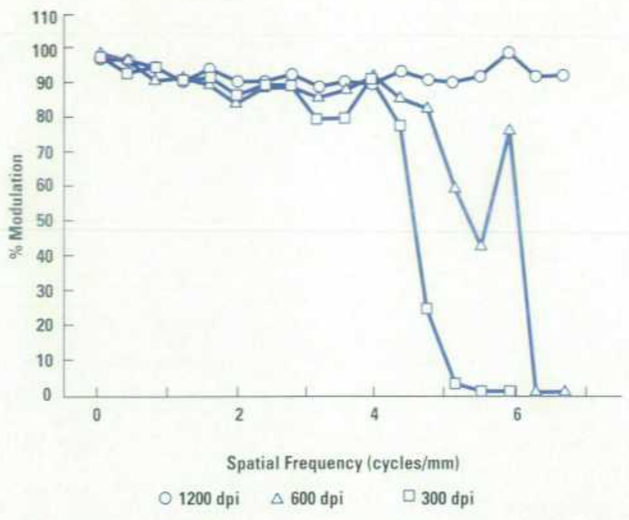


Fig. 9. Percent modulation as a function of spatial frequency for 300-dpi, 600-dpi, and 1200-dpi printers.

toners give smoother line edges, less scatter, and consequently better contrast. The CTF plots in Fig. 11 illustrate the impact of this over the spatial frequency range. The curve for the special toner remains high through the human sensitivity range. The print quality jury invariably preferred the special toners to the standard toner.

Color Samples. Color print samples of the concentric circle pattern were generated by 300-dpi, 400-dpi, and 2400-dpi color printers. Data shown here is for green print samples. The choice of a secondary color introduces the added parameter of color-to-color registration, which can be separately evaluated by the method. The difference in resolution and the wider stroke width of the 300-dpi printer combined to make the 400-dpi printer clearly superior. Text and graphics samples judged by the print quality jury followed the same order of preference. When the 300-dpi printer had its stroke width modified by deleting one pixel width every line, it became the better of the two printers, according to the CTF data. This is illustrated in Fig. 12. Human evaluation gave the same result.

CTF Analysis Compared to Human Perception. Five black-and-white printers and two color printers were evaluated by CTF analysis and print quality jury evaluation. Two CTF methods were compared to human perception. The first was the quick method covering five frequencies and seven print angles, which measured average optical density to approximate the contrast function. This narrow-range method has the advantages of simplicity and speed, and is adequate for many applications. In addition, it has good correlation with the print quality jury findings, approximately 83% for pairwise comparisons. This data is presented in Table I.

Table I
Preferred Graphics Samples
Results of Human vs Machine-Graded Tests

Sample Set	Parameter Tested	Printer	Print Quality Jury	Quick CTF	Concentric Circle Target
1-2	Edge Smoothing	Q	1	1	1
1-8	Toners	Q	1	8	1
3-4	Toners	P	3	3	3
5-6	Dot Size	Q,R	6	6	6
5-8	Edge Smoothing	Q	8	8	8
6-7	Toners	R	7	7	7

"Sample set" refers to the code numbers of the print conditions being compared. The numbers under the method headings are the preferred sample in each set. In the case of the two CTF methods, the area under the CTF curve is the figure of merit used to predict to preferred sample.

The print quality jury consisted of 14 trained observers. The quick CTF test used only 5 spatial frequencies from 0.85 to 2.0 cycles per mm, and only 7 angles of print axis. The concentric circle target used frequencies from 0 to 6.5 cycles per mm, and 11 angles of print axis. Graphics only are considered in this test set. For the machine-graded tests, the integral under the CTF curves was used as a figure of merit to determine which sample was better. The preferred sample in each two-sample set is listed by number in Table I.

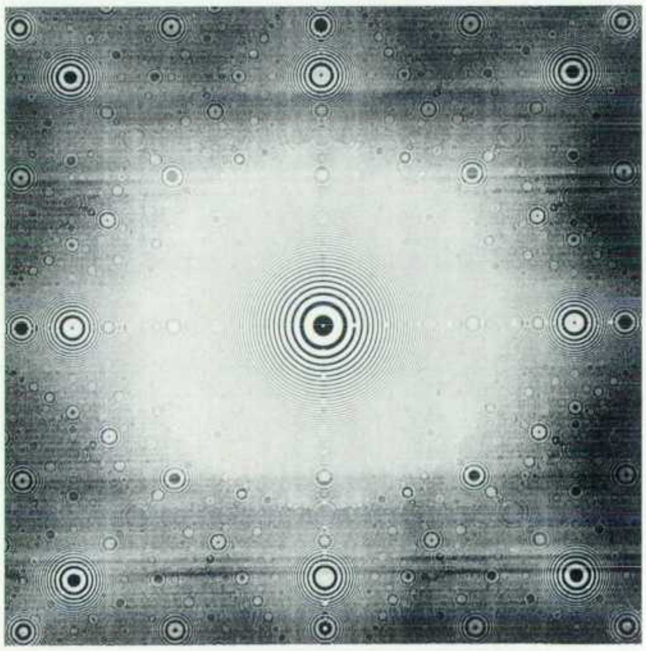


Fig. 10. Test target output from a 300-dpi printer, showing the effect of a special toner.

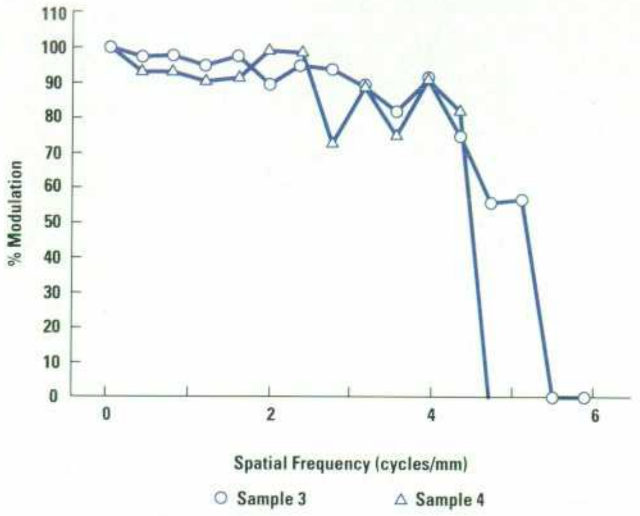


Fig. 11. Plot of percent modulation as a function of spatial frequency for printer P. Sample 3 was printed with a special toner and sample 4 was printed with standard toner.

The concentric circle target method is much more time and labor intensive, but has 100% correlation with the print quality jury for this data set. Since it covers a broader frequency range and more print angles, it does distinguish print fidelity more completely. Paired comparison of samples 8 and 1 (Fig. 13) illustrates this advantage. The quick CTF method predicts that sample 8 is better than sample 1. In the same frequency range, the concentric circle method shows slightly better contrast for sample 1. However, at higher frequencies, the concentric circle pattern reveals significantly better performance for sample 1. The print quality jury preferred sample 1. The frequencies through which sample 1 outperforms sample 8 are within human perception, and apparently correlate with factors that influenced the committee.

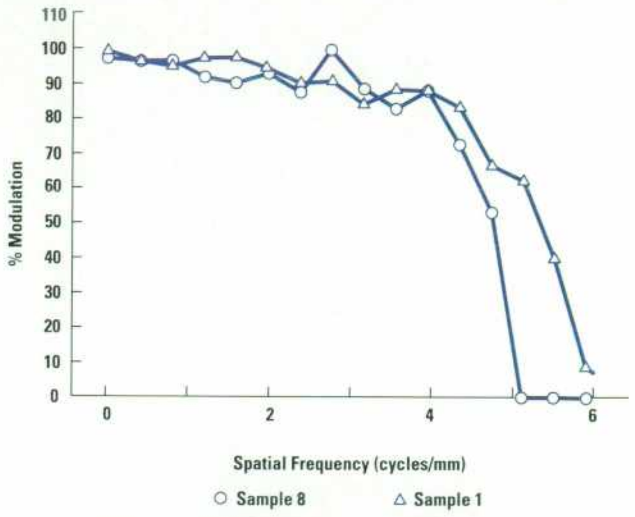


Fig. 13. Plot of percent modulation as a function of spatial frequency for paired comparison of samples 8 and 1. Sample 8 is from printer Q with the edge smoothing algorithm turned off and standard toner. Sample 1 is the same except that special toner was used.

A comparison test of sample 1 against sample 2 also shows this effect (Fig. 14). Based on the magnitude of the integral under the CTF curves, the quick method shows a very slight difference between the samples with 1 better than 2. The concentric circle method, in the same range, also gives sample 1 a very slight edge, but in the higher-frequency region, sample 1 distinctly outperforms sample 2. The print quality jury overwhelmingly preferred sample 1. Apparently, this frequency region is important to human print quality evaluation and should be included in machine-graded tests if the increased likelihood of correlation with human perception justifies the increased time for the test.

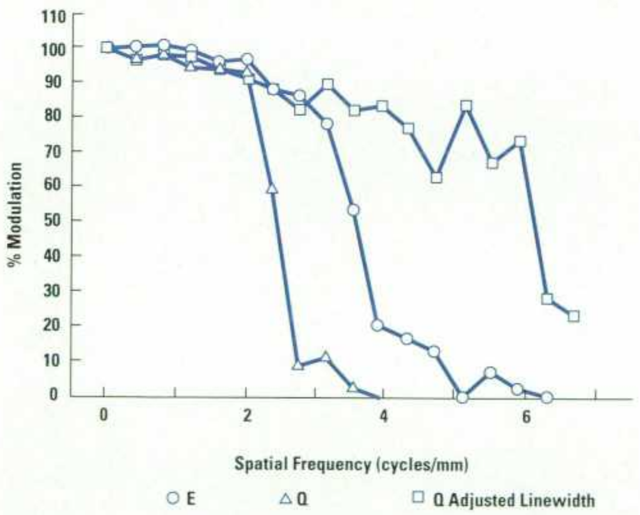


Fig. 12. Plot of percent modulation as a function of spatial frequency for three color (green) test plots. The test pattern for data curve "E" was printed with a 400-dpi color printer. The test pattern for data curve "Q" was printed with a 300-dpi color printer. Curve "Q adj. linewidth" is for the 300 dpi-printer with a narrower linewidth.

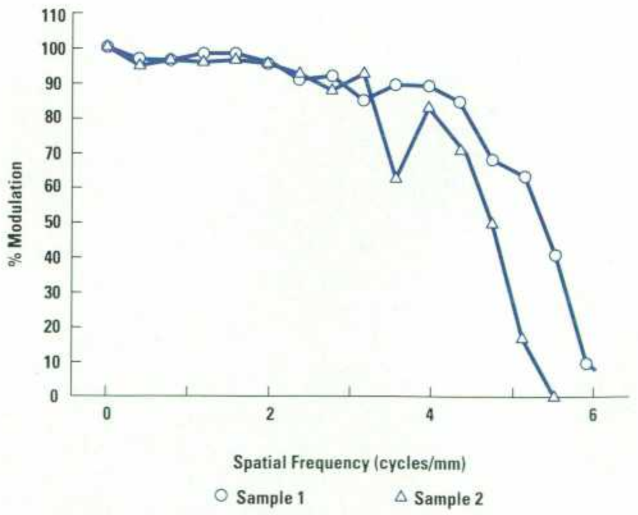


Fig. 14. Plot of percent modulation as a function of spatial frequency for paired comparison of samples 1 and 2. Sample 1 is from printer Q with special toner and resolution enhancement technique off. Sample 2 is the same except that an edge smoothing algorithm is applied.

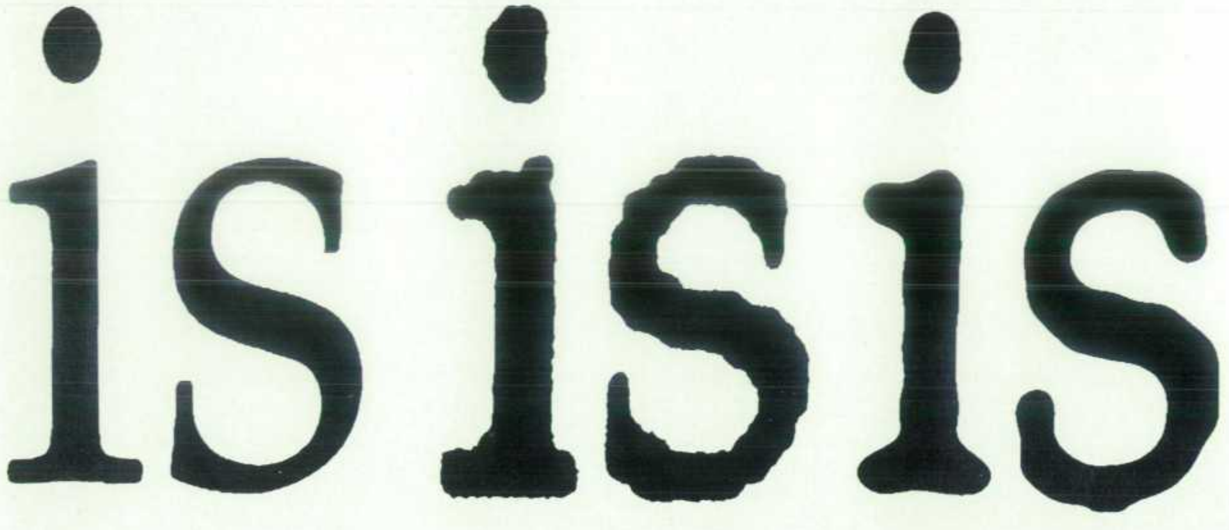


Fig. 15. Enlarged 12-point text showing a 2400-dpi original sample, a 300-dpi original, and the same 2400-dpi sample after being Fourier transformed, filtered, and reconstructed.

Fourier Transform Results. In Fig. 15, three images are compared. The first is a 2400-dpi image, which has been chosen to represent an "ideal" image. The second is 300-dpi output of the same bit map which has been scaled to accommodate the change in addressability. The third is the same 2400-dpi image which has been transformed, filtered, and reconstructed to resemble the 300-dpi image. The filter notched the Fourier transform to approximate the frequency limitations of the 300-dpi printer. Mathematical addition of some of the spatial frequency components back into the notched Fourier transform, with subsequent inverse transformation, shows which frequencies are responsible for which print defects. When the source of the frequency dropout is identified, it can either be corrected or accepted as a fundamental limitation on printer performance. The transforms of the two images may also be subtracted from each other, with the difference corresponding directly to spatial frequency limitations of the 300-dpi printer.

Conclusions

CTF methods, applied here in pairwise comparisons, differentiated between algorithms, dot sizes, stroke widths, dpi, edge smoothing, and toners. In addition, the method shows whether system changes will be expected to improve text, graphics, neither, or both, based on the spatial region in which the CTF response is altered.

The Fourier transform method is useful for identifying spatial frequencies that affect various image characteristics. It also demonstrates usefulness for predicting where the fundamental limitations of the printer have been reached. This will have an impact on engine design.

In all comparisons of printed samples, the results corresponded to the overall subjective preferences of a trained print quality panel. From this it is concluded that this method shows promise as an automated print quality analysis technique, with application to both black and white and color printers.

References

1. E.M. Crane, "Acutance and Granulance," *SPIE*, no. 310, 1981, p. 125.
2. R. Welch, "Modulation Transfer Functions," *Photogrammetric Engineering*, Vol. 37, no. 3, 1971.
3. S.K. Park, R. Schowengerdt, and M.A. Kaczynski, "Modulation Transfer Function Analysis for Sampled Image Systems," *Applied Optics*, Vol. 23, no. 15, 1984.
4. J. Shu, R. Springer, and C.L. Yeh, "Moire Factors and Visibility in Scanned and Printed Halftone Images," *Optical Engineering*, Vol. 28, 1989, p. 805.
5. K.L. Yip and E. Muka, "MTF Analysis and Spot Size Selection for Continuous-Tone Printers," *Journal of Imaging Technology*, Vol. 15, no. 5, 1989.
6. M.C. King and M.R. Goldrick, "Optical MTF Evaluation Techniques for Microelectronic Printers," *Solid State Technology*, Vol. 19, no. 2, 1977.
7. M. Maltz and J. Szczepanik, "MTF Analysis of Xerographic Development and Transfer," *Journal of Imaging Science*, Vol. 25, no. 1, 1988.

Parallel Raytraced Image Generation

Simulations of an experimental parallel processor architecture have demonstrated that four processors can provide a threefold improvement in raytraced image rendering speed compared to sequential rendering.

by Susan S. Spach and Ronald W. Pulleyblank

Computer graphics rendering is the synthesis of an image from a mathematical model of an object contained in a computer. This synthesized image is a two-dimensional rendering of the three-dimensional object. It is created by calculating and displaying the color of every controllable point on the graphics display screen. A typical display contains a grid of 1280 by 1024 of these controllable points, which are called pixels (picture elements). The memory used to store pixel colors is called the frame buffer. Specialized hardware accelerators available on today's workstations, such as HP's Turbo SRX and Turbo VRX products,[1] can render models composed of polygons in real time. This makes it possible for the user to alter the model and see the results immediately. Real-time animation of computer models is also possible.

The most time-consuming operation in rendering is the computation of the light arriving at the visible surface points of the object that correspond to the pixels on the screen. Real-time graphics accelerators do this by transforming polygonalized objects in the model to a desired position and view, calculating an illumination value at the polygon vertices, projecting the objects onto a 2D plane representing the display screen, and interpolating the vertex colors to all the pixels within the resulting 2D polygons. This amounts to approximating the true surface illumination with a simplified direct lighting model.

Direct lighting models only take into account the light sources that directly illuminate a surface point, while global illumination models attempt to account for the interchange of light between all surfaces in the scene. Global illumination models result in more accurate images than direct lighting models. Images produced with global lighting models are often called photorealistic.

Fig. 1 shows the contrast between hardware shading and photorealistic renderings. Fig. 1a was computed using a local illumination model while Figs. 1b, 1c, and 1d were computed using global illumination algorithms.

The disadvantage of photorealistic renderings is that they are computationally intensive tasks requiring minutes for simple models and hours for complex models.

Raytracing is one photorealistic rendering technique that generates images containing shadows, reflections, and transparencies. Raytracing is used in many graphics applications including computer-aided design, scientific visualization, and computer animation. It is also used as a tool for solving problems in geometric algorithms such as evaluation of constructive solid geometry models and geometric form factors for radiative energy transfer.

The goal of our research is to develop parallel raytracing techniques that render large data models in the fastest possible times. Our parallel raytracing techniques are being implemented to run on the Image Compute Engine (ICE) architecture. ICE, under development in our project group at HP Laboratories, is a multiprocessor system intended to accelerate a variety of graphics and image processing applications. ICE consists of clusters of floating-point processing elements, each cluster containing four processors with local and shared memory. The clusters are networked using message passing links and the system topology is configured using a crossbar switch. A prototype system of eight clusters is under construction. Data distribution, load balancing, and algorithms possessing a good balance between computation and message passing are research topics in our parallel implementation.

Raytracing Overview

Generation of synthetic images using the raytracing technique was introduced by Appel[2] and MAGI[3] in 1968 and then extended by Whitted in 1980.[4] Raytracing is a method for computing global illumination models. It determines surface visibilities, computes shading for direct illumination, and computes an approximation to the global illumination problem by calculating reflections, refractions, and shadows. The algorithm traces simulated light rays throughout a scene of objects. The set of rays reaching the view position is used to calculate the illumination values for the screen pixels. These rays are traced backwards from the center of projection through the viewing plane into the environment. This approach makes it unnecessary to compute all the rays (an infinite number) in the scene. Only a finite number of rays needed for viewing are computed.

An observer view position (the center of projection or "eye" position) and a view plane are specified by the user (Fig. 2). The raytracer begins by dividing a window on the view plane into a rectangular grid of points that correspond to pixels on the screen and then proceeds to determine the visibility of surfaces. For each pixel, an eye ray is traced from the center of projection through the

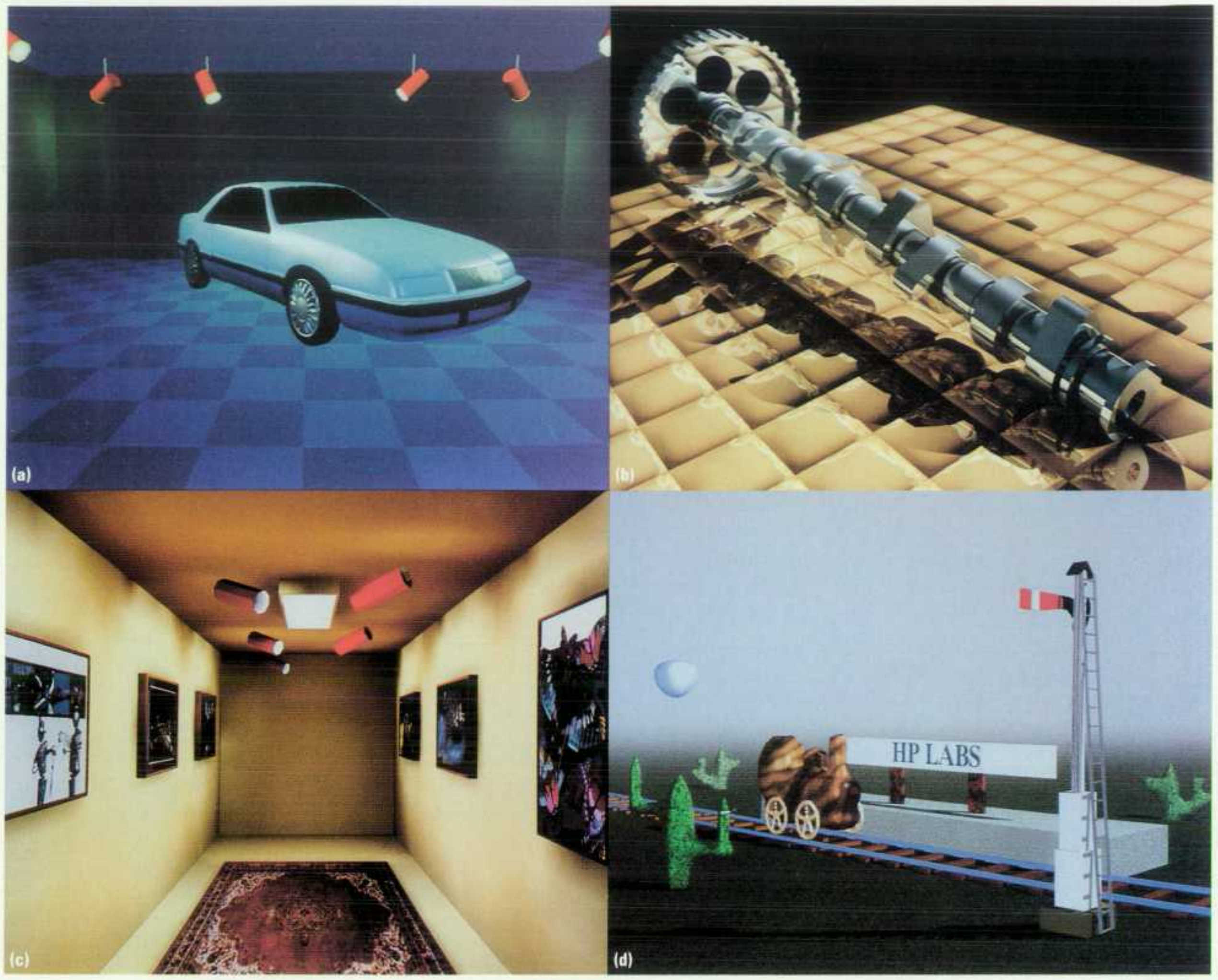


Fig. 1. (a) A scene computed using a local illumination model. (b) (c) (d) Photorealistic renderings computed using global illumination algorithms.

pixel out into the scene environment. The closest ray/object intersection is the visible point to be displayed at that pixel. For each visible light source in the scene, the direct illumination is computed at the point of intersection using surface physics equations. The resulting illumination value contributes to the value of the color of the pixel. These rays are referred to as primary rays.

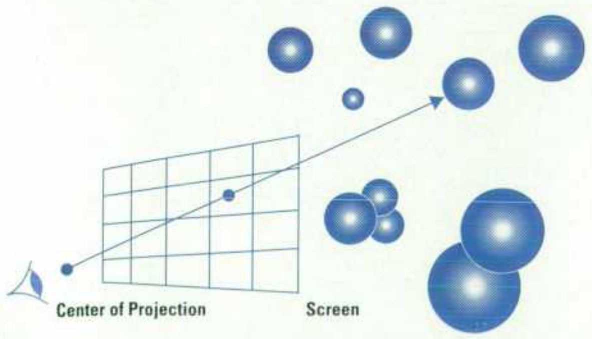


Fig. 2. The raytracing technique traces rays of light from the viewer's eye (center of projection) to objects in the scene.

The raytracing algorithm proceeds to calculate whether or not a point is in shadow. A point is not in shadow if the point is visible from the light source. This is determined by sending a ray from the point of intersection to the light source. If the ray intersects an opaque object on the way, the point is in shadow and the contribution of the shadow ray's light source to the surface illumination is not computed. However, if no objects intersect the ray, the point is visible to the light source and the light contribution is computed. Fig. 3a illustrates shadow processing. The point on the sphere surface receives light from light source A, but not from light source B.

A ray leaving the surface toward the view position has three components: diffuse reflection, specular reflection, and a transmitted component. Specular and transmitted rays are determined by the direction of the incoming ray and the laws of reflection and refraction. The light emitted by these rays is computed in the same manner as the primary ray and contributes to the pixel corresponding to the primary ray. Figs. 3b and 3c show the reflection and transmitted rays of several objects in a scene. Diffuse reflection (the scattering of light equally in all

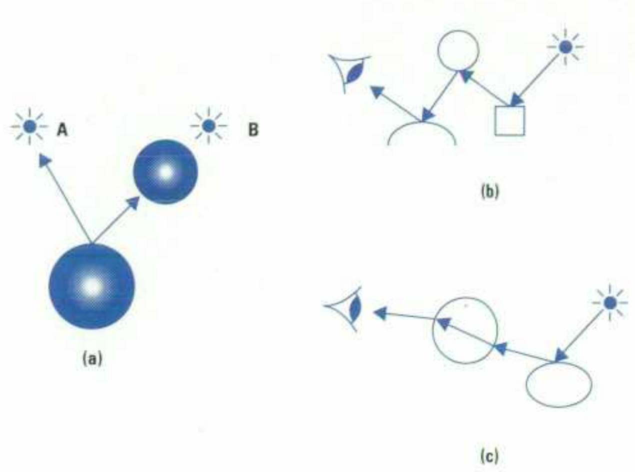


Fig. 3. Types of rays. (a) Shadow. (b) Reflection. (c) Refraction.

directions) is approximated by a constant value. Accurate computation of the diffuse component requires the solving of energy balance equations as is done in the radiosity rendering algorithm.[5,6,7] Diffuse interreflections can also be approximated using raytracing techniques[8,9] but this requires excessive computation.

The raytracing algorithm is applied recursively at each intersection point to generate new shadow, reflection, and refraction rays. Fig. 4 shows the light rays for an environment. The rays form a ray tree as shown in Fig. 5. The nodes represent illumination values and the branches include all secondary rays generated from the primary ray. Conceptually, the tree is evaluated in bottom-up order with the parent's node value being a function of its children's illumination. The weighted sum of all the node colors defines the color of a pixel. A user-defined maximum tree depth is commonly used to limit the size of the tree. It is evident from Figs. 4 and 5 that shadow rays dominate the total ray distribution.

The basic operations in raytracing consist of generating rays and intersecting the rays with objects in the scene. An advantage of raytracing is that it is easy to incorporate many different types of primitives such as polygons, spheres, cylinders, and more complex shapes such as parametric surfaces and fractal surfaces. The only requirement to be able to use an object type is that there be a procedure for intersecting the object with a ray. One of the main challenges in raytracing is making the ray intersection operation efficient. Algorithmic techniques have been developed that include faster ray-object intersections, data structures to limit the number of ray-object intersections, sampling techniques to generate fewer rays, and faster hardware using distributed and parallel processing.[10,11,12] Our research effort concentrates on using data structures to limit the number of ray-object intersections and on using parallel techniques to accelerate the overall process.

Spatial Subdivision

Spatial subdivision data structures are one way to help limit the number of intersections by selecting relevant objects along the path of a ray as good candidates for ray intersection. Spatial subdivision methods partition a volume of space bounding the scene into smaller volumes, called voxels. Each voxel contains a list of objects wholly or partially within that voxel. This structuring yields a three-dimensional sort of the objects and allows the objects to be accessed in order along the ray path.

We employ a spatial subdivision technique termed the hierarchical uniform grid[13] as the method of choice. This approach divides the world cube bounding the scene into a uniform three-dimensional grid with each voxel containing a list of the objects within the voxel (Fig. 6a). If a voxel contains too many objects, it is subdivided into a uniform grid of its own (Fig. 6b). Areas of the scene that are more populated are more finely subdivided, resulting in a hierarchy of grids that adapts to local scene complexities.

The hierarchical uniform grid is used by the raytracer to choose which objects to intersect. We find the voxel in the grid that is first intersected by the ray. If that voxel contains objects, we intersect the ray with those objects. If one or more intersections occur within the voxel, the closest intersection to the ray origin is the visible point and secondary rays are spawned. If there are no intersections or if the voxel is empty, we traverse the grid, to the next voxel and intersect the objects in the new voxel (Fig. 7a). The ray procedure ends if we exit the grid,

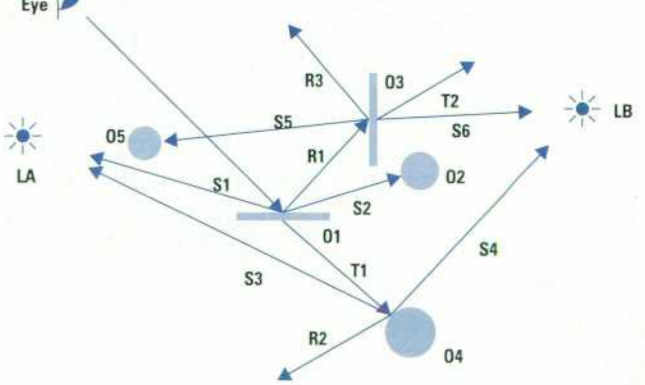


Fig. 4. Light sources, objects, and rays for an environment.

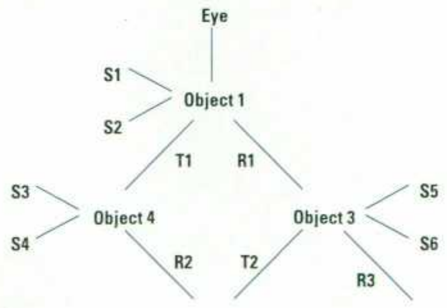
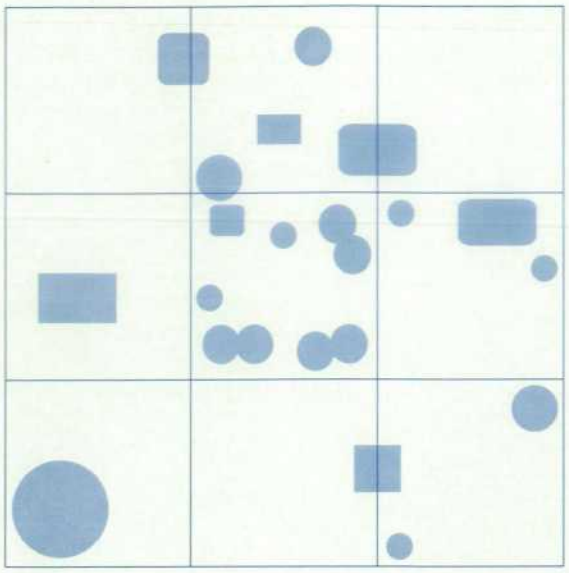
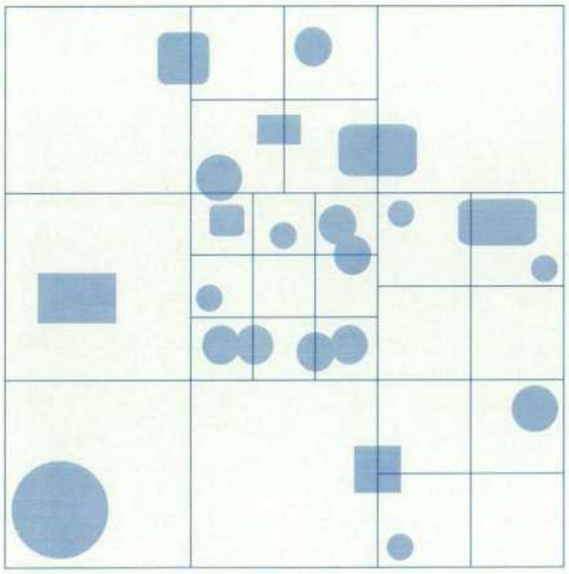


Fig. 5. Ray tree for the environment of Fig. 4.



(a)



(b)

Fig. 6. The hierarchical uniform grid spatial subdivision technique. (a) The world cube surrounding the scene is divided into a uniform three-dimensional grid of volumes called voxels. (b) If a voxel contains too many objects, it is subdivided into a uniform grid of its own.

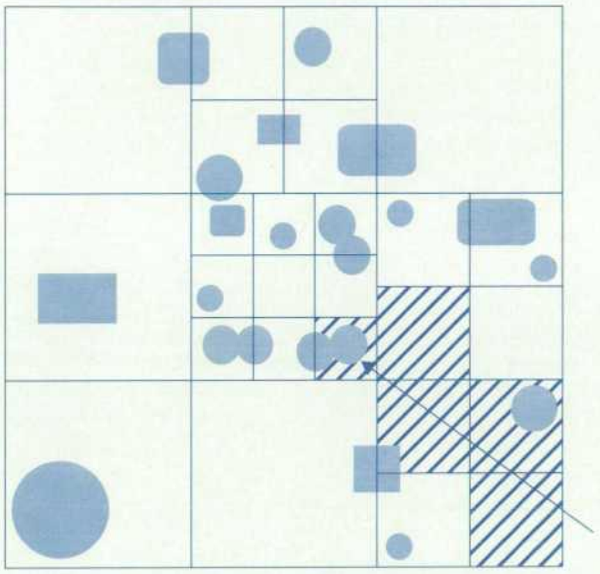
indicating that no object in the scene intersects the ray (Fig. 7b).

Grid traversal is fast because the grid is uniform, allowing the use of an incremental algorithm for traversing from voxel to voxel. There is a penalty for moving up and down the hierarchy to different grids but this is the cost of having the data structure efficiently adapt to the scene complexity.

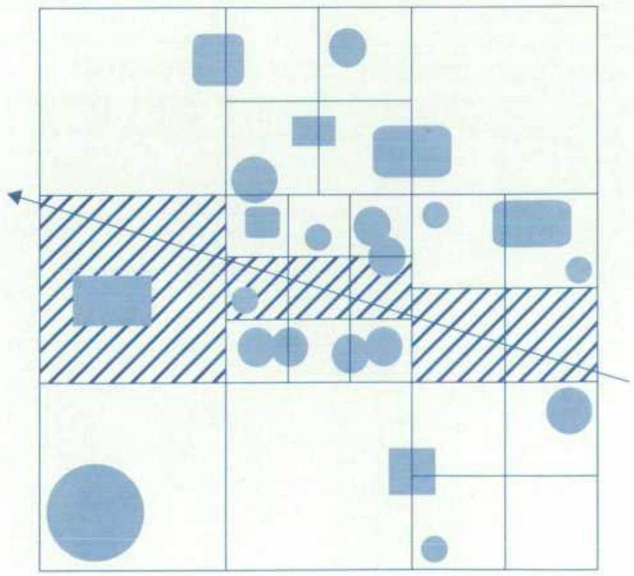
Adjacent voxels are likely to contain the same object because objects may overlap several voxels. Two critical implementation details are included to avoid erroneous results and repeated ray intersections. First, the intersection point of an object must occur within the current voxel. Second, intersection records, containing the ID of the last ray intersected with that object and the result, are stored with the object to prevent repeated intersection calculations of the same ray with the same object as the ray traverses the grid.

ICE Overview

ICE is a parallel architecture composed of clusters of floating-point processors. Each cluster has four processors and roughly 64M bytes of shared memory accessible for reading and writing by all four processors. Each processor has 4M bytes of local memory, which is used to hold private data and program code. The clusters



(a)



(b)

Fig. 7. Hierarchical uniform grid traversal. (a) Hit: the ray intersects an object. (b) Miss: no ray-object intersection.

communicate using message passing links and the system topology is configurable with a crossbar switch. There is a data path from the common display buffer to the cluster's shared memory. Fig. 8 shows the ICE architecture.

Each shared memory can be configured to hold frame buffer data and/or can be used to hold data accessible by all four processors. The frame buffer data can be configured as a complete 1280-by-1024 double buffered frame buffer of RGB and Z values or a rectangular block subset of the frame buffer. The message passing links are used for intercluster communication and access to common resources such as a disk array. The host workstation can broadcast into all local and shared memories via the message passing links.

The frame buffers in each cluster's shared memory are connected by custom compositing hardware to a double buffered display frame buffer which is connected to a monitor. The compositing hardware removes the bottleneck from the cluster frame buffers to the single display frame buffer. The compositing hardware can function in three different modes: Z buffer mode, alpha blend mode, and screen space subdivision mode.

In Z buffer mode, the compositing hardware simultaneously accesses the same pixel in all the cluster frame buffers, selects the one with the smallest Z, and stores it in the display buffer. This mode is used for Z-buffered polygon scan conversion.

In alpha blend mode the same pixel is simultaneously accessed in all cluster frame buffers. Pixels from adjacent data blocks are sorted into nearest and farthest and blended using the blending rule: $\alpha \times \text{nearest} + (1-\alpha) \times \text{farthest}$. The final result is a blend of pixels from all the clusters and is presented to the display buffer. This mode is used in volumetric rendering of sampled data.

In screen space subdivision mode, each cluster contains pixels from a subset of the screen and the compositing hardware simply gathers the pixels from the appropriate cluster. This mode is used in raytracing applications.

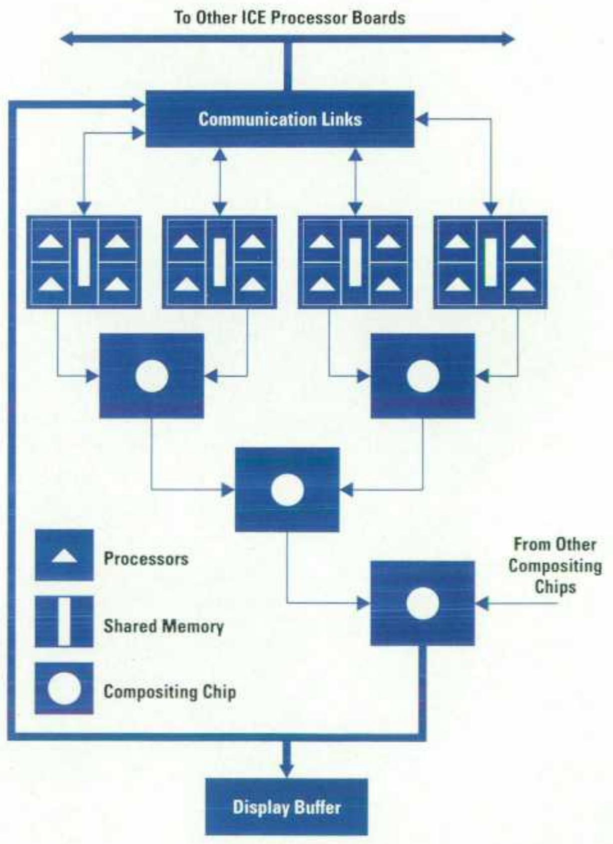


Fig. 8. Image Compute Engine (ICE) architecture.

Parallel Raytracing on ICE

Raytracing is well suited for parallelization because the task consists mainly of intersecting millions of independent rays with objects in the model. Much research in recent years has concentrated on using multiprocessors to speed up the computation. Two approaches have been used to partition the problem among the processors: image space subdivision and object space subdivision.[14, 15, 16, 17, 18, 19]

In image space subdivision, processor nodes (clusters) are allocated a subset of the rays to compute and the entire data set is stored at each node. While this method achieves almost linear speed increases, it is not a feasible solution for rendering data sets that require more memory than is available on a processing node. With object space methods, computations (rays) and object data are both distributed to processing nodes and coordination between them is accomplished through message passing between clusters. We have chosen an object space subdivision approach for implementation on ICE because of its ability to handle very large data sets.

Parallel object space subdivision is efficient if it results in low interprocessor communication and low processor idle time. As we partition the computation and object data, several decisions need to be made. How are the object data, ray computations, and frame buffer pixels distributed among the processor nodes? How are ray computations and the corresponding object data brought together? How is load balancing accomplished?

The screen is subdivided into blocks of pixels which are assigned to clusters where they are stored in the shared memory. When the picture is complete these pixels are gathered into the display frame buffer by the custom compositing chips.

The spatial subdivision grid data structure is stored at every processing node. Voxels for which the data is not stored locally are designated as remote voxels. The data associated with the voxels in the grid data structure is distributed among the clusters in a way that attempts to statically balance the computational load among the processor clusters. This is accomplished by grouping adjacent voxels into blocks and distributing the blocks among clusters so that each cluster contains many blocks selected uniformly from throughout the model space. Voxels distributed in this manner to a cluster are called the primary voxels for that cluster. Voxels are distributed in blocks to maintain coherence along a ray and reduce intercluster communication (it is likely that the next voxel will be on the same cluster for several iterations of grid traversal).

The distribution of the voxels of a grid is performed for all the grids in the hierarchy so that all portions of the model that have a great deal of complexity are subdivided into voxels and distributed throughout the network of clusters. Thus, no matter where the viewpoint may be, whether zoomed in or not, the objects in the view, and thus the computational load, should be well-distributed among the clusters of processors.

When the data is distributed among processing nodes as it is for large data sets, and we trace a ray through our grid data structure, we may come to a voxel in which the data resides on a different processing node, that is, a remote voxel. At this point we send the ray information to the processing node that contains the data and the computation continues there.

If the data associated with the primary distribution of the data set does not fill up a cluster's shared memory, additional data, which duplicates data in another cluster, is added. The first additional data added is data used to speed up shadow testing. This is important because shadow rays are launched from every ray-object intersection point towards every light source, creating congestion at voxels containing light sources. To alleviate this, the data in voxels nearest a light source that fall wholly or partially within the cones defined by the light source (cone vertex) and the cluster's primary voxels (cone bases) are added to the data stored within the cluster's shared memory. If there is still space available in shared memory after the shadow data is added, voxel data from voxels adjacent to the cluster's primary voxel blocks is added. If there is space enough to store the complete data set in every cluster, that is done.

Each processor within a cluster maintains a workpool, located in group shared memory, of jobs defined by either a primary or a secondary ray. As new rays are formed they are placed in a processor's workpool. When a processor finds its workpool empty it takes jobs from its neighbor's workpool. This organization is intended to keep processors working on different parts of the database to minimize group shared memory access conflicts.

Each cluster is responsible for determining which primary rays originate in its primary voxels and initializing its workpools accordingly. This can be done with knowledge of the viewing parameters by projecting the faces of certain primary voxels (those on faces of the world cube facing the eye position) onto the screen and noting which pixels are covered. Jobs consisting of primary rays are listed as runs on scan lines to minimize the job creation time.

A ray is taken from the workpool by a processor in the cluster, which attempts to compute the remainder of the ray tree. Any rays, primary or secondary, that cannot be processed at a cluster because it does not contain the necessary voxel and its associated model data are forwarded to the cluster that contains the required voxel as a primary voxel. A queue of rays is maintained for each possible destination cluster; these are periodically bundled into packets and sent out over the message passing links.

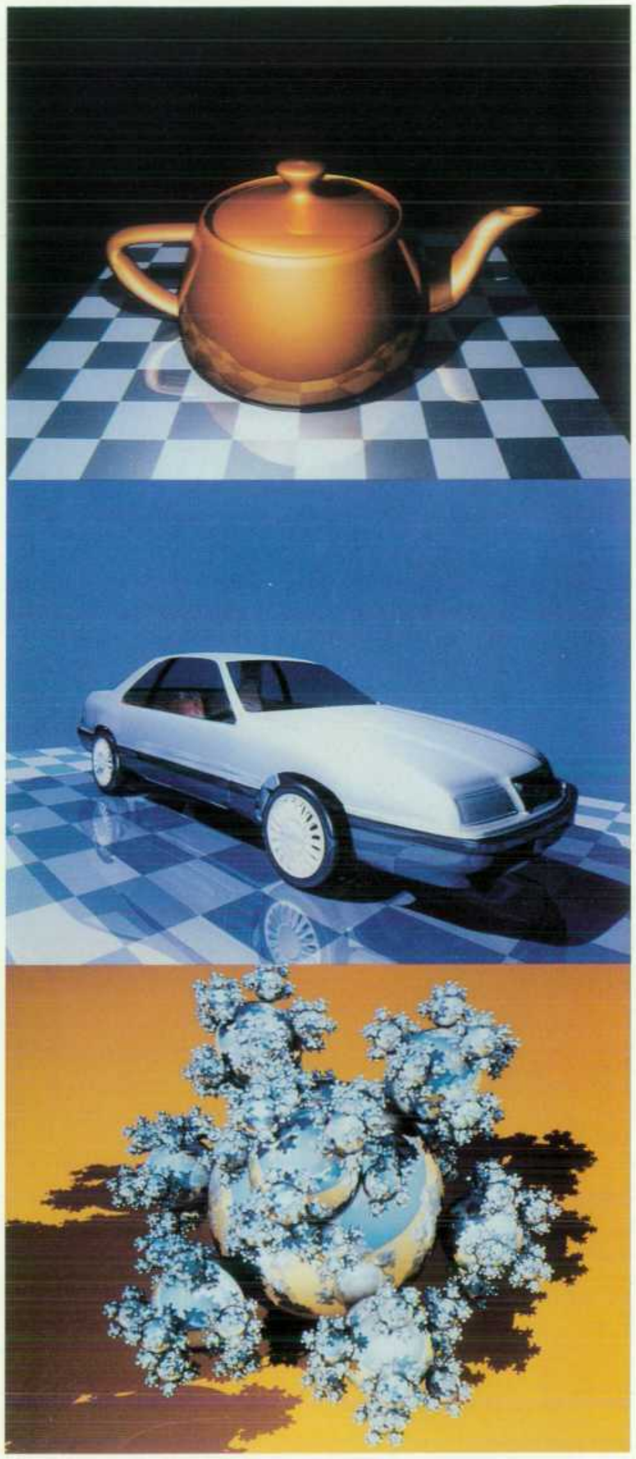


Fig. 9. Three scenes used to measure the rendering speed improvement of parallel processing over sequential processing.

Each ray includes information about what pixel its color contribution must be accumulated in. These color contributions of rays may be computed in any of the clusters but the results are sent to the cluster that has responsibility for the portion of the frame buffer containing that pixel. There, the contributions are accumulated in the pixel memory.

Raytracing completion is determined using a scoreboarding technique. The host computer keeps a count of rays created and rays completed. Clusters send a message to the host when a ray is to be created, and the host increments its count of rays created. Similarly, when a ray completes in a cluster, the cluster tells the host and the host increments its count of rays completed. When these two counts are equal, the rendering job is done and the compositing hardware, operating in screen space subdivision mode, transfers all the frame buffer data from each cluster group shared memory to the display frame buffer.

When static load balancing by uniform distribution of data among clusters and dynamic load balancing by commonly accessible workpools within clusters are inadequate, then dynamic load balancing between clusters is carried out. Our plan for accomplishing this is to create workpools of rays for mutually exclusive blocks of voxels in each cluster. Rays are placed on the voxel workpool according to which voxel the ray has just entered. These workpools are organized as a linked list. Processors get a voxel workpool from the linked list for processing. In this way, processors are working on different regions of the data set, thereby reducing contention for group shared memory. When a cluster exhausts its workpools it asks the other clusters for information on their workloads and requests from the busiest cluster one of its unused workpools together with its associated data.

Results

The ICE hardware, currently under construction, is expected to be completed in the spring of 1992. Parallel raytracing software in C has been written and simulations on an Apollo DN10000 have been performed. The DN10000 workstation has four processors and 128M bytes of shared memory, similar to one cluster on ICE.

The DN10000 software includes a parallel programming toolset based on the Argonne National Laboratories macro set.[20] This toolset includes macros for implementing task creations and memory synchronization. Our simulation is of one cluster with workpools for dynamic load balancing within a cluster. It is capable of rendering objects composed of polygons and spheres.

Fig. 9 shows three scenes that were rendered sequentially and with the parallel software on the DN10000. The teapot and the car are B-spline surface objects that have been tessellated into polygons. The teapot contains 3600 polygons, the car contains 46,000 polygons, and the sphereflake contains 7300 spheres. Table I gives the rendering times in seconds for a screen of 500 by 500 pixels. Each scene experienced at least a threefold speed improvement using four processors.

Table I
Results
(500 by 500 pixels)

	1 Processor	4 Processors	Improvement
Teapot	422 s	130 s	3.2
Car	879 s	288 s	3.0
Sphereflake	1458 s	392 s	3.7

Conclusions and Future Work

An overview of the raytracing algorithm and a discussion of a parallel implementation of raytracing for the ICE architecture have been presented. A first version of the parallel software is running on an Apollo DN10000 yielding a threefold improvement in speed over the sequential software. The DN10000 simulations provide a vehicle for parallel code development and statistical gathering of scene renderings. A version of the multicluster software is being written on the DN10000 to develop code for simulation of message passing and load balancing. We will have a version of the code to port directly to the ICE architecture when the hardware is finished.

ICE will provide a platform for parallel algorithm development and experimentation for a variety of graphics applications. Raytracing characteristics, such as grid size, ray tree depth, ray type distribution (shadow, reflection, refraction), and required interprocessor communication bandwidths, are scene dependent, making any sort of theoretical analysis difficult. The goal of our future work is an extremely fast implementation of raytracing capable of handling very large data sets. At the same time, we would like to develop an understanding of how best to distribute data and perform load balancing on the ICE architecture.

Acknowledgments

Mark Smith is the main hardware designer for the ICE project with support from Jim Christy, Manu Thapar, and Darrin Young. Tom Malzbender designed the compositing chips and is developing volume rendering algorithms for ICE. David Kirk provided the DN10000 parallel toolset.

References

1. R. Swanson and L. Thayer, "A Fast Shaded-Polygon Renderer," *Computer Graphics (SIGGRAPH 1986 Proceedings)*, Vol. 20, no. 4, 1986, pp. 95-101.
2. A. Appel, "Some Techniques for Shading Machine Renderings of Solids," *Proceedings of the AFIPS 1968 Spring Joint Computer Conference*, Vol. 32, 1968, pp. 37-45.
3. Mathematical Applications Group, Inc., "3D Simulated Graphics," *Datamation*, February 1968.

4. T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, Vol. 23, no. 6, June 1980, pp. 343-349.
5. C. Goral, K. Torrance, and D. Greenberg, "Modeling the Interaction of Light Between Diffuse Surfaces," *Computer Graphics (SIGGRAPH 1984 Proceedings)*, Vol. 18, no. 3, 1984, pp. 213-222.
6. M. Cohen, et al, "The Hemi-cube: A Radiosity Solution for Complex Environments," *Computer Graphics (SIGGRAPH 1985 Proceedings)*, Vol. 19, no. 4, 1985, pp. 31-40.
7. M. Cohen, et al, "A Progressive Refinement Approach to Fast Radiosity Image Generation," *Computer Graphics (SIGGRAPH 1988 Proceedings)*, Vol. 22, no. 4, 1988, pp. 75-84.
8. J.T. Kajiya, "The Rendering Equation," *Computer Graphics (SIGGRAPH 1986 Proceedings)*, Vol. 20, no. 4, 1986, pp. 143-150.
9. G. Ward, et al, "A Ray Tracing Solution for Diffuse Interreflection," *Computer Graphics (SIGGRAPH 1988 Proceedings)*, Vol. 22, no. 4, 1988, pp. 85-92.
10. M.A.J. Sweeney, and R.H. Bartels, "Ray Tracing Free-Form B-spline Surfaces," *IEEE Computer Graphics and Applications*, Vol. 6, no. 2, February 1986.
11. P. Hanrahan, "Raytracing Algebraic Surfaces," *Computer Graphics (SIGGRAPH 1983 Proceedings)*, Vol. 17, no. 3, 1983, pp. 83-89.
12. D. Kirk, and J. Arvo, "A Survey of Raytracing Acceleration Techniques," *Introduction to Raytracing*, Academic Press, 1989, pp. 201-262.
13. D. Jevans and B. Wyvill, "Adaptive Voxel Subdivision for Ray Tracing," *Computer Graphics Interface 89*, 1989.
14. M. Dippe and J. Swensen, "An Adaptive Subdivison Algorithm and Parallel Architecture for Realistic Image Synthesis," *Computer Graphics (SIGGRAPH 1984 Proceedings)*, Vol. 18, no. 3, 1984, pp. 149-158.
15. J.G. Cleary, B.M. Wyvill, G.M. Birtwistle, and R. Vatti, "Multiprocessor Ray Tracing," *Computer Graphics Forum*, Vol. 5, 1986, pp. 3-12.
16. H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, and Y. Shigei, "Load Balancing Strategies for a Parallel Ray-Traced System Based on Constant Subdivision," *The Visual Computer*, no. 4, 1988, pp. 197-209
17. S.A. Green, D.J. Paddon, and E. Lewis, "A Parallel Algorithm and Tree-Based Computer Architecture for Ray Traced Computer Graphics," *International Conference, University of Leeds*, 1988.
18. S.A. Green and D.J. Paddon, "A Highly Flexible Multiprocessor Solution for Raytracing," *The Visual Computer*, no. 6, 1990, pp. 62-73.
19. V. Isle, C. Aykanat, and B. Ozguc, "Subdivision of 3D Space Based on Graph Partitioning for Parallel Ray Tracing," *Proceedings of the Eurographics Workshop on Image Synthesis*, 1991.
20. J. Boyle et al, *Portable Programs for Parallel Processors*, Holt, Rinehart, and Winston, Inc., 1987.

Fr: SUSAN WRIGHT / 97LDC 00054984
446

To: LEWIS, KAREN
HP CORPORATE HEADQUARTERS
DDIV 0000 20BR

HEWLETT-PACKARD
JOURNAL

June 1992 Volume 43 • Number 3

Technical Information from the Laboratories of Hewlett-Packard Company

Hewlett-Packard Company, P.O. Box 51827
Palo Alto, California, 94303-0724 U.S.A.

Yokogawa-Hewlett-Packard Ltd., Suginami-Ku Tokyo 168 Japan

Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive, Mississauga, Ontario L4V 1M8 Canada

HEWLETT
PACKARD

5091-4263E