POWER CHALLENGE™

Technical Report

Chapter 1 Overview

In the last fourteen years, Silicon Graphics has supplied engineers, scientists, and creative professionals with hardware and software solutions that have redefined the state of the art. Silicon Graphics POWER CHALLENGE supercomputing servers, successors to the precedent-setting POWER Series™ of shared-memory multiprocessing systems, use faster, denser CMOS VLSI technology to continue leading the industry in symmetric multiprocessing (SMP).

1.1 Innovative Supercomputing Design

Figure 1-1 summarizes the Silicon Graphics corporate supercomputing vision.

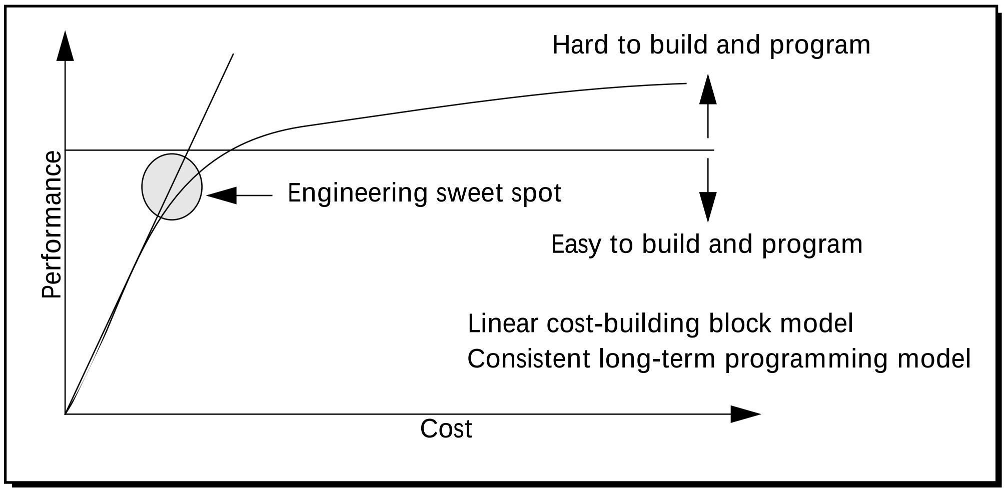

Hard to build and program
Engineering sweet spot
Easy to build and program
Linear cost-building block model
Consistent long-term programming model
Performance
Cost

Figure 1-1 Silicon Graphics Supercomputing Vision

It plots the achievable performance of computers against the cost of building the systems. The knee of the curve represents the point of diminishing marginal returns. To get higher performance, the designer ends up paying very high costs.

To achieve higher and higher system performance, some designers have either resorted to exotic device technologies, such as ECL or gallium arsenide, or have chosen an inherently expensive topology to interconnect processors.

Silicon Graphics aims to build systems using mass-produced technology and to push the technology just far enough so that the marginal returns on investment are significantly high. Silicon Graphics ability to build supercomputers out of esoteric technologies derives from certain trends in technology, such as:

- Continuing improvements in integrated circuit technology and computer architecture have driven microprocessors to performance levels that rival those of traditional supercomputers, at a fraction of the price.
- The use of sophisticated memory hierarchies enables microprocessor-based machines to have very large memories built from commodity DRAMs while retaining the high bandwidth and low access time needed in a high-performance machine.
- Advanced optimization techniques in compiler technology that maximize the utilization of the system resources have become available.

These technology trends are discussed in greater detail in Chapter 2, “The Silicon Graphics Supercomputing Philosophy.”

1.2 History of High-End Computing at Silicon Graphics

One of the first companies to embrace RISC microprocessor-based system design, Silicon Graphics introduced shared-memory multiprocessing systems in 1988.

Beginning with a two-way symmetric shared-memory multiprocessor using the 16 MHz MIPS R3000 microprocessor, the POWER Series of multiprocessor systems was enhanced in computing power to contain up to eight 40 MHz MIPS R3000® microprocessors.

Figure 1-2 on page 3 shows the history of high-end computing at Silicon Graphics.

Here are some highlights of the evolution of multiprocessing systems at Silicon Graphics:

- IRIX has its roots in the AT&T UNIX system 5. IRIX 3 and IRIX 4 were SVR3-based operating systems. IRIX 4, the second-generation implementation, featured symmetric multiprocessing feature enhancements and refinements. The IRIX operating system implements sophisticated features in the process scheduler for resource utilization in a multiprocessing complex. Both IRIX 5 and IRIX 6 are SVR4-based operating systems. IRIX 6 is a true 64-bit operating system that maintains complete forward binary compatibility with earlier IRIX releases.
- The compiler family has evolved through three generations.

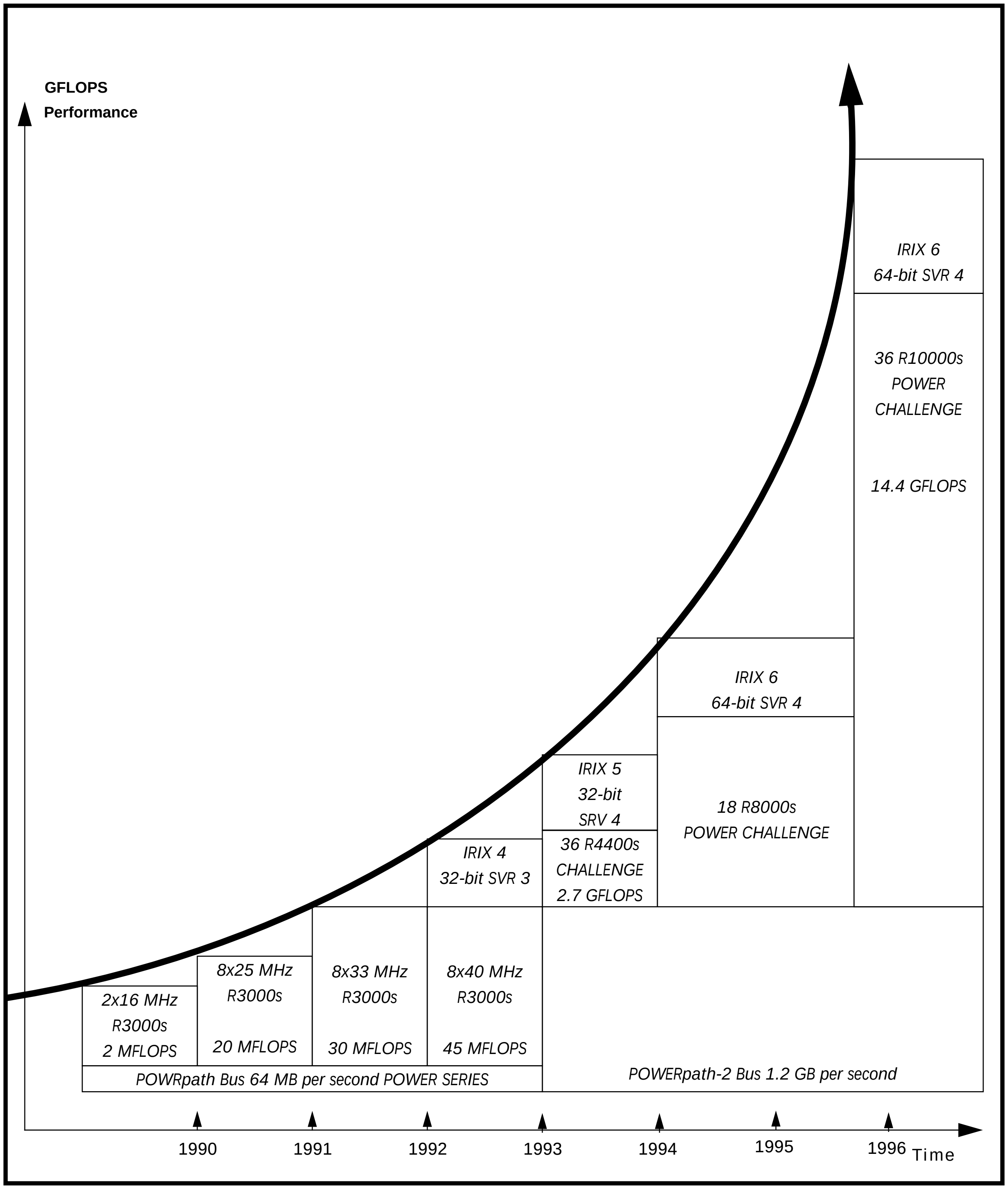
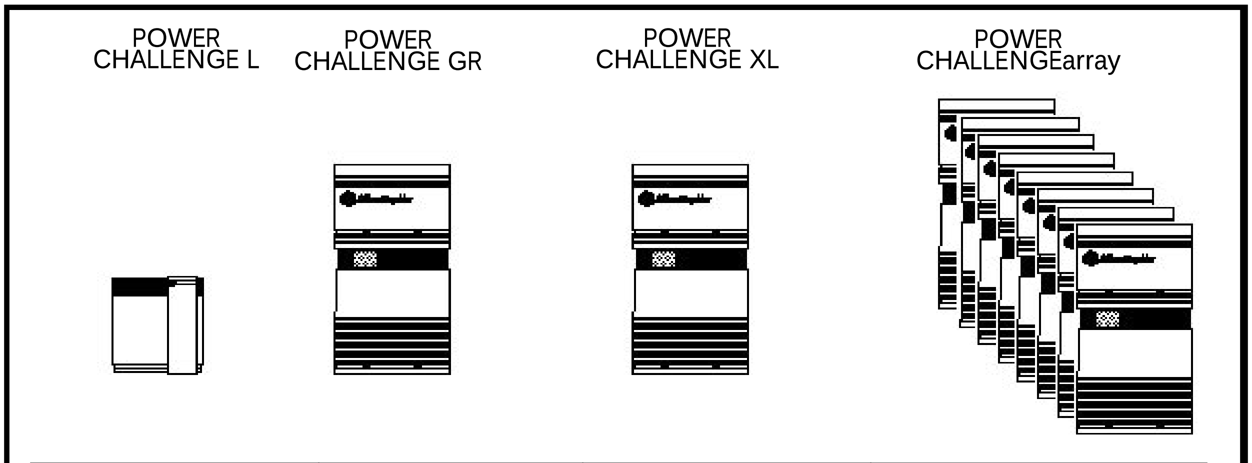


Figure 1-2 The Growth of High-end Computing at Silicon Graphics

- The system hardware has evolved through two generations: POWERpath and POWERpath-2 architectures. Noteworthy points in this evolution include:
  - Bandwidth of the system bus has increased twentyfold, from 64MB (megabytes) per second on the POWERpath bus to the 1.2GB (gigabytes) per second on the POWERpath-2 bus.
  - Floating-point performance has increased by a factor of 7200 from the original POWER SERIES systems to the POWER CHALLENGE XL servers.
  - Memory capacity has increased from 256MB on the POWER SERIES servers to 16GB of eight-way interleaved memory on POWER CHALLENGE multiprocessors.
  - I/O bandwidth was increased from a maximum of 256 MB per second on the POWER SERIES to 1.2GB per second on the POWER CHALLENGE multiprocessors. Connectivity options, including very high-performance FDDI and HiPPI, were greatly enhanced. Disk capacity has increased from a few hundred GB to 17TB (terabytes) on POWER CHALLENGE XL.

1.3 Product Line Overview

The POWER CHALLENGE product line includes the low-priced deskside POWER CHALLENGE L, the mid-range, graphics-ready POWER CHALLENGE GR, the highly expandable rack POWER CHALLENGE XL, and the multiple-cabinet, high-scalability POWER CHALLENGEarray™. All POWER CHALLENGE servers feature MIPS R10000 and R8000™ superscalar CMOS RISC CPUs. Figure 1-3 compares the four POWER CHALLENGE systems.



POWER CHALLENGE L	POWER CHALLENGE GR	POWER CHALLENGE XL	POWER CHALLENGEarray
2 to 12 R10000's–4.8 GFLOPS or 1 to 6 R8000'sfl2.16 GFLOPS up to 6 GB of memory up to 960 per second I/O	2 to 24 R10000's – 9.6 GFLOPS or 1 to 12 R8000s – 4.3 GFLOPS up to 16 GB of memory up to 1 GB per second I/O	2 to 36 R10000's – More than 13.5 GFLOPS or 2 to 18 R8000s – 5.4 GFLOPS up to 16 GB of memory up to 1.2 GB per second I/O	2 to 288 R10000's – 115.2 GFLOPS or 2 to 144 R8000s – 51.8 GFLOPS up to 128 GB of memory up to 9.6 GB per second I/O

Figure 1-3 POWER CHALLENGE Systems

Highlights of these systems include:

- POWER CHALLENGE L is a deskside multiprocessor system that expands from two to 12 CPUs with secondary caches of either 1 MB or 4 MB. Memory can be expanded from 64 MB up to six GB and can be four-way interleaved. The system uses the Silicon Graphics HIO bus for high-performance I/O. Up to three such buses can be configured, each with a sustained bandwidth of 320MB per second for a total of 960MB per second of I/O bandwidth. I/O can be expanded through the industry-standard VME64.
- POWER CHALLENGE GR is a rack configuration supporting from two to 24 CPUs. Memory expandability ranges from 64MB to 16GB, and it can be eight-way interleaved. With as many as four HIO busses, the GR's maximum I/O bandwidth is 1.2 GB per second. Because POWER CHALLENGE GR is graphics-ready, it can be upgraded with Onyx® InfiniteReality™ or Reality Engine2 subsystems at the factory or in the field. (For more information on advanced graphics subsystems from Silicon Graphics, refer to the *Onyx® Technical Report*.) I/O can be expanded through the industry-standard VME64.
- The POWER CHALLENGE XL is a highly-configurable, high-end, rack multiprocessor system that can be expanded from two to 36 CPUs. Memory expandability is from 64MB to 16GB and can be eight-way interleaved. This system can be configured with up to six HIO busses. I/O can be expanded through the industry-standard VME64.
- POWER CHALLENGEarray is the Grand CHALLENGE class system, expandable from eight to 288 CPUs. Memory is expandable up to 128GB, and I/O up to 9.6GB per second. POWERnodes in the array can include POWER CHALLENGE L, XL, and GR configurations. For more information on the POWER CHALLENGEarray, see the *POWER CHALLENGEarray Technical Report.*

All POWER CHALLENGE servers run the 64-bit IRIX 6 operating system and the MIPSPro compiler suite as well as supporting integrated visualization capabilities with the Extreme™ Visualization Console.

1.4 Usage Environments

POWER CHALLENGE systems and their software are designed to provide supercomputing solutions for you in a variety of computational environments. POWER CHALLENGE multiprocessor servers can be configured to suit very high computational, memory, or I/O demands depending on the work load. Typically, these servers are used in:

- Desktop supercomputing
- Dedicated project supercomputing
- Departmental supercomputing
- Enterprise supercomputing
- Grand challenge computing
- Interactive visual supercomputing

1.4.1 Desktop Supercomputing

In today's computing environment, supercomputing has found its way to the desktop. For many applications, single-threaded, compute-intensive jobs can be effectively computed within the confines of a single desktop workstation. Alternatively, several independent jobs may be dispersed to systems accessible through a local area network to achieve a simple level of coarse-grained parallelism. High performance can be achieved in R10000 and R8000-based systems, including POWER Indigo^2, Indigo^2 IMPACT, Infinity Station, and POWER CHALLENGE L with Visualization Console. Figure 1-4 depicts such a configuration.

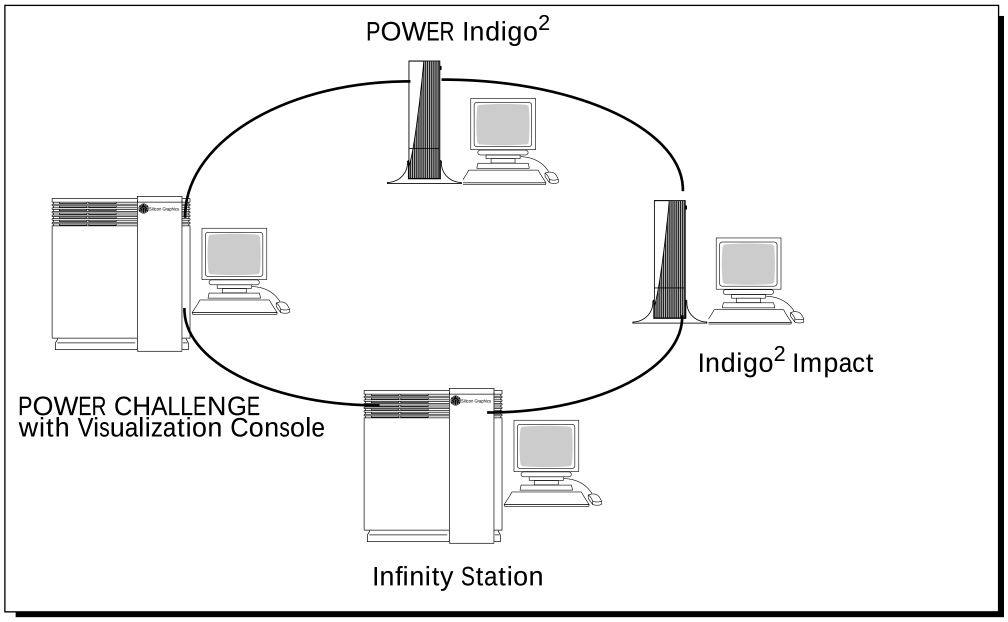


Figure 1-4 Desktop Supercomputing

1.4.2 Dedicated Project Supercomputing

Dedicated project supercomputing is perhaps the best example of deployable supercomputing: individual projects can deploy one or more POWER CHALLENGE supercomputers to solve a particular kind of problem for an organization. For example, a POWER CHALLENGE can be dedicated to simulating molecular dynamics. Such a machine would run the same simulation codes again and again for a team of four to eight molecular biologists, providing a very low time to solution. Figure 1-5 diagrams dedicated project supercomputing

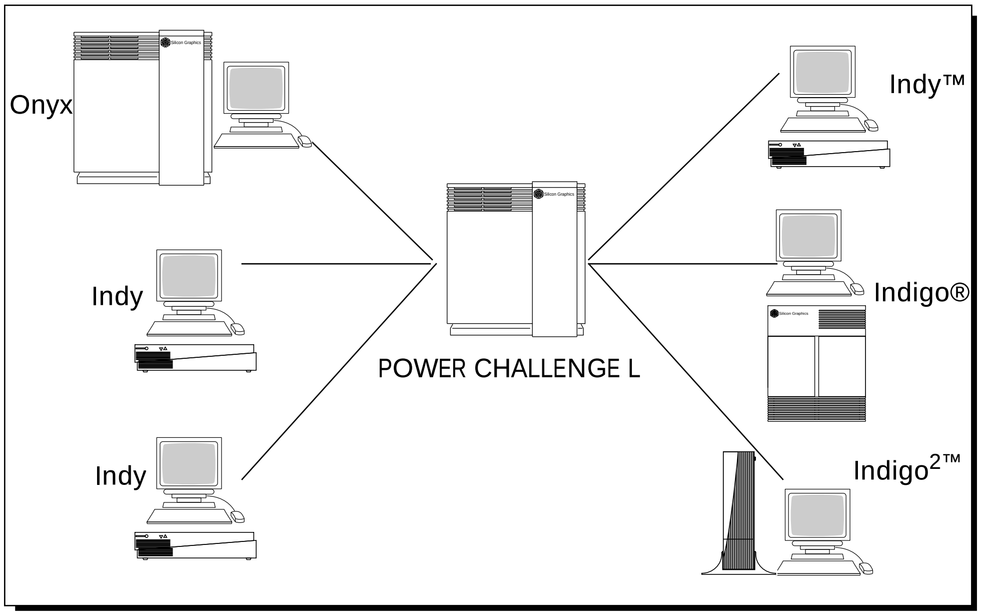


Figure 1-5 Dedicated Project Supercomputing

1.4.3 Departmental Supercomputing

The multidisciplinary needs of departments are amply met by the scalable POWER CHALLENGE machines. A typical example would be an automobile corporation, with a department of 20 to 30 people doing structural analysis as well as safety design of automobiles, and interacting with other corporate departments to share design data. The growing computational needs of the departments can easily be met by scaling the CPUs or the number of chassis configured. A rich selection of network connectivity—Ethernet, FDDI, HiPPI, or ATM—provides low-cost or high-performance connections to file servers and mass storage. The agility of configurations permitted by POWER CHALLENGE multiprocessors makes the computational center architect's job easy. Figure 1-6 on page 8 diagrams departmental supercomputing.

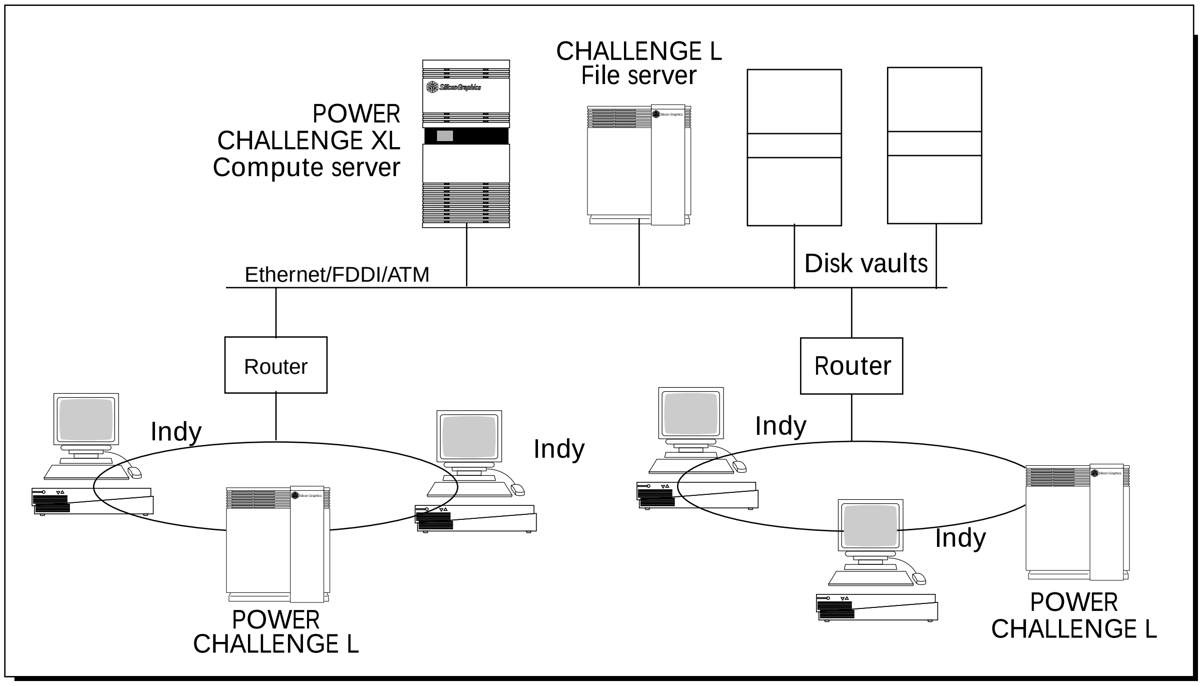


Figure 1-6 Departmental Supercomputing

1.4.4 Enterprise Supercomputing

Either a single large POWER CHALLENGE XL server or a number of smaller machines can be employed as shown with file servers and visual client workstations using high-performance networks as an enterprise-wide computing solution. The highly scalable I/O capability of POWER CHALLENGE servers, via either the HIO bus or VME64, offers connectivity options such as Ethernet, FDDI, HiPPI, SCSI-2, and ATM to hundreds of client workstations, file servers, high-performance rotating media and data archival and retrieval systems. Figure 1-7 on page 9 diagrams enterprise supercomputing.

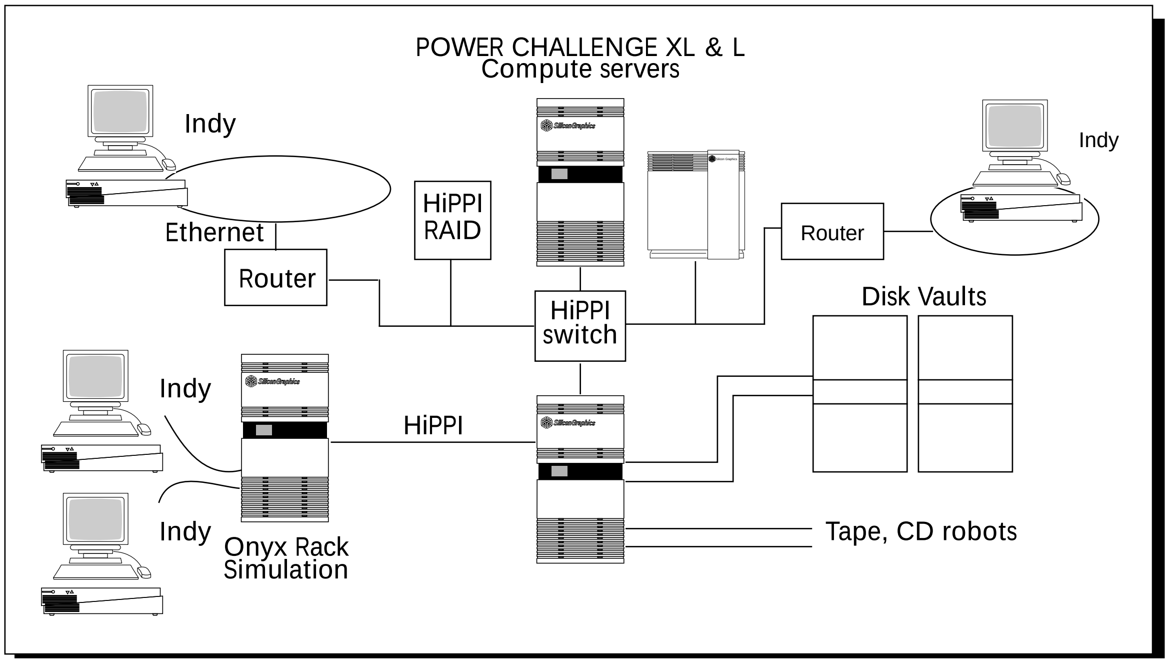


Figure 1-7 Enterprise Supercomputing

1.4.5 Grand Challenge Supercomputing

Multiple POWER CHALLENGE systems may be tightly coupled across HiPPI interconnects to form a POWER CHALLENGEarray, as shown in Figure 1-8. Each SMP-based POWERnode exploits fine-grained shared-memory parallelism internally as well as message-passing for communication among nodes. High-bandwidth interfaces support visualization and disk subsystems for visualizing and archiving computational results.

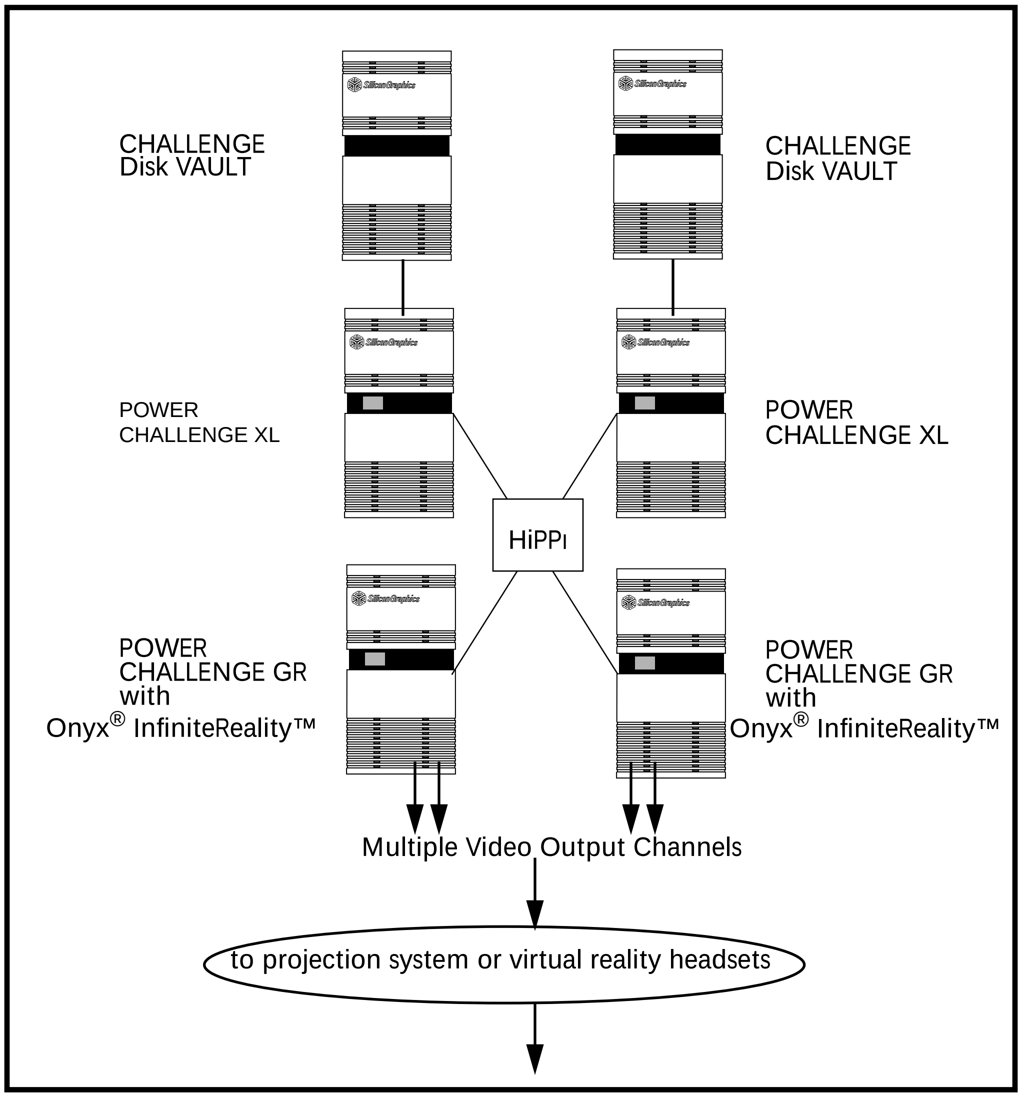

CHALLENGE Disk VAULT
CHALLENGE Disk VAULT
POWER CHALLENGE XL
POWER CHALLENGE XL
HiPPI
POWER CHALLENGE GR with Onyx® InfiniteReality™
POWER CHALLENGE GR with Onyx® InfiniteReality™
Multiple Video Output Channels
to projection system or virtual reality headsets

Figure 1-8 Using POWER CHALLENGEarray to Solve Grand Challenges

1.4.6 Interactive Visual Supercomputing

In many environments, tight integration of supercomputing and visualization technology is being exploited to define a new class of applications that allow users to interact with computations as they are being performed. Computational steering applications optimize time-to-insight by removing the batch/queue paradigm, thus providing individual users with full command of the computational resource through a high-performance, interactive, visualization interface.

In such environments, a POWER CHALLENGE GR configured with Onyx® InfiniteReality™ or RealityEngine^2 graphics provides a unique solution by integrating the visualization and computational subsystems into a single, high-performance system solution.

Chapter 2 The Silicon Graphics Supercomputing Philosophy

The Silicon Graphics supercomputer design is founded upon these beliefs:

- Riding trends in integrated circuit technology, RISC microprocessors, and memory hierarchies can result in very high-performance supercomputers.
- It is essential to build scalable, high-performance, easily programmable supercomputers. A shared-memory programming model is common to all Silicon Graphics multiprocessor platforms.
- Building very high-performance supercomputers that *scale down* in price is important. This desire has direct implications for achieving the highest performance, which the most demanding applications require.
- Users should be able to develop applications on inexpensive, binary-compatible, low-end platforms and deploy them on high-performance supercomputers. High-bandwidth interconnects are needed to upload and download data sets to binary-compatible, state-of-the-art graphics workstations and high-performance file servers.
- High-performance visualization subsystems should be tightly integrated into the design of the supercomputer to enable new and emerging interactive supercomputing applications.

Silicon Graphics continues to capitalize on trends in technology. This chapter discusses:

- Trends in integrated circuit technology
- High-performance microprocessors
- Meeting memory demands
- Shared-memory programming
- Designing for economic scalability

2.5 Trends in Integrated Circuit Technology*

For the last 30 years, improvements in integrated circuit technology have increased computer performance and capability. During this time, continuous reductions in the size of transistors and wires have made it possible to increase the number of devices that can be put on a single silicon die. Further improvements in manufacturing technology have allowed die sizes to increase because the number of flaws is now much lower.

Decreases in the size of devices and lengths of wires also lead to faster transistors and faster interconnections, which thus lead to improvements in the speed of the chip. Thus, the chip

* This section was excerpted from “Microprocessors: From Desktops to Supercomputers,” by Forest Baskett and John L. Hennessey, *Science*, August 13, 1993, Volume 261, pp. 864-871.

architect is motivated to find ways to improve performance that take advantage of this increase in transistor count as well as the faster transistors.

The impact of these dramatic improvements in density is most easily seen in memory technology. Memory technology is of two main types: dynamic random access memory (DRAM) and static random access memory (SRAM). DRAM uses fewer transistors per bit of memory (one transistor versus four to six for SRAM) and thus is higher density. New memory technology offering a factor-of-four improvement in density has been introduced every three years, leading to tremendous increases in the number of bits per chip, as shown in Figure 2-1.

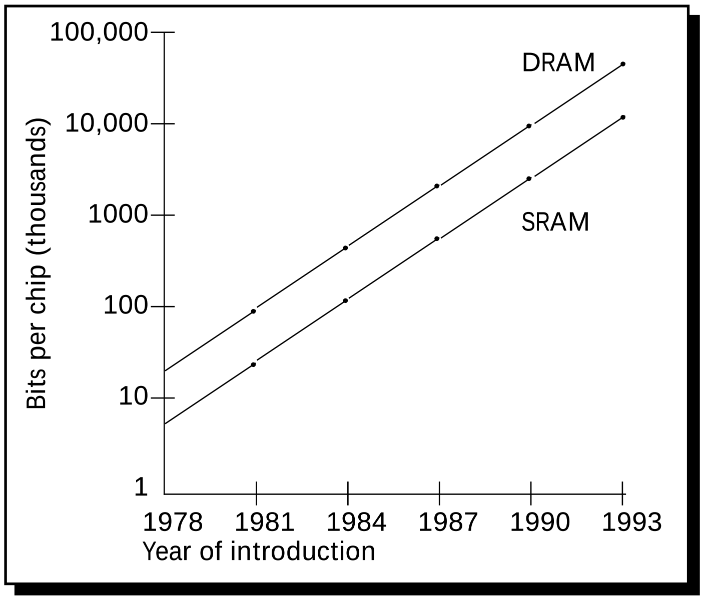


Figure 2-1 Growth in Number of Bits on Single DRAM and SRAM Chips

Because of the advantage in density, DRAMs are cheaper per bit. In addition, since DRAMs have become the primary technology for building main memories, they also have the benefit of high-volume production, costing less per bit than their advantage in density, typically six to ten times less per bit. Thus, for large main memories on computers ranging from PCs to supercomputers, DRAM is the technology of choice for building main memory.

Prices per bit from DRAM have decreased at roughly the same rate as density has increased. The commodity price for DRAMs in 1993 was about $25 per MB, which means that each 1 MB chip is selling for about $3. By 1995, this price dropped to less than $10 per MB.

However, DRAM has a disadvantage: it is significantly slower than SRAM. This speed difference arises from the significant internal differences between SRAM and DRAM and from the standard low-cost package which DRAMs use. The standard DRAM package uses fewer pins to reduce cost, but this leads to increased access time. However, this emphasis on low cost and density is the focus of DRAM technology development, as opposed to access time. For example, since 1980 DRAM access times have decreased by more than a factor of three, while density has increased by a factor of more than 250.

Furthermore, microprocessor cycle times, which in 1980 were similar to DRAM access times, have improved by a factor of more than 100. The situation is somewhat better for SRAM technology, where access time is also emphasized. This large and growing gap between the rate at which the basic memory technology can cycle and the rate at which modern processors cycle could lead to fundamental difficulties in building faster machines. However, computer designers have used the concept of memory hierarchies to overcome this limitation.

2.6 High-Performance Microprocessors

In engineering high-performance microprocessors, designers take advantage of the improvements in technology both to increase the clock rate at which the processor operates, as well as to increase the amount of work done per clock cycle. The amount of work done per clock cycle is typically measured by the CPI (Cycles Per Instruction). Together the CPI and the clock rate determine the instruction execution rate:

$$\text{Instruction execution rate} = \frac{\text{Clock rate}}{\text{CPI}}$$

Since instruction execution rate determines performance, we can maximize performance by increasing the clock rate and decreasing the CPI. The clock rate and the CPI can trade off against one another: by doing more work in a clock cycle, we can lower the CPI (since the number of clock cycles required will be less), but we may end up decreasing the clock rate by an equal factor, yielding no performance benefits. Thus, the challenge is to increase clock rate without increasing CPI and to decrease the CPI without decreasing the clock rate.

Clock rate increases have been achieved both through the use of better technology that offers shorter switching delays and through better computer architectures that reduce the number of switching elements that must be traversed in a clock cycle. Figure 2-2 on page 16 shows the improvement in clock rate over time for microprocessors. The factor of 50 improvement in eight years derives from a combination of device speed, which has contributed about 40 percent of the improvement, and improved architecture, which has contributed roughly 60 percent of the improvement.

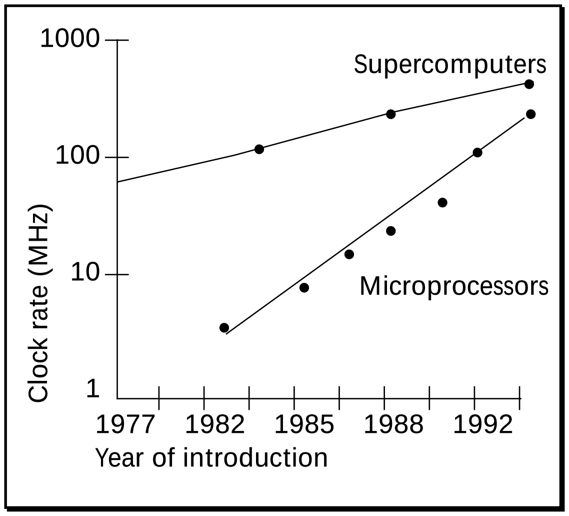


Figure 2-2 Clock Rate Improvement for Cray Supercomputers and High-performance Multiprocessors

This factor-of-50 improvement compares to a factor-of-3 improvement in traditional supercomputer clock rates in the same time frame. The result of this, shown in Figure 2-2, is that microprocessor clock rates are nearly equal to those of supercomputers.

As mentioned earlier, the challenge is to increase clock rate and to decrease CPI. Designers make use of the reduced feature size to build smaller and faster processors. Simultaneously, designers take advantage of the even larger increase in device count to build machines that accomplish more per clock cycle. For scientific computation a good way to measure how much work is done in a clock period is to measure the number of FLoating-point OPerations (FLOPS) per clock. As the number of FLOPS per clock increases, the CPI will fall and performance will increase.

How do designers increase the number of FLOPS per clock? They implement architectural techniques that take advantage of parallelism among the instructions. There are two basic ways in which this can be done:

- *Pipelining*

  Instructions are overlapped in execution and a new operation is started every clock cycle, even though it takes several clock cycles for one operation to complete. Pipelining is the computer architect’s version of an assembly line.

- *Multiple issue*

  Several instructions or operations are started on the same clock cycle. The instructions can be executed in parallel.

In either case, the overlapping instructions must be independent of one another, otherwise, they cannot be executed correctly at the same time. In most machines, this independence property is checked by the hardware, though many compilers help out by trying to create independent instruction sequences that can be overlapped by the hardware.

Many machines combine these techniques. For example, vector machines allow a single vector instruction to specify a number of floating-point operations that are executed in pipelined fashion. When vector instructions are executed in parallel, multiple floating-point operations can be completed in a single clock cycle.

Microprocessors have used pipelining for about ten years, though the earlier machines used little pipelining in the floating-point units, since transistor counts did not allow it. Pipelined floating-point units became the norm in microprocessors about five years ago. In the last few years, several microprocessors have implemented a capability for issuing more than one instruction per clock. The result of this evolution has been a steady increase in the number of floating point operations per clock cycle.

There are two different aspects of this increase in instruction throughput per clock to consider: the peak floating point execution rate and the rate for a benchmark problem. Figure 2-3 shows the peak rate for floating-point operations; this data reflects both the clock rate and the peak floating point issue capability.

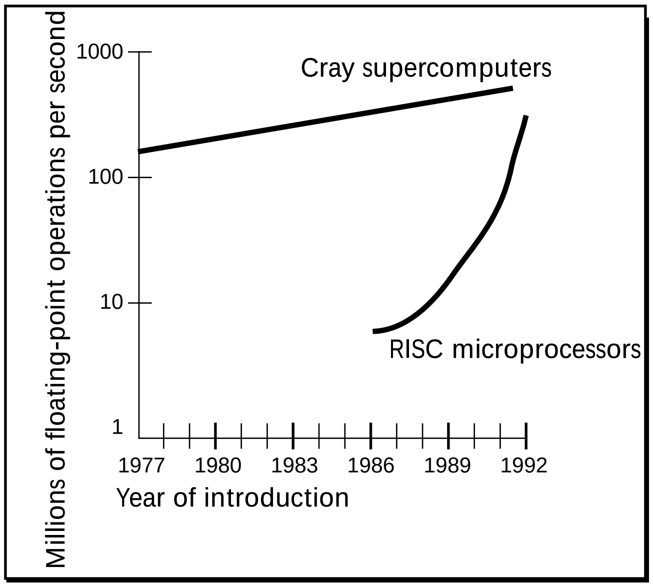


Figure 2-3 Peak Floating-Point Execution Rates Measured with the 1000 x 1000 Linpack

Figure 2-4 shows the rate for a 1000 x 1000 Linpack problem, a common benchmark that involves doing a LU decomposition.

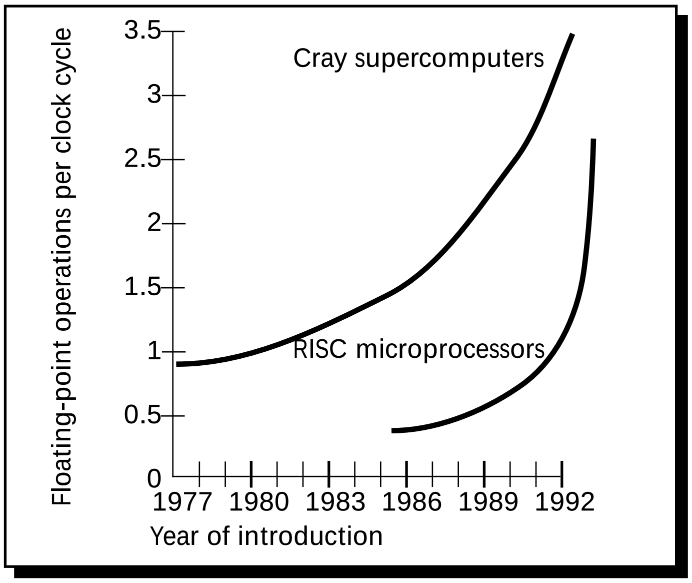


Figure 2-4 Floating-Point Operations per Clock, as Measured Using 1000 x 1000 Linpack

The dramatic improvements in integrated circuit technology have enabled microprocessors to close the performance gap with conventional supercomputers. The improvement in device speed and integration allows the entire CPU core to be placed on a single chip, yielding significant advantages in the speed of interconnections and allowing microprocessor clock speeds to approach, and in the near future surpass, the clock speeds of high-end vector supercomputers.

Furthermore, rapid improvements in density have allowed microprocessor architects to incorporate many of the techniques used in large machines for increasing the throughput per clock. The combined result is that microprocessor-based machines have rapidly closed the performance gap with supercomputers. Because of the enormous price advantage that microprocessors have over conventional supercomputers, microprocessors are becoming the computing engines of choice for all levels of computing.

2.7 Meeting Memory Demands

To maintain a high instruction-throughput rate, memory requests must be satisfied at a high rate. These memory requests consist of accesses for both instructions and data. In scientific and engineering applications, the access patterns for instructions and data are fairly specialized, which allows us to optimize the memory system accordingly. For example, for many scientific applications the program is much smaller than the data set. The program also exhibits high locality, that is, only a small portion of the program is heavily used during any interval. This *temporal locality* arises because the program often consists of nested loops that execute the same instructions many times. Temporal locality can be exploited by keeping recently accessed instructions in a place where they can be fetched quickly.

Data accesses often contain temporal locality, even with large data sets. Another form of locality in data accesses is *spatial locality*—the tendency to use data elements that are close together in memory at the same time. For example, in accessing a matrix, many programs access all the elements in a row or column in sequence. Another example is points in a mesh that are accessed in some sequential order. One can take advantage of spatial locality by retrieving memory words that are close to a word that is requested in parallel with the requested word, with the hope that the processor will need the nearby words soon.

Given locality in memory accesses and speed-versus-cost trade-offs between SRAMs and DRAMs, how can a memory system be organized that meets the demand of a high-performance CPU and also takes advantage of low-cost DRAM technology? The answer lies in using a *hierarchy* of memories with the memories closest to the CPU being composed of small, fast, but more expensive memories.

A memory hierarchy is managed so that the most recently used data is kept in the memories closer to the CPU. These memories, which hold copies of data in the levels further away from the CPU, are called *caches*. The lowest level of the memory hierarchy is built using the lowest cost (and lowest performance) memory technology, namely DRAMs. Figure 2-5 shows the typical structure in a memory hierarchy and some typical sizes and access times.

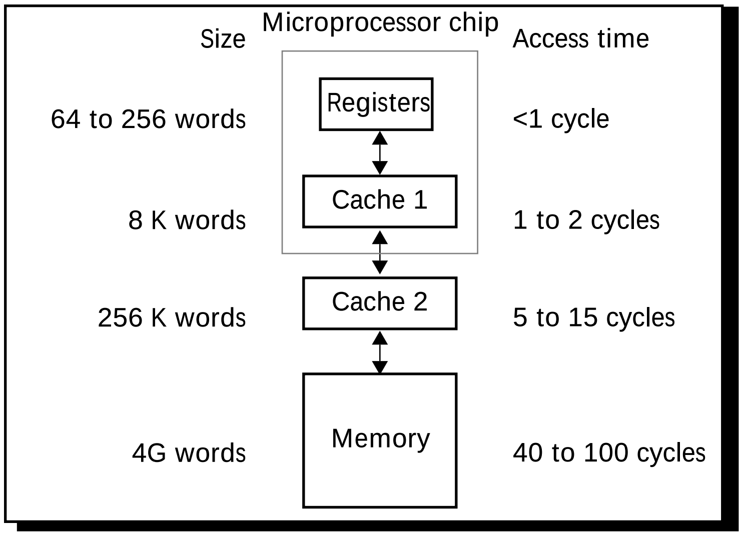


Figure 2-5 Typical Memory Hierarchy

The goal of this organization is to allow most memory accesses to be satisfied from the fastest memory, while still allowing most of the memory to be built from the lowest cost technology. Temporal locality is exploited in such a structure, since newly accessed data items are kept in the top levels of the hierarchy. Spatial locality is exploited by moving blocks consisting of multiple words from a lower level to an upper level, when a request cannot be satisfied in the upper level. Today, machines from low-end personal computers to high-speed multiprocessors use caches as the most cost-effective method to meet the memory demands of fast CPUs.

Vector supercomputers also use a memory hierarchy, but most such machines do not contain caches. Instead, most vector supercomputers contain a small set of vector registers and provide instructions to move data from the main memory to the vector registers in a high bandwidth bulk transfer. The goal in such a design is to be able to move an arbitrary vector from memory to the CPU as fast as possible. Moving an entire vector takes advantage of spatial locality. Once the vector is loaded into the CPU, the compiler tries to keep it in a vector register to take advantage of temporal locality. Since there are not many vector registers (typically, eight to 64) only a small number of vectors can be kept.

What are the important differences between these two approaches? One major advantage of a memory hierarchy with caches is that it can provide much lower average access time, since it can use the fastest parts for the upper levels of the cache and that memory is small, yielding a faster access. To reduce the long access time in a vector supercomputer, many such machines construct the main memory from SRAM. This reduces the time to access this large memory and makes it easier to provide high bandwidth. This approach, however, is much more expensive, since SRAM is six to ten times as expensive per bit. For example, using the hierarchy and main memory size shown in Figure 2-5 and assuming that SRAM is ten times as expensive per bit as DRAM, the memory hierarchy system is about one-tenth the cost of a system that uses only main memory implemented in SRAMs.

The major disadvantage of a cache memory hierarchy is that it typically provides lower bandwidth to the main memory. The result is that programs that do not exhibit good temporal or spatial locality are penalized. Although this has been a major drawback, recent progress in algorithms and compiler technology has led to the development of methods for improving the locality of access in programs. Although these techniques do not apply to all problems, many of the most important scientific problems are amenable to such techniques.

One technique, called *blocking*, can significantly reduce the number of requests to main memory, making caches work extremely well for algorithms with blockable data access. Another technique that is becoming important for capturing temporal locality more efficiently is called is called *prefetching*, in which compilers predict the need of data and cause it to be staged up the memory hierarchy at the right time.

Finally, caches that are typically larger and more general purpose than vector registers makes cache-based memory hierarchies more efficient across a larger range of computing problems.

2.8 Shared Memory Programming

As microprocessors gain dominance in science and engineering computations, with less than one tenth the cost of conventional supercomputers, you may question whether multiple microprocessors can be used in parallel for single problems to further increase speed or make possible new and more ambitious applications. Research has demonstrated this potential in most of the application areas of science and engineering.

This section examines one way in which multiple microprocessors are used in parallel for single problems and for multiple problems: *shared-memory multiprocessing*. Shared memory in this context refers to the singularity of the machine address space. That is, memory in the machine is logically shared by all processes executing on any or all of processors that make up the multiprocessing system. To the programmer, this situation implies implicit communication between multiple processes using loads and stores (machine instructions).

The opposite concept, distributed memory, makes use of multiple private address spaces. Irrespective of the physical organization of memory, this model implies that processors in a multiprocessing machine have logically private memories. To the programmer, this situation implies explicit communication between multiple processes using sends and receives (higher-level constructs usually involving a message-passing interface specification). To understand these two programming models, consider the way machines are programmed and used.

2.8.1 Throughput Processing

Many applications, particularly commercial applications and most general-purpose technical applications, are inherently serial, or are very difficult to automatically parallelize. A good example of multiprocessing is the typical development environment workload, consisting of compilation, document editing, simulations, mail processing, database queries, printing, spreadsheet processing, and so on. These tasks are automatically time-sliced onto the available CPUs in the system. For this kind of workload, POWER CHALLENGE systems deliver very high throughput, since there are many high-performance CPUs executing these tasks simultaneously, as in Figure 2-6.

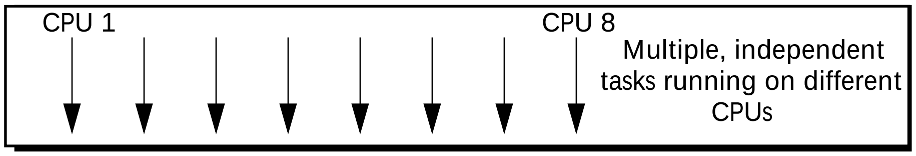


Figure 2-6 Throughput Mode of Computations

This capability is referred to as *N on N*, meaning that *N* different tasks may be executing on *N* different CPUs in parallel, each task performing unique, serial work.

With the extraordinary throughput capabilities of POWER CHALLENGE architecture, it may be better to think of this capability as *M on N*, where *M* is the number of tasks to run and *N* is the number of processors. In a heavily loaded situation, $M > N$, and IRIX shows its powerful abilities to properly balance the execution of these many tasks while providing excellent response times and high system throughput.

2.8.2 Parallel Processing

Many applications exhibit parallelism that can be exposed by compilers such as the MIPSpro Power C and MIPSpro Power FORTRAN 77 compilers. In general, a normal application can be parallelized into an alternating sequence of serial and parallel sections. This is because there are actions the application takes which are inherently serial (opening files, initializing data areas, etc.) and actions which are potentially parallel (loops, subroutine calls, and more).

Conceptually, upon entering a parallel segment, the application splits into a number of threads which execute portions of the parallel segment at once. At the end of the parallel segment there exists a *barrier* where the threads are collected until all are finished executing, at which time the execution of the next serial segment begins. This kind of parallelism is referred to as *1 on N*, meaning that one application is running on *N* CPUs. Figure 2-7 illustrates this situation.

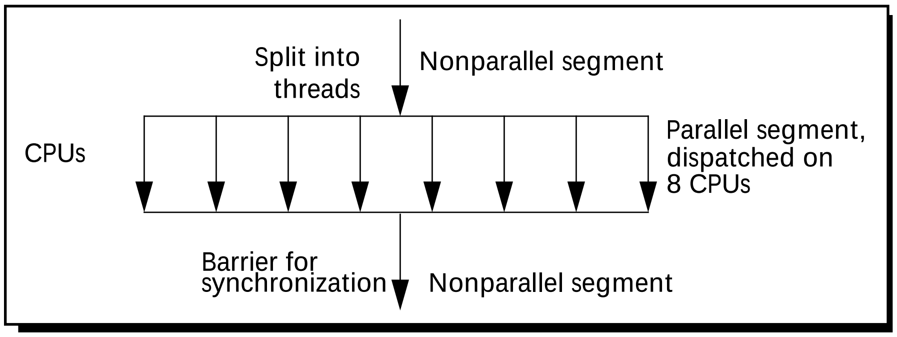

Split into threads
Nonparallel segment
CPUs
Parallel segment, dispatched on 8 CPUs
Barrier for synchronization
Nonparallel segment

Figure 2-7 Parallel Mode of Computation

For most parallel applications, there is a limit to the performance that can be achieved through parallelism because of various factors, including the overhead of communication between the threads of the application and the parallelism support provided by the underlying hardware. Most of the parallel speedup achievable occurs with between four and eight processors (see Figure 2-8 on page 23), although this number varies, depending on the application and the manner in which it is written.

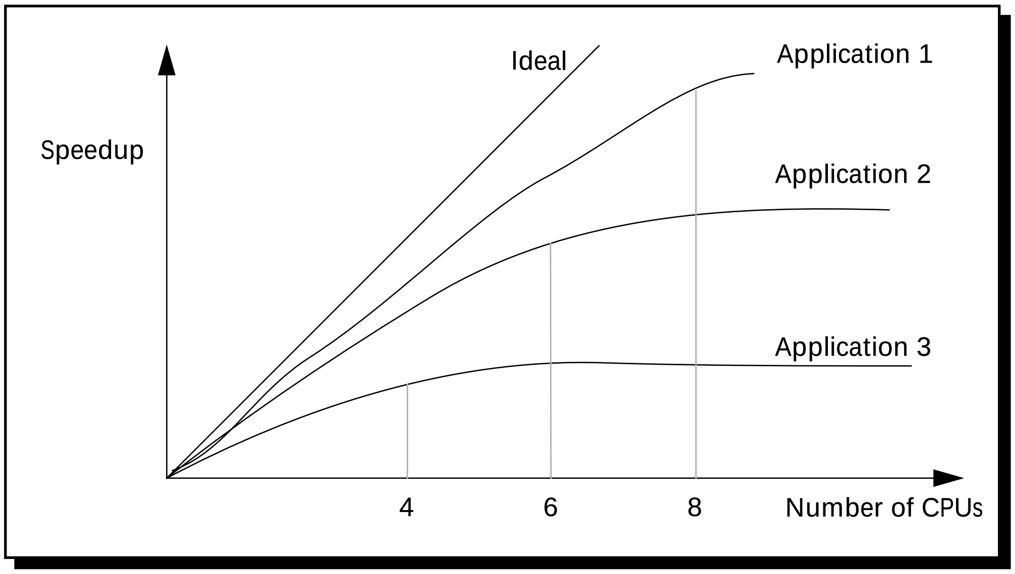


Figure 2-8 Typical Parallel Application Speedup

The IRIX operating system includes specific features for managing these kinds of threaded applications, including thread manipulation, shared-memory and high-performance user-level locks and semaphores. It is generally acknowledged that a shared-memory programming model provides the highest performance and the most efficient programming environment for most threaded applications.

2.8.3 Parallel Throughput Processing

Workloads that include multiple parallel applications are often characterized by one or more of the following:

- More than one user of the same parallel application or a different parallel application
- Parallel applications that typically achieve the best efficiency with a small number of parallel threads, usually four to eight, this number being less than the maximum number of CPUs in POWER CHALLENGE multiprocessors
- Loosely coupled message-passing applications, which in most respects resemble a typical multiprocessor workload
- High multiprocessor workload, in addition to a parallel-execution workload, to satisfy the general-purpose and non-parallel application needs of the users

POWER CHALLENGE servers are excellent for these types of workloads. *Parallel throughput* refers to the performance achievable for multiple parallel applications and serial applications running at the same time on a multiprocessor. Figure 2-9 on page 24 illustrates this workload.

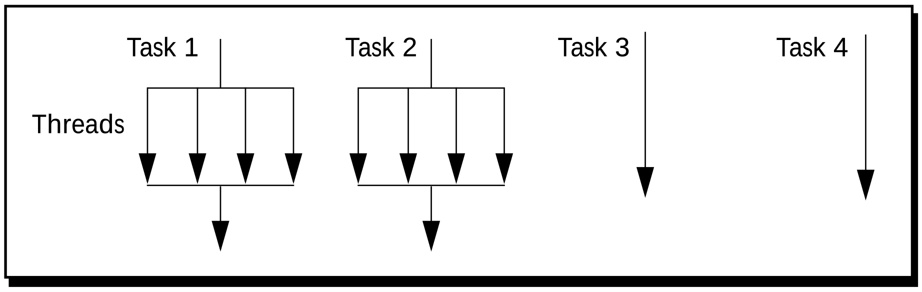


Figure 2-9 Parallel Throughput Mode of Computations

As one might imagine, mixing these different kinds of parallel processing places a number of interesting demands upon the underlying operating system. The operating system must be able to quickly start a number of threads for an application when entering a parallel segment, and then just as quickly shut them down when leaving it, while normal computing applications and other parallel applications execute at the same time. Balancing the demands of competing applications to achieve the greatest possible system throughput, the highest performance for parallel applications, and the fastest possible response time, is truly a grand challenge for operating system designers.

In summary, the shared-memory, single-address-space programming model is not only the most intuitive way to program computers, but a very efficient one as well. This efficiency depends on the amount of data sharing that goes on in parallel processing. Workstation clusters, which are an extreme example of a multiple-address-space programming model, scale well for throughput processing or for embarrassingly parallel codes, but are unsuitable for codes which have a poor computation-to-communication ratio. In such clusters, the communication to synchronize between multiple threads of the same process occurs on a network link such as Ethernet or FDDI. These links provide much less bandwidth for synchronization than either busses or specially designed high-bandwidth interconnects.

Providing a shared-memory, single-address-space programming model does not come for free. The hardware designer faces the challenge of providing hardware cache coherence, whether the memory is physically shared or replicated closer to each individual processor.

2.9 Designing for Economic Scalability

The recent trend in the industry toward massively parallel processing has focused attention on the problem of achieving scalability to thousands of processors. Many interconnection topologies have arisen out of efforts to maximize the interconnection bisection bandwidth and to minimize the latency of access to remote private memories. The goal of achieving scalability to a large number of processors, by necessity, requires a design infrastructure in terms of the connectivity that prohibits economical scaling of such machines into the deployable, affordable range.

At Silicon Graphics, the emphasis is on designing multiprocessors with a moderate range of parallelism, while scaling down to very affordable low-end multiprocessors in the same family, based on the same technology.

Ultra-high-performance may be achieved with multiple shared-memory multiprocessor systems connected by a very high bandwidth interconnect, such as HiPPI, in an optimized topology. POWER CHALLENGEarray is an interconnection of high-performance, shared-memory multiprocessors with demonstrable performance on grand challenge problems. The POWER CHALLENGEarray approach, while involving no more work than the traditional distributed-memory, private-address-space multiprocessors, has the great advantage of a high computation-to-communication ratio, which means that the amount of message-passing is low compared to the amount of work done for each message sent.

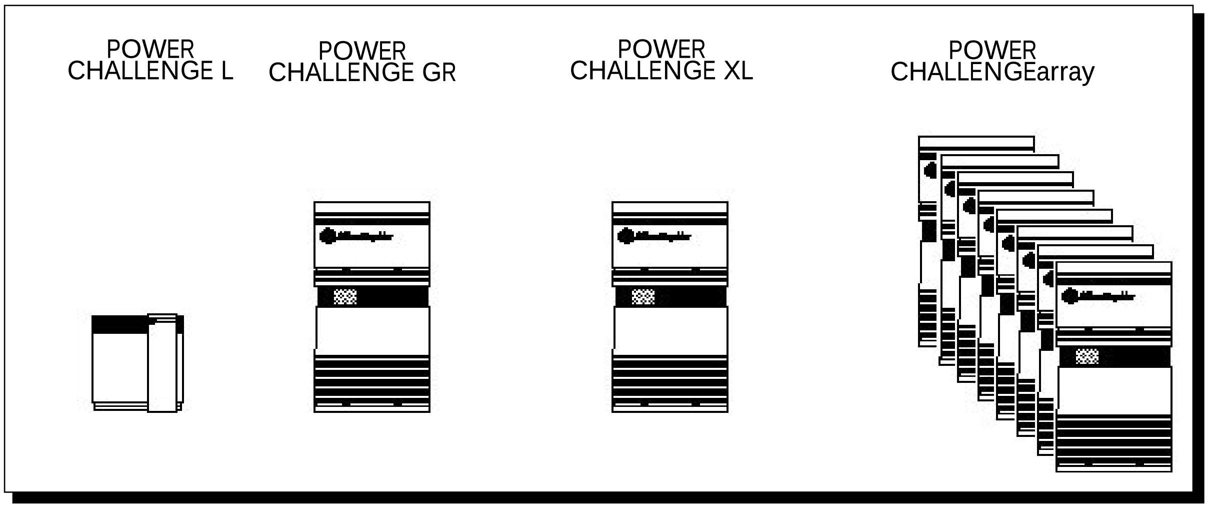


Figure 2-10 Silicon Graphics Scalable Supercomputer Family

No matter how affordable or deployable a supercomputer is, it is still highly desirable to use the large multiprocessor machine for production runs, while program development and debugging is done on decentralized computers, preferably desktops. Silicon Graphics provides this environment with desktop platforms based on the same MIPS RISC microprocessor. From a choice of graphics options and a complete suite of development and debugging tools on the low-cost desktop workstations to a scalable family of powerful super-computers, the Silicon Graphics family of products provides a complete develop-ment and deployment environment for high-performance computing needs.

Chapter 3 POWER CHALLENGE Hardware Architecture

POWER CHALLENGE supercomputers are shared-memory multiprocessor computers designed to give engineers and scientists powerful computational, design, and visualization tools at aggressive price/performance levels. Balance and scalability across processor, memory, and I/O subsystems provide super-computer performance at a price associated with commodity microprocessors.

For high-performance and flexible configuration to meet the needs of wide- ranging application environments, the POWER CHALLENGE series features:

- *Parallel processing* for running a single application on multiple processors simultaneously to improve time to solution
- *Multiprocessing throughput* for running many applications simultaneously on different CPUs for increased system throughput
- *File and network services* for accessing and manipulating data residing in large secondary storage systems or on other network-interconnected computers

The POWER CHALLENGE system consists of at least one of each of the following: quad R10000™ CPU boards, interleaved memory boards, and POWERchannel-2™ I/O boards, plus a wide variety of I/O controllers and peripheral devices. These boards are interconnected by the POWERpath-2 bus, which provides high-bandwidth, low-latency, cache-coherent communication between processors, memory, and I/O.

To provide a cost-effective solution for every need, POWER CHALLENGE is available in either a deskside configuration, supporting up to 12 processors, or a rack configuration, supporting up to 36 processors and very large memory and I/O subsystems. Table 1: summarizes maximum system configurations:

Table 1: POWER CHALLENGE XL Maximum System Configurations[a]

Component	Description
Processor	36 MIPS R10000 streaming superscalar RISC
Main memory	16 GB, 8-way interleaving
I/O bus	6 POWERchannel-2, each providing 320 MB per second I/O bandwidth
SCSI channels	42 fast and wide independent SCSI-2 channels
Disk	17.4-TB disk (RAID) or 5.6-TB disk (non-RAID)
Connectivity	4 HiPPI channels, 20 Ethernet channels, 6FDDI channels, 8 ATM channels
VME slots	5 VME64 expansion buses provide 25 VME64 slots

a. For expansion beyond XL cabinet restraints, see the *POWER CHALLENGEarray Technical Report*

This chapter describes:

- R10000 processor architecture
- R8000 processor architecture
- POWERpath-2 Coherent interconnect
- Synchronization
- Memory subsystem
- I/O subsystem
- HIO bus modules
- VME bus
- Peripherals

3.10 R10000™ Processor Architecture

The R10000 microprocessor from MIPS Technologies is a four-way superscalar architecture that fetches and decodes four instructions per cycle. Each decoded instruction is appended to one of three instruction queues, and each queue can perform dynamic scheduling of instructions. The queues determine the execution order based on the availability of the required execution units. Instructions are initially fetched and decoded in order, but can be executed and completed out-of-order, allowing the processor to have up to 32 instructions in various stages of execution. The impressive integer and floating-point performance of the R10000 make it ideal for applications such as scientific computing, engineering workstations, 3-D graphics workstations, database servers, and multi-user systems. The high throughput is achieved through the use of wide, dedicated data paths and large on-and-off chip caches.

The R10000 microprocessor implements the MIPS IV instruction set architecture. MIPS IV is a superset of the MIPS III instruction set architecture and is backward compatible. The R10000 microprocessor delivers peak performance of 800 MIPS (400 MFLOPS) with a peak data transfer rate of 3.2GB per second to secondary cache. The R10000 microprocessor is available in a 599 CLGA package and is fabricated using a CMOS sub-.35-micron silicon technology.

Key features of the R10000 include:

- ANDES Advanced Superscalar Architecture
  - Supports four instructions per cycle
  - Two integer and two floating-point execute instructions plus one load/store per cycle
- High-performance design
  - 3.3 volt technology
  - Out-of-order instruction execution
  - 128-bit dedicated secondary cache data bus
  - On-chip integer, FP, and address queues
  - Five separate execution units

  - MIPS IV instruction set architecture
- High Integration Chip-Set
  - 32 KB 2-way set associative, 2-way interleaved data cache with LRU replacement algorithm
  - 32 KB 2-way set associative instruction cache
  - 64 entry translation lookaside buffer
  - Dedicated second level cache support
- Second Level Cache Support
  - Dedicated 128-bit Data Bus
  - Generation of all necessary SSRAM signals
  - 3.2 GB per second peak data transfer rate
  - Programmable clock rate to SSRAM
- Compatible with Industry Standards
  - ANSI/IEEE Standard 754-1985 for binary floating-point arithmetic
  - MIPS III instruction set compatible
  - Conforms to MESI cache consistency protocol
  - IEEE Standard 1149.1/D6 boundary scan architecture
- Avalanche Bus System Interface
  - Direct connect to SSRAM
  - Split transaction support
  - Programmable interface

3.10.1 Modern Computing Challenges

The current generation of today's microprocessor architectures outperform their earlier counterparts by orders of magnitude. Such radical increases in performance, speed, and transistor count from generation to generation, often separated by only a few years, can seem overwhelming to the casual observer.

Although current microprocessor designs vary greatly, there are many commonalities between them. Each one performs address generation, each contains arithmetic logic units, register files, and a system interface. Most have on-chip caches, a translation lookaside buffer (TLB), and almost all current architectures have on-chip floating-point units.

Different design techniques perform these basic functions, but the nature and existence of these functions and the need to perform them lend themselves to inherent problems that must be overcome. This section discusses some of the common computing challenges faced by all microprocessor designers. section 3.10.2 on page 32 discusses some of the techniques used to overcome these challenges. section 3.10.3 on page 35 discusses how the MIPS R10000 microprocessor implements the techniques discussed in Section 3.10.2.

3.10.1.1 Memory and Secondary Cache Latencies

Early microprocessors had to fetch instructions directly from memory. Histor-ically, memory access times have lagged far behind the data-rate requirements of the processor. After issuing a request for data the processor had to wait long periods of time for the data to return. This severely hampered the processor's ability to operate efficiently at the speeds for which it was designed.

Implementation of off-chip secondary cache memory systems has helped to fix this problem. A cache memory system comprising a small amount of memory, normally 32–256 KB, contains a block of memory addresses comprising a small portion of main memory. Cache memory has much faster access times and can deliver data to the processor at a much higher rate than main memory.

On-chip cache memory systems can greatly improve processor performance because they often allow access to be completed in one cycle. Performance improvements of on-chip cache systems have caused many architectures to dedicate increasing amounts of space and logic to cache design. Many cache system designs requires up to half of the total die. Performance is highest when the application can run within the cache. However, when the application is too large for the cache, performance decreases significantly. Figure 3-1 on page 30 shows the relationship between application performance and size.

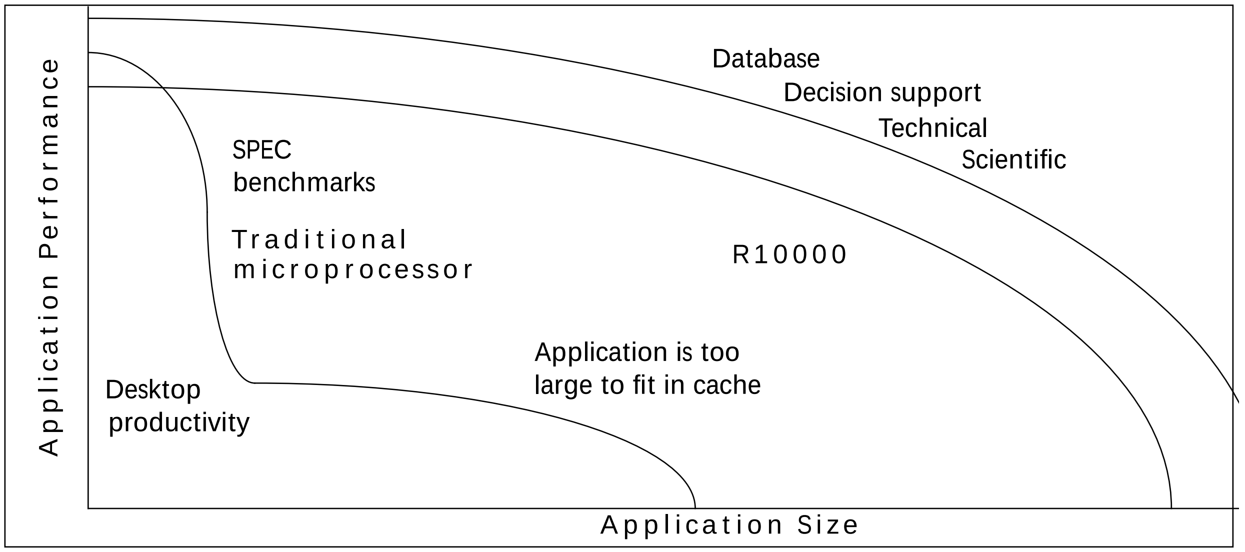


Figure 3-1 Application Performance versus Size

The on-chip cache contains a range of addresses which comprise a subset of those addresses in the secondary cache. In turn, the secondary cache contains a range of addresses which comprise a subset of those addresses in main memory. Figure 3-2 shows the relationship between caches in a typical computer system.

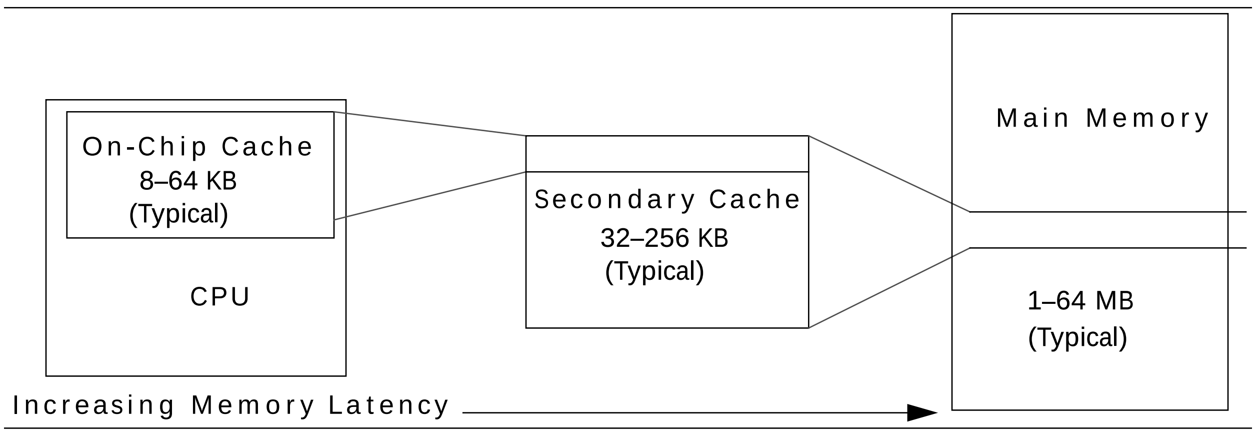


Figure 3-2 Memory Relationships in a Typical Computer System

As beneficial as on-chip cache systems are to processor performance, current semiconductor technology and available transistor counts limit cache size. Currently 64 KB (32KB Data, 32KB Instruction) is a large on-chip cache requiring several million transistors to implement.

Limiting size factors regarding on-chip caches place increasing importance on secondary cache systems, where cache size is only limited by the market into which the product is being sold. However, cache memory has its limitations.

The access times of most currently available RAM devices are long relative to processor cycle times and force the memory system designer to find ways to hide them. Interleaving the cache system is one way to accomplish this. Interleaved cache memory systems allow processor memory requests to be overlapped. Both cache and main memory can be interleaved. Two-way and four-way interleaving is common in memory systems. Increasing the amount of interleaving allows the ability to hide more of the access and recovery times of each bank, but increased complexity is required to support them. See “Interleaved Memory” on page 34. for a further discussion of memory interleaving.

3.10.1.2 Data Dependencies

In a computer program, instructions are fetched from the instruction cache, decoded, and executed. The corresponding data is often fetched from a register, manipulated within an ALU, and the result placed either in the same register or perhaps in another register.

Data dependency occurs when the next instruction in the sequence requires the result of the previous instruction before it can execute. This can impact the performance of instructions requiring many cycles, as execution of the second instruction waits until the first instruction is done and the result written to the register. Some dependencies can be avoided by rearranging the program so that the result of a given instruction is not used by the next few instructions.

Out-of-order execution using register-renaming helps alleviate data dependency problems. Register renaming is explained in section 3.10.2.1 on page 32.

3.10.1.3 Branches

All computer programs contain branches. Some branches are unconditional, meaning that the program flow is interrupted as soon as the branch instruction is executed. Other branches are conditional, meaning that the branch is taken only if certain conditions are met. Program flow interruption is inherent to all computer software and the microprocessor hardware has little choice but to deal with branches in the most efficient way possible.

When a branch is taken, the new address where the program is to resume may or may not reside in the secondary cache. The latency is increased depending on where the new instruction block is located. Since the access times of the main memory and secondary cache are far greater than the on-chip cache, as shown in Figure 3-2, branching can often degrade processor performance.

The branching problem is further compounded in superscalar machines where multiple instructions are fetched every cycle and progress through stages of a pipeline toward execution. At any given time, depending on the size of the pipeline, numerous instructions can be in various stages of execution. When a conditional branch instruction is executed, it is not known until many cycles later, when the instruction is actually executed, whether or not the branch should have been taken.

Implementation of branching is an important architectural problem. To improve performance, many current architectures incorporate branch prediction circuitry, which can be implemented in a number of ways. section 3.10.2.2 on page 33 discusses some commonly used branch prediction techniques.

3.10.2 Tolerating Memory Latency

Memory latency reduction is a critical issue in increasing processor performance. This section discusses some of the common architectural techniques used to reduce memory latency.

3.10.2.1 High-Bandwidth Secondary Cache Interface

In an ideal secondary cache interface, the cache receives a data request from the processor and can always return data in the following clock. This is referred to as a true zero wait-state cache. To design a secondary cache that can approach zero wait state performance, the processor’s system interface must be designed such that data can be transferred at the maximum rate allowed.

The address and data busses of most processors interface to the entire computer system. Any number of different devices can be accessed by the processor at any given time.

Whenever an on-chip cache miss occurs, an address is driven out onto the bus and the secondary cache is accessed, transferring the requested data to the on-chip cache.

If an on-chip cache miss occurs in a shared-bus system, and the processor is using the external bus to read or write some other device, the access to secondary cache must wait until the external data and address busses are free. This can take many cycles depending on the peripheral being accessed.

In a dedicated bus system the data, address, and control busses for the secondary cache are separate from those which interface to the rest of the system. These busses allow secondary cache accesses to occur immediately following an on-chip cache miss, despite what else is happening in the system.

Figure 3-3 on page 33 is a block diagram of both a shared and dedicated secondary cache interface. Refer to section 3.10.9.1 on page 44 for more information on the dedicated secondary cache interface of the R10000 microprocessor.

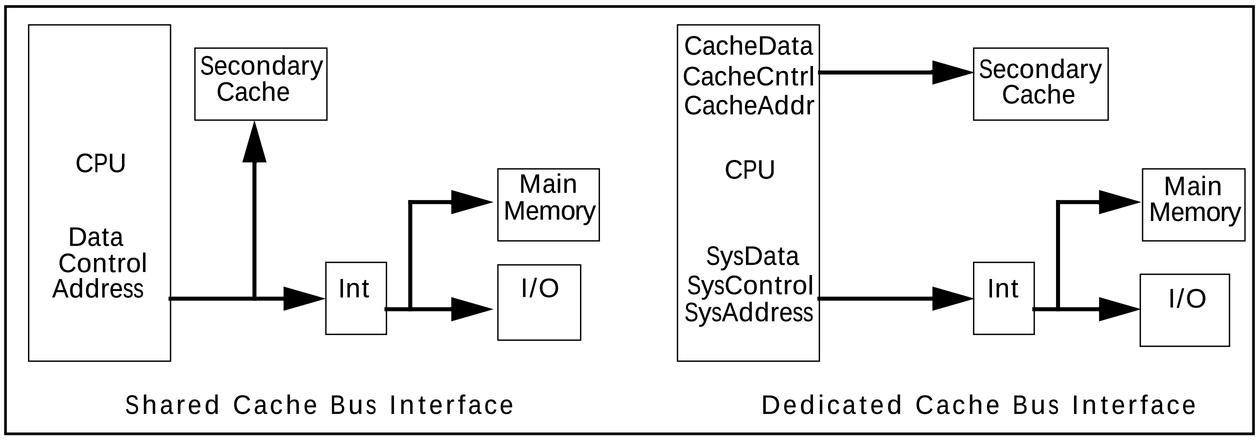


Figure 3-3 Dedicated Secondary Cache Bus Interface

3.10.2.2 Block Accesses

When an on-chip cache miss occurs there is normally a programmable number of bytes which are transferred each time the secondary cache is accessed. This number is referred to as the cache line size. A common line size for many current architectures is 32 bytes.

The number of accesses required to perform a line fill depends on the size of the external data bus of the processor. For example, a processor with a 64-bit data bus interfacing to a 64-bit wide memory performing a 32-byte (256 bits) cache line fill would require four secondary cache accesses to fetch all of the data. To accomplish this the processor must generate four separate addresses and drive each one out onto the external address bus, along with the appropriate control signals.

Block access mode allows the processor to generate only the beginning address of the sequence. The remaining three addresses are generated either by cache control logic, or within the cache RAM itself. The R10000 microprocessor system interface supports block accesses.

3.10.2.3 Interleaved Memory

Interleaving is a technique used to increase memory bandwidth. The concept of interleaving can be applied both to secondary cache and main memory.

Simple memory systems have one bank of memory. If the memory is accessed, some amount of time must pass before it can be accessed again. This time depends both on the system design as well as the speed of the memory devices being used. Having multiple banks allows bank accesses to be overlapped. The ability to overlap bank accesses helps to hide these inherent memory latencies and becomes increasingly important as the amount of data requested increases.

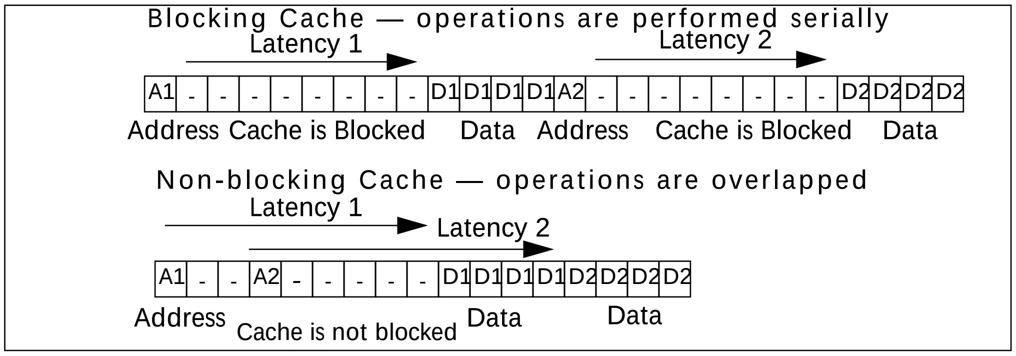
A typical interleaved memory system has even and odd banks. For example, the processor places a request for data at an even address. The memory controller then initiates a cycle to the even bank. Once the address has been latched by the memory control logic, the processor can generate a new address, often in the next clock. If the new address is to the odd bank of memory, memory access can begin immediately as the odd bank is currently idle. By the time the access time to the even bank has elapsed and the data has been returned, the odd bank is also ready to return data. Zero-wait-state performance is achievable as long as sequential accesses to the same bank are a minimal.

Two-way and four-way interleaved memory systems are the most common. The number of banks and the data bandwidth of each is often determined by the processor. For example, if the cache line size of the processor is 32 bytes, this means that each time a memory access is initiated 32 bytes must be returned to the processor. Since 32 bytes = 256 bits, a common approach is to have four banks of 64 bits each. This scheme would require a processor with a 64-bit data bus in order to alleviate any external multiplexing of data. Each bank is accessed in an order determined by the processor. section 3.10.5.1 on page 38 discusses the interleaving characteristics of the R10000 microprocessor.

3.10.2.4 Non-Blocking Cache

In a typical implementation, the processor executes out of the cache until a cache miss is taken. A number of cycles elapse before data is returned to the processor and placed in the on-chip cache, allowing execution to resume. This type of implementation is referred to as a blocking cache because the cache cannot be accessed again until the cache miss is resolved.

Non-blocking caches allow subsequent cache accesses to continue even though a cache miss has occurred. Locating cache misses as early as possible and performing the required steps to solve them is crucial in increasing overall cache system performance. Figure 3-4 shows an example of how a blocking and non-blocking cache would react to multiple cache misses.



Blocking Cache — operations are performed serially
Latency 1
Latency 2
A1 - - - - - - - - D1 D1 D1 D1 A2 - - - - - - - - D2 D2 D2 D2
Address
Cache is Blocked
Data
Address
Cache is Blocked
Data
Non-blocking Cache — operations are overlapped
Latency 1
Latency 2
A1 - - A2 - - - - - D1 D1 D1 D1 D2 D2 D2 D2
Address
Cache is not blocked
Data
Data

Figure 3-4 Multiple Misses in a Blocking and Non-Blocking Cache

The major advantage of a non-blocking cache is the ability to stack memory references by queuing up multiple cache misses and servicing them simultaneously. The sooner the hardware can begin servicing the cache miss, the sooner data can be returned.

3.10.2.5 Prefetch

Prefetching of instructions is a technique whereby the processor can request a cache block prior to the time it is actually needed. The prefetch instruction must be integrated as part of the instruction set and the appropriate hardware must exist to execute the prefetch instruction.

For example, assume the compiler is progressing sequentially through a segment of code. The compiler can make the assumption that this sequence will continue beyond the range of addresses available in the on-chip cache and issue a prefetch instruction which fetches the next block of instructions in the sequence and places them in the secondary cache. Therefore, when the processor requires the next sequence, the block of instructions exist in the secondary cache or a special instruction buffer as opposed to main memory and can be fetched by the processor at a much faster rate. If for some reason the block of instructions is not needed, the area in the secondary cache or the buffer is simply overwritten with other instructions.

Prefetching allows the compiler to anticipate the need for a given block and place it as close to the CPU as possible.

3.10.3 Data Dependency

Two common techniques are used to reduce the negative performance impact of data dependencies. Each of these is discussed below.

3.10.3.1 Register Renaming

Register renaming distinguishes between logical registers, which are referenced within instruction fields, and physical registers, which are located in the hardware register file. Logical registers are dynamically mapped into physical register numbers using mapping tables which are updated after each instruction is decoded. Each new result is written into a new physical register. However, the previous contents of each logical register is saved and can be restored in case its instruction must be aborted following an exception or an incorrect branch prediction.

As the processor executes instructions myriad temporary register results are generated. These temporary values are stored in register files along with permanent values. The temporary values become new permanent values when the corresponding instructions graduates. An instruction graduates when all previous instructions have been successfully completed in program order.

The programmer is aware of only logical registers. The implementation of physical registers is hidden. Logical register numbers are dynamically mapped into physical register numbers. The mapping is implemented using mapping tables which are updated after each instruction is decoded. Each new result is written into a physical register. However, until the corresponding instruction graduates, the value is considered temporary.

Register renaming simplifies data dependency checks. In a machine that can execute instructions out-of-order, logical register numbers can become ambiguous as the same register may be assigned a succession of different values. But because physical register numbers uniquely identify each result, dependency checking becomes unambiguous. section 3.10.7 on page 40 discusses how the R10000 microprocessor implements register renaming.

3.10.3.2 Out-of-Order Execution

In a typical pipelined processor, which executes instructions *in order*, each instruction depends on the previous instruction, which produced its operands. Execution cannot begin until the operands become valid. If operands are invalid, the pipeline stalls until those operands become valid. Because instructions execute *in order*, stalls usually delay all subsequent instructions.

In an *in-order* superscalar machine where multiple instructions are fetched each cycle, several consecutive instructions can begin execution simultaneously if all corresponding operands are valid. However, the processor stalls at any instruction whose operands are not valid.

In *out-of-order* superscalar machine, each instruction can begin execution when its operands become available, despite the original instruction sequence. The hardware effectively rearranges instructions to keep the various execution units busy. This process is called *dynamic issuing*. section 3.10.7.1 on page 41 discusses the out-of-order implementation used in the R10000 microprocessor.

3.10.4 Branch Prediction

Branches interrupt the pipeline flow. Therefore, branch prediction schemes are needed to minimize the number of interruptions. Branches occur frequently, averaging about one out of every six instructions. In superscalar architectures where more than one instruction at a time is fetched, branch prediction becomes increasingly important. For example, in a four-way superscalar architecture, where four instructions per cycle are fetched, a branch instruction can be encountered every other clock.

Most branch prediction schemes use algorithms that keep track of how a conditional branch instruction behaved the last time it was executed. For example, if the branch history circuit shows that the branch was taken the last time the instruction was executed, the assumption could be made that it will be taken again. A hardware implementation of this assumption would mean that the program would vector to the new target address and that all subsequent instruction fetches would occur at the new address. The pipeline now contains a conditional branch instruction fetched from some address, and numerous instructions fetched afterward from some other address. Therefore, all instructions fetched between the time the branch instruction is fetched and the time it is executed are said to be speculative. That is, it is not known at the time they are fetched whether or not they will be completed. If the branch was predicted incorrectly, the instructions in the pipeline must be aborted.

Many architectures implement a *branch stack* which saves alternate addresses. If the branch is predicted to be not-taken, the address of the actual branch instruction is saved. If the branch is predicted to be taken, the address immediately following the branch instruction is saved. section 3.10.5.4 on page 39 discusses the branch mechanism of the R10000 microprocessor.

3.10.5 R10000 Product Overview

The R10000 microprocessor implements many of the techniques mentioned above. This section discusses some of these features. Figure 3-5 shows a block diagram of the R10000 microprocessor.

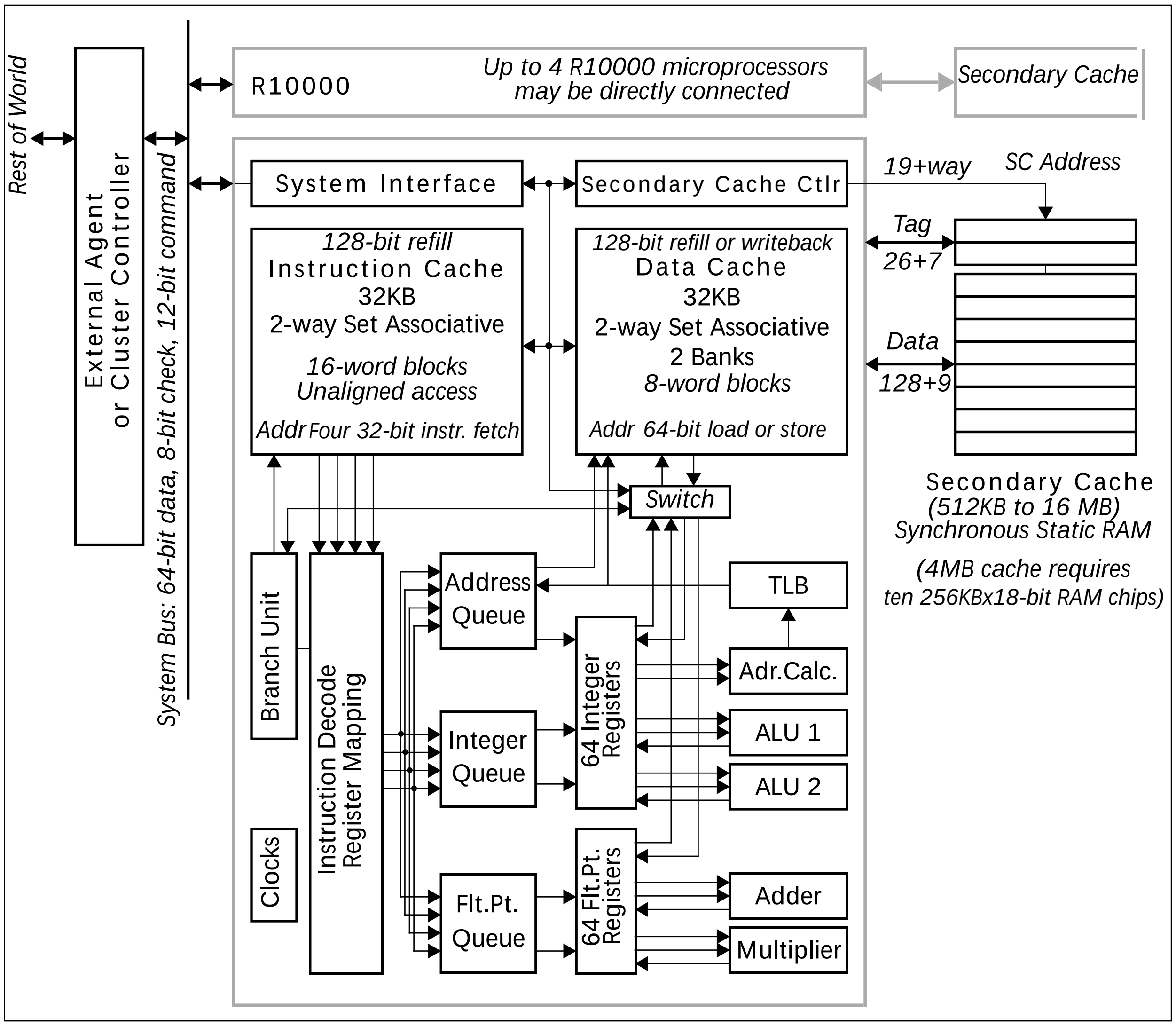


Figure 3-5 R10000 Processor Block Diagram

3.10.5.1 Primary Data Cache

The primary data cache of the R10000 microprocessor is 32KB in size and is arranged as two identical 16KB banks. The cache is *two-way interleaved*. Each of the two banks is *two-way set associative*. Cache line size is 32 bytes.

The data cache is *virtually indexed* and *physically tagged*. The virtual indexing allows the cache to be indexed in the same clock in which the virtual address is generated. However, the cache is physically tagged in order to maintain coherency with the secondary cache.

3.10.5.2 Secondary Data Cache

The secondary cache interface of the R10000 microprocessor provides a 128-bit data bus which can operate at a maximum of 200MHz, yielding a peak data transfer rate of 3.2GB per second. All of the standard Synchronous Static RAM interface signals are generated by the processor. No external interface circuitry is required. The minimum cache size is 512KB. Maximum cache size is 16MB. Secondary cache line size is programmable at either 64 bytes or 128 bytes.

3.10.5.3 Instruction Cache

The instruction cache is 32KB and is two-way set associative. Instructions are partially decoded before being placed in the instruction cache. Four extra bits are appended to each instruction to identify the execution unit to which the instruction will be dispatched. The instruction cache line size is 64 bytes.

3.10.5.4 Branch Prediction

The branch unit of the R10000 microprocessor can decode and execute one branch instruction per cycle. Since each branch is followed by a delay slot, a maximum of two branch instructions can be fetched simultaneously, but only the earlier one will be decoded in a given cycle.

A *branch bit* is appended to each instruction during instruction decode. These bits are used to locate branch instructions in the instruction fetch pipeline.

The path a branch will take is predicted using a *branch history RAM*. This two-bit RAM keeps track of how often each particular branch was taken in the past. The two-bit code is updated whenever a final branch decision is made.

Any instruction fetched after a branch instruction is speculative, meaning that it is not known at the time these instructions are fetched whether or not they will be completed. The R10000 microprocessor allows up to four outstanding branch predictions which can be resolved in any order.

Special on-chip *branch stack* circuitry contains an entry for each branch instruction being speculatively executed. Each entry contains the information needed to restore the processor’s state if the speculative branch is predicted incorrectly. The branch stack allows the processor to restore the pipeline quickly and efficiently when a branch misprediction occurs.

3.10.6 Queueing Structures

The R10000 microprocessor contains three instruction queues. These queues dynamically issue instructions to the various execution units. Each queue uses instruction tags to track instructions in each execution pipeline stage. Each queue performs dynamic scheduling and can determine when the operands that each instruction needs are available. In addition, the queues determine the execution order based on the availability of the corresponding execution units. When the resources become available, the queue releases the instruction to the appropriate execution unit.

3.10.6.1 Integer Queue

The *integer queue* contains 16 entries and issues instructions to the two integer arithmetic logic units. Integer instructions are written into empty queue entries and up to four entries may be written each cycle. Integer Instructions remain in the queue until being issued to an ALU.

3.10.6.2 Floating-Point Queue

The *floating-point queue* contains 16 entries and issues instructions to the floating-point adder and floating-point multiplier execution units. Floating-point instructions are written into empty queue entries and up to four entries may be written each cycle. Instructions remain in the queue until being issued to an execution unit. The floating-point queue also contains multiple-pass sequencing logic for instructions such as the *multiply-add*. This instruction is dispatched first to the multiply unit, then passed directly to the adder unit.

3.10.6.3 Address Queue

The *address queue* issues instructions to the Load-Store unit and contains 16 entries. The queue is organized as a circular FIFO (first-in first-out) buffer. Instructions can be issued in any order, but must be written to or removed from the queue in sequential order. Up to four instructions can be written every cycle. The FIFO maintains the programs original instruction sequence so that memory address dependencies may be computed easily.

An issued instruction may fail to complete because of a memory dependency, a cache miss, or a resource conflict. In these cases the address queue must reissue the instruction until it is completed.

3.10.7 Register Renaming

Dependencies between instructions can degrade the overall performance of the processor. Register renaming is a technique used to determine these dependencies between instructions and provide for precise exception handling. When a register is *renamed* the logical registers which are referenced in an instruction are mapped to physical registers using a mapping table. A logical register is mapped to a new physical register whenever it is the destination of an instruction. Hence when an instruction puts a new value in a logical register, that logical register is renamed to use the new physical register. However, the previous value remains in the old physical register. Saving the old register value allows for precise exception handling.

While each instruction is renamed, its logical register numbers are compared to determine the dependencies between the four instructions being decoded during the same cycle.

3.10.7.1 Mapping Tables

The instruction mapping scheme implemented in the R10000 microprocessor consists of a mapping table, an active list, and a free list. Separate mapping tables and free lists are provided for integer and floating-point instructions. To maintain sequential ordering of instructions, only one active list contains both integer and floating-point instructions.

The R10000 microprocessor contains 64 physical registers. At any given time each physical register value is contained within one of these lists. Figure 3-6 on page 41 shows a block diagram of the integer instruction mapping scheme.

Instructions are fetched from the instruction cache and placed in the mapping table shown in Figure 3-6. At any given time, each of the 64 physical registers is located in one of these three blocks.

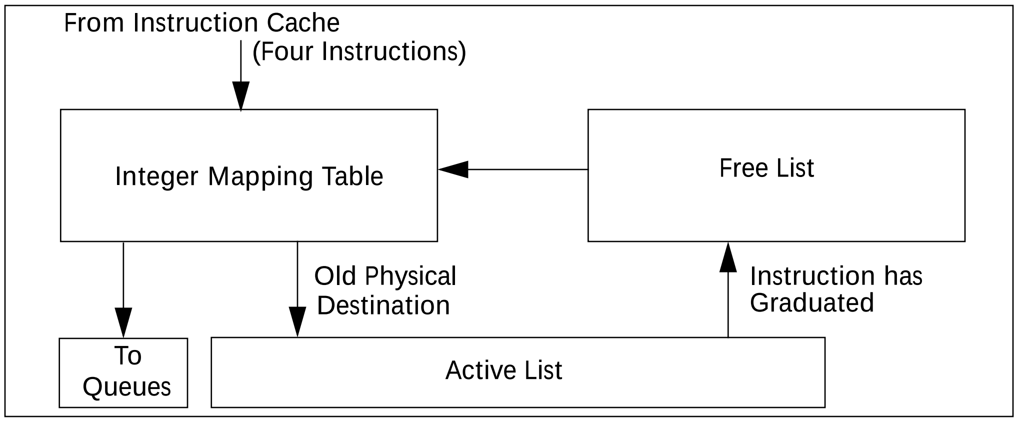


Figure 3-6 Integer Instruction Mapping Scheme

The active list maintains a listing of all 32 instructions in the pipeline at any given time. This list is always in order. The instructions in the queues can be executed out of order, but before the value can be stored as final, the result must be stored in the order determined by the active list. Once the value is stored it becomes obsolete and is no longer active. The logical destination can then be returned to the free list.

Each instruction can be uniquely identified by its location within the active list. A 5-bit value called the *instruction's tag* accompanies each instruction and allows it to be easily located within the 32-instruction active list so that it can be marked as done when the instruction graduates

When a value is taken from the free list it is passed to the mapping table and the mapping table is updated. The register value now contains the current value of an operand. The old value from the mapping table is then placed on the active list. The value remains on the active list until the instruction graduates, meaning that it has been completed in program order. An instruction can graduate only after it and all previous instructions have been successfully completed. Once an instruction has graduated, previous values are lost.

The R10000 microprocessor contains 64 physical registers and 32 logical registers. The active list contains a maximum of 32 values. The free list also has a maximum of 32 values. If the active list is full there could be 32 committed values and 32 temporary values, hence the need for 64 physical registers.

3.10.8 Execution Units

The R10000 microprocessor contains five execution units which operate independently of one another. There are two integer arithmetic logic units (ALU), two primary floating-point units, and two secondary floating-point units which handle long-latency instructions such as *divide* and *square root*.

3.10.8.1 Integer ALUs

There are two integer ALUs in the R10000 microprocessor defined as ALU1 and ALU2. Integer ALU operations, with the exception of the multiply and divide operations, execute with a one-cycle latency and a one-cycle repeat rate.

Both ALUs perform standard add, subtract, and logical operations. These operations complete in one cycle. ALU1 handles all branch and shift instructions, while ALU2 handles all multiply and divide operations using iterative algorithms. Integer multiply and divide instructions place their results in the EntryHi and EntryLo registers.

During multiply operations other single-cycle instructions can be executed within ALU2 while the multiplier is busy. However, once the multiplier has finished, ALU2 is busy for two cycles while the result is stored in two registers. For divide operations which have extra long latencies, ALU2 is busy for the duration of the operation.

Integer multiply operations generate a double-precision product. For single-precision operations the result is sign-extended to 64 bits before being placed in the *EntryHi* and *EntryLo* registers. Double-precision latencies are approximately twice that of single precision. Refer to Table 2, “Instruction Latencies and Repeat Rates,” on page 43.

3.10.8.2 Floating-Point Units

The R10000 microprocessor contains two primary floating-point units. The adder unit handles add operations and the multiply unit handles multiply operations. In addition, two secondary floating-point units exist which handle long-latency operations such as divide and square root.

Addition, subtraction, and conversion instructions have a 2-cycle latency and a 1-cycle repeat rate and are handled within the adder unit. Instructions which convert integer values to single-precision floating-point values have a 4-cycle latency as they must pass through the adder twice. The adder is busy during the second cycle after the instruction is issued.

Table 2: shows latency and repeat rates for integer and floating-point units.

Table 2: Instruction Latencies and Repeat Rates

Instruction	Latency	Repeat Rate
Integer Add, Subtract, Logical Operations, branches	1	1
Integer Load/Store (primary cache hit)	2	1
Integer Multiply (single precision)	5 (Lo)–6 (Hi)	6
Integer Multiply (double precision)	9 (Lo)–10 (Hi)	10
Integer Divide (single precision)	34 (Lo)–35 (Hi)	35
Integer Divide (double precision)	66 (Lo)–67 (Hi)	67
Integer to Floating-Point conversion (single precision)	4	1
Floating-Point Add, Subtract, Conversion, Logical operations	2	1
Floating-Point Load/Store	3	1
Floating-Point Multiply (double precision)	2	1
Floating-Point Multiply-Add	2/4	1
Floating-Point Divide (single precision)	12	14
Floating-Point Divide (double precision)	19	21
Floating-Point Square Root (single precision)	18	20
Floating-Point Square Root (double precision)	33	35
Floating-Point Reciprocal Square Root (single precision)	30	20
Floating-Point Reciprocal Square Root (double precision)	52	35
Integer Add, Subtract, Logical Operations, Branches	1	1
Integer Load/Store (primary cache hit)	2	1
Integer Multiply (single precision)	5 (Lo)–6 (Hi)	6
Integer Multiply (double precision)	9 (Lo)–10 (Hi)	10

All floating-point multiply operations execute with a two-cycle latency and a one-cycle repeat rate and are handled within the multiplier unit. The multiplier performs multiply operations. The floating-point divide and square root units perform calculations using iterative algorithms. These units are not pipelined and cannot begin another operation until the current operation is completed. Thus, the repeat rate approximately equals the latency. The ports of the multiplier are shared with the divide and square root units. A cycle is lost at the beginning of the operation (to fetch the operand) and at the end (to store the result).

The floating-point multiply-add operation, which occurs frequently, is computed using separate multiply and add operations. The multiply-add instruction (MADD) has a four-cycle latency and a one-cycle repeat rate. The combined instruction improves performance by eliminating the fetching and decoding of an extra instruction.

The divide and square root units use separate circuitry and can be operated simultaneously. However, the floating-point queue cannot issue both instructions during the same cycle.

3.10.9 Load/Store Units and the TLB

Load/Store units consist of the address queue, address calculation unit, translation lookaside buffer (TLB), address stack, store buffer, and primary data cache. Load/Store units perform load, store, prefetch, and cache instructions.

All load or store instructions begin with a three-cycle sequence which issues the instruction, calculates its virtual address, and translates the virtual address to the physical address. The address is translated only once during the operation. The data cache is accessed and the required data transfer is completed, provided there was a primary data cache hit.

If there is a cache miss, or if the necessary shared register ports are busy, the data cache and data cache tag access must be repeated after the data is obtained from either the secondary cache or main memory.

The TLB contains 64 entries and translates virtual addresses to physical addresses. The virtual address can originate from either the address calculation unit or the program counter (PC).

3.10.9.1 Secondary Cache Interface

Secondary cache support for the R10000 microprocessor is provided by an internal secondary cache controller with a dedicated secondary cache port. A dedicated 128-bit bus transfers data yielding a maximum secondary cache data transfer rate of 3.2GB per second. The R10000 microprocessor also provides a 64-bit system interface data bus.

The secondary cache is implemented as two-way set associative. Maximum cache size is 16MB. Minimum cache size is 512KB. Transfer width is 128 bits, or (4) 32-bit words. Consecutive cycles are used to transfer larger blocks of data as shown below.

- Four-word accesses (128 bits) are used for the CACHE instruction.
- Eight-word accesses (256 bits) are used for primary data cache refills and write-backs.
- 16-word accesses (512 bits) are used for primary instruction cache refills; SCache refills and write-backs (if SCache line size is selected to be 16 words).
- 32-word accesses (1024 bits) are used for secondary cache refills and write-backs (if SCache line size is set to 32 words).

3.10.9.2 System Interface

The system interface of the R10000 microprocessor provides a gateway between the R10000 and its associated secondary cache, and the rest of the computer system. The system interface operates at the frequency of *SysClk* being supplied to the processor. The programmability of the system interface allows for clock speeds of 200, 133, 100, 80, 67, 57, and 50 MHz. All system interface outputs, as well as all inputs, are clocked on the rising edge of *SysClk*, allowing the system interface to run at the highest possible clock frequency.

In most microprocessor systems only one system transaction can occur at any given time. The R10000 microprocessor supports a split-level bus transaction protocol. Split-transaction allows additional processor and external requests to be issued while waiting for a previous response. A maximum of four outstanding transactions are supported at any given time.

3.10.9.3 Multiprocessor Support

Two configurations of multiprocessor systems can be implemented using the R10000 microprocessor. One way is to have a dedicated external agent interface to each processor. The external agent is typically an ASIC which provides a gateway to the memory and I/O subsystems. In this type of configuration the processors do not interface directly together but rather through each external agent. Although this implementation is commonly used, cost as well as overall system complexity are increased due to the fact that at least one external agent must accompany each processor.

The R10000 microprocessor provides pin support for a cluster bus configuration. In a cluster configuration, up to four R10000 CPUs may be connected together via a cluster bus. Only one external agent is then required to interface to other system resources. Each processor interfaces to the same external agent. The cluster-bus implementation reduces not only the complexity but the number of ASICs, and hence the cost of the system by requiring only one external agent per four processors.

In addition to the 64-bit multiplexed address/data bus, a two-bit state bus is used for issuing processor coherency state responses. Also, a five-bit system response bus is used by the external agent for issuing external completion responses. Figure 3-7 shows a block diagram of a cluster bus configuration.

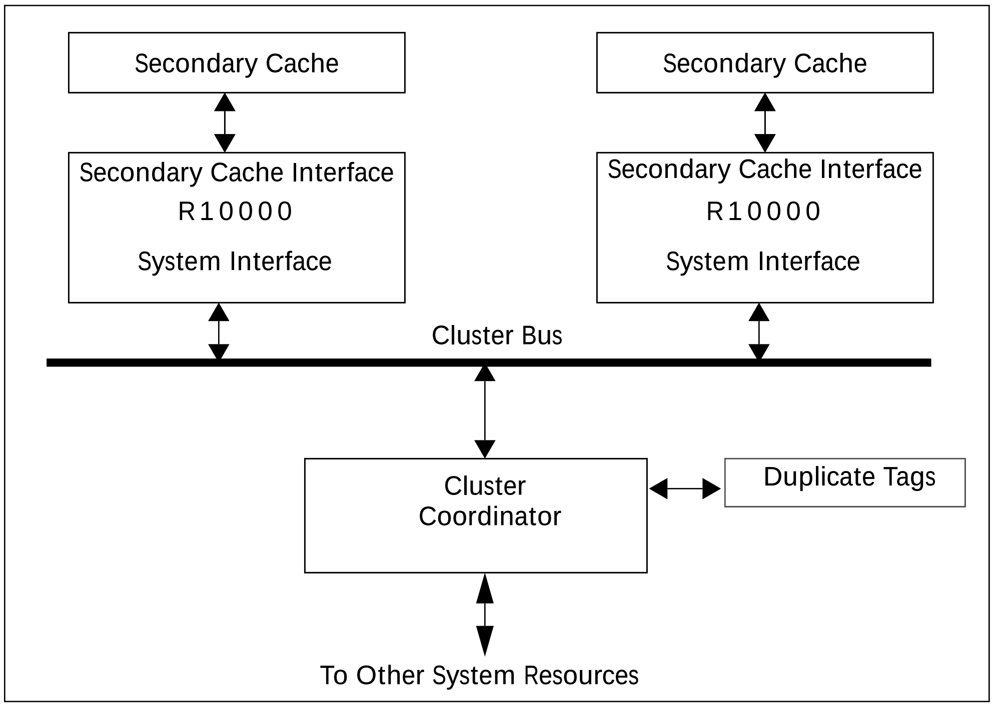


Figure 3-7 Multiprocessor System Using the Cluster Bus

3.11 R8000™ Processor Architecture

The MIPS® R8000 microprocessor chip set supports the MIPS IV Instruction Architecture (ISA). Its superscalar implementation achieves extremely affordable supercomputer performance, providing 360 MFLOPS peak floating-point performance. The R8000 processors combine a large fast-access, high-throughput cache subsystem with high-performance floating-point capabilities. Its bandwidth satisfies applications with large working sets of data. The R8000 system can sustain levels of performance once only possible on much larger and more costly supercomputer systems.

In the past, microprocessors typically had to restrict on-chip functionality. Small cache subsystems and other limiting features degraded performance for large scientific applications. Silicon Graphics leveraged new chip technology and an advanced instruction set architecture to overcome these limitations and expand applicability. The highly integrated R8000 chip set successfully meets the requirements of numeric-intensive applications in the technical market.

The chip set delivers peak performance of 360 double-precision MFLOPS and 360 MIPS/90 MHz. The R8000 uses the MIPS IV instruction set, which is a superset of the MIPS III architecture and is backward-compatible with all previous MIPS processors. Figure 3-8 is a block diagram of the R8000 chip set.

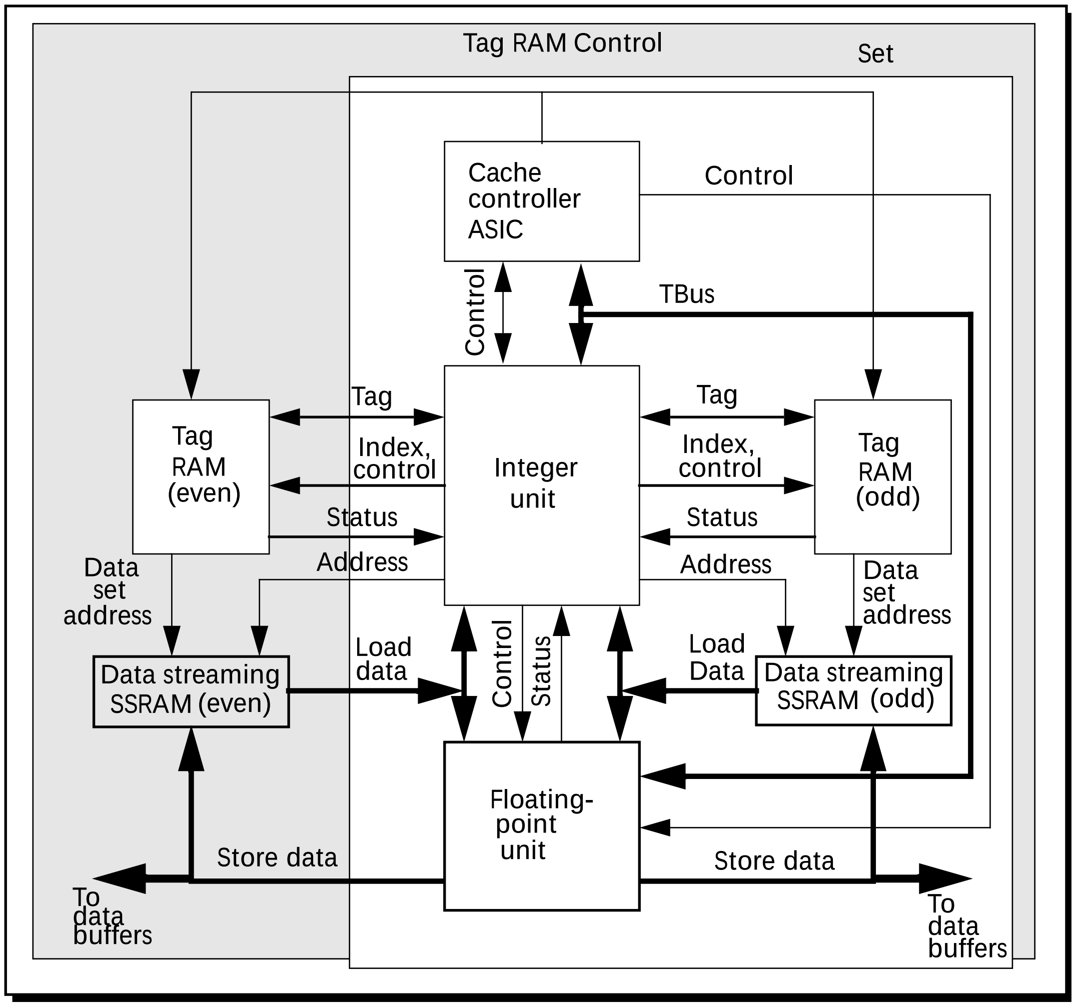


Figure 3-8 R8000 Microprocessor Chip Set Block Diagram

The MIPS R8000 processor is designed to deliver extremely high floating-point performance. Its key features include:

- Multicomponent chip set consisting of an integer unit (IU), floating-point unit (FPU), tag RAMs, and 4 MB of data-streaming cache
- Four-way superscalar architecture, six operations per clock cycle
- True 64-bit microprocessor with 64-bit integer and floating-point operations, registers, and virtual addresses
- 16 KB of instruction cache (I-cache) in IU, 16 KB of dual-ported data cache (D-cache) in IU, 1 K entries of branch prediction cache
- Memory Management Unit (MMU) in IU contains a 384-entry, dual-ported, three-way set associative Translation Lookaside Buffer (TLB)
- ANSI/IEEE-754 standard floating-point coprocessor with imprecise interrupts
- 32 double-word (64-bit) general-purpose registers in IU and 32 double-word (64-bit) floating-point registers in FPU
- 128-bit data bus and a separate 40-bit address bus that can access up to 1 TB of physical memory
- Full compatibility with earlier 32-bit and 64-bit MIPS microprocessors

This section explains the R8000 microprocessor’s:

- Superscalar implementation
- Integer unit organization
- Integer operations
- Floating-point unit organization
- Data-streaming cache and tag RAM
- FPU operations

3.11.1 Superscalar Implementation

The R8000 chip set is a cost-effective solution for the high-performance scientific computing community. A superscalar implementation was selected for:

- Sustained high performance on vectorizable code
- Accelerated compute-intensive scalar code
- Maintained binary compatibility with low-end products

Several features and design concepts collectively enable a cost-effective solution. This section describes the R8000’s balanced memory throughput implementation, extensions to the instruction set architecture and addressing modes, and data streaming cache.

3.11.1.1 Balanced Memory Throughput Implementation

The R8000 chip set is optimized for floating-point performance. However, both floating-point and integer computational capabilities are balanced with the required memory throughput. Two load/store units in the IU provide integer and floating-point functional units with the 64-bit (double word) information required for sustained operation and support up to two memory references in a single cycle. Any combination of floating-point and integer loads and stores are possible, except one integer store followed by another in the same cycle. There are no restrictions on pipelining any combination of floating-point and integer loads and stores.

The R8000 can send four instructions including two memory accesses per cycle:

- In total, up to two integer instructions can be dispatched to the integer functional units (two integer ALU units, one integer multiply unit, and one branch unit) on the IU. Only one integer instruction can be dispatched to an integer functional unit.
- In total, two floating-point instructions can be dispatched to the floating-point functional units on the FPU. The dual floating-point functional units in the FPU handle parallel multiply, add, multiply-add, divide, and square root operations. These can execute up to four floating-point operations per cycle.

The characteristics of the R8000 memory subsystem—number of ports, sizes and algorithms of the caches, tag RAM, and buffering schemes—complement the high-performance computational capabilities of the R8000 and ensure that memory bandwidth demands from the floating-point and integer units are met.

3.11.1.2 Instruction and Addressing Mode Extensions

Several new instructions improve performance for numerically intensive code. Floating-point multiply-add instructions achieve results comparable to chaining vector operations: multiple floating-point operations execute each machine cycle. This results in greater precision and higher performance.

Many scientific applications have separately compiled subroutines containing parameter matrices with variable dimensions. Standard register-plus-offset addressing requires an extra integer addition for each access to these arrays. In contrast, the R8000’s indexed addressing mode (base register plus index register) eliminates the extra addition for floating-point loads and stores.

For high-end numeric processing, loops containing IF statements must execute efficiently. The R8000 design includes a set of four conditional move operators that allow IF statements to be represented without branches. The bodies of THEN and ELSE clauses are unconditionally computed, and results are put in temporary registers. Conditional move operators then selectively transfer the temporary results to their true register. In summary, both legs of an IF statement are computed, and one of them is discarded.

3.11.1.3 Data Streaming Cache Architecture

The R8000 chip set incorporates a unique coherent cache scheme to address two disparate computational requirements. Most programs contain a mix of integer, floating-point, and address computation. Integer and address computation is best accomplished with a moderately sized low-latency, fast data cache. Floating-point calculations require large amounts of memory or a large data cache subsystem. The cache need only be moderately fast, but it must be sizable enough to hold large data sets and have high throughput to the floating-point functional units.

The R8000 chip set provides a unique cache hierarchial implementation. The level-one data cache, on the integer chip, is 16KB and allows very fast access for integer loads and stores. A large 4MB off-chip cache, called the data streaming cache, serves as a second-level cache for integer data and instructions, and as a first-level cache for floating-point data. This configuration allows floating-point loads and stores to bypass the on-chip cache and communicate with the large off-chip cache directly. The data streaming cache is pipelined to allow for continuous access by the floating-point functional units. Data can be transferred at the rate of two 64-bit double words per cycle, or 1.2GB per second. The large pipelined data streaming cache provides the capacity and sustained bandwidth needed to handle floating-point data objects.

3.11.2 R8000 Integer Unit Organization

The IU chip, a 591-pin device, is the main computing component of the R8000 microprocessor chip set. Its multiple execution units are supported by dedicated buses that allow independent operation of functional units. The complex bussing scheme alleviates the need for multiplexing data and addresses and allows simultaneous floating-point loads and stores from and to the data streaming cache. Figure 3-9 on page 51 is a block diagram of the R8000 unit.

The IU contains four caches: the instruction cache, the data cache, the branch prediction cache, and the Translation Lookaside Buffer (TLB) cache. Dedicated interfaces let the IU generate and provide address information to the data streaming cache, the floating-point unit, and the tag RAMs. An additional 80-bit bus allows the IU and FPU to communicate. The IU also contains two integer arithmetic logic units (ALUs) and two address generation units. These functional units support up to four instructions per cycle—two integer instructions and two data accesses.

3.11.2.1 Instruction Cache and Instruction Cache Tag RAM

The instruction cache contains instructions to be executed. The 16-KB I-cache in the IU is direct-mapped, arranged as 1024 entries by 128 bits: each entry contains four 32-bit instructions and every access to the cache fetches four instructions. The I-cache is virtually indexed and virtually tagged, alleviating the need for address translation on I-cache accesses.

The I-cache tag RAM determines if a valid instruction exists in the I-cache. The I-cache tag RAM has 512 entries, one tag for every two I-cache entries or one tag for each I-cache line. Each I-cache tag RAM entry contains a tag, an address space identifier (ASID), a tag valid bit, and two region bits. The ASID differentiates instructions between processes and allows instructions with the same virtual address, but different physical addresses from multiple processes, to reside in the I-cache at the same time. It also ensures that two processes accessing the same memory space will not overwrite each other's instructions.

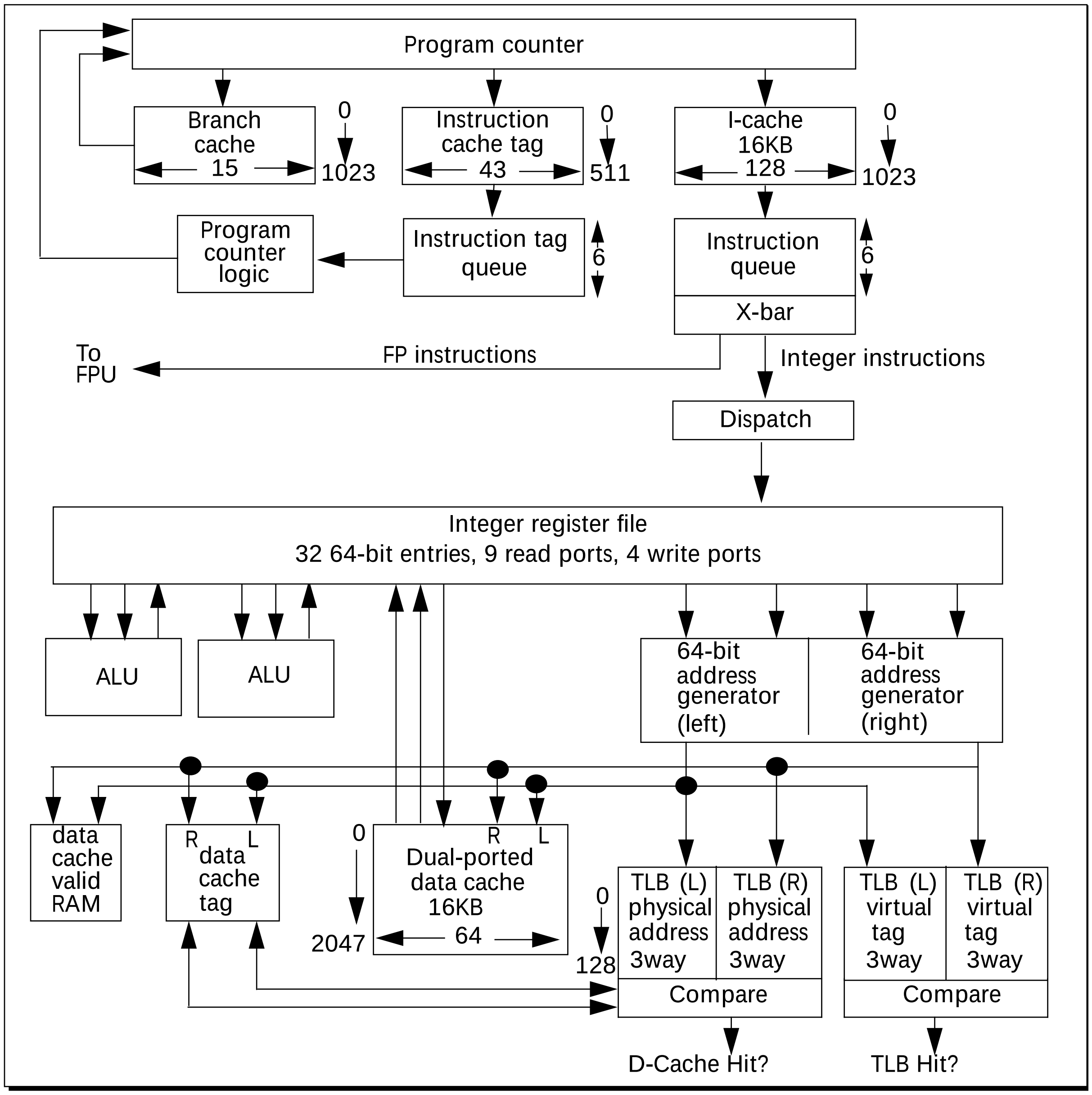


Figure 3-9 R8000 Integer Unit Block Diagram

3.11.2.2 Instruction/Tag Queues and Dispatching

Effective utilization of a superscalar processor requires access to multiple instructions per cycle and the ability to dispatch multiple instructions per cycle. In the R8000's creative approach for fetching and dispatching instructions, the integer unit fetches four instructions or 128 bits per cycle from the I-cache. These instructions are partially decoded and then placed in a six-deep temporary storage queue. The queue is required because dependencies might prevent instructions from being dispatched.

The X-bar can dispatch from zero to four instructions every cycle. It uses resource modeling to determine when instructions are ready to be dispatched. The X-bar monitors the status of each execution unit from cycle to cycle and determines interdependencies between any of the four instructions in a given line. Figure 3-10 diagrams instruction dispatch.

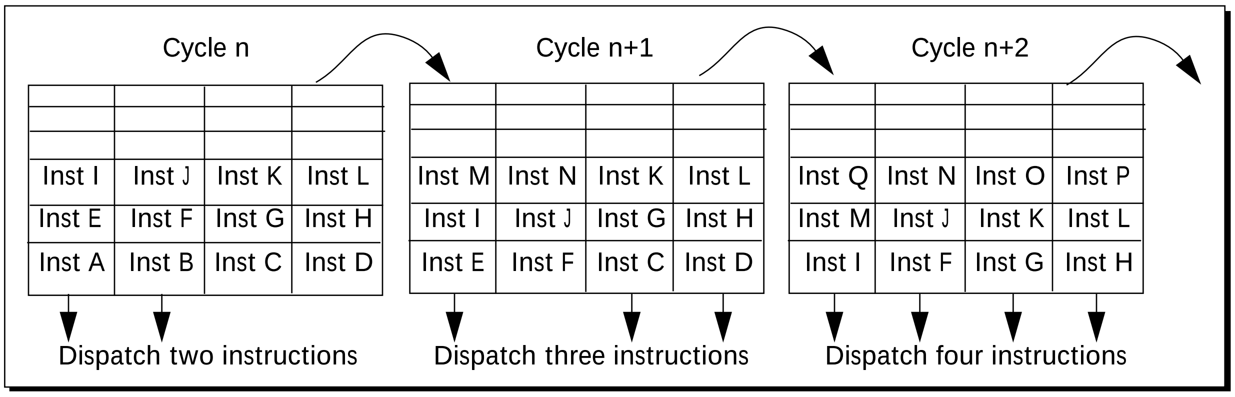

Cycle n
Cycle n+1
Cycle n+2
Inst I Inst J Inst K Inst L
Inst E Inst F Inst G Inst H
Inst A Inst B Inst C Inst D
Inst M Inst N Inst K Inst L
Inst I Inst J Inst G Inst H
Inst E Inst F Inst C Inst D
Inst Q Inst N Inst O Inst P
Inst M Inst J Inst K Inst L
Inst I Inst F Inst G Inst H
Dispatch two instructions
Dispatch three instructions
Dispatch four instructions

Figure 3-10 R8000 Instruction Dispatch

3.11.2.3 Branch Prediction Cache and Branch Prediction

When a branch is taken, the contents of the program counter are altered. In a pipelined processor implementation, this forces one or more wasted cycles and reduces the performance of the processor. Research shows that during execution of a typical application, a branch instruction occurs every six to eight instructions. Hence, in the superscalar R8000 processor, a branch can be expected every other cycle. Branch prediction is therefore crucial for maintaining the pipeline and ensuring continuous execution. Figure 3-11 diagrams the branch prediction cache.

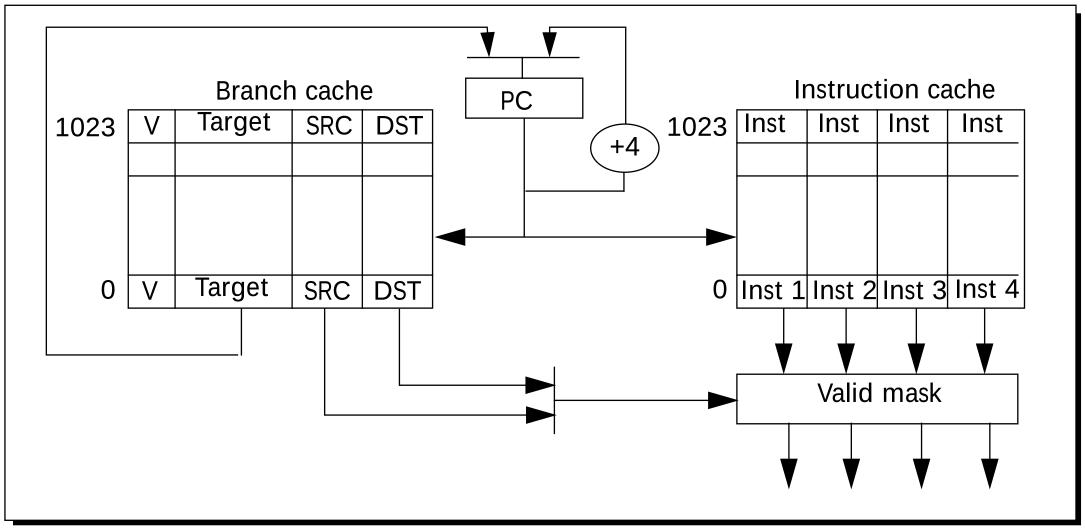


Figure 3-11 R8000 Branch Prediction Cache

The R8000 branch prediction cache is used to modify the program counter (the location of the target instruction) when the processor encounters a branch instruction. It works in conjunction with the I-cache and incorporates a simple branch prediction mechanism. The branch prediction cache has 1024 entries, one for each entry in the I-cache. Each entry is 15 bits wide, ten of which are used as a branch target address to any of the 1024 entries in the I-cache. The remaining bits indicate whether the branch predicted is taken, and where in the target quadword execution should begin. Table 3: summarizes parameters for the instruction cache and branch prediction cache.

Table 3: Parameters for R8000 Instruction Cache and Branch Prediction Cache

Parameter	Instruction Cache	Branch Prediction Cache
Location	IU	IU
Contents/entry	128 bits (4 instructions	16 bits
Size	16KB	2KB
Mapping	Direct-mapped	Direct-mapped
Index	Virtual address	Virtual address
Tag	Virtual address	N/A
Data access	Single cycle	Single cycle
Data transfer	128 bits/4 instructions per cycle	16bits per cycle
Cache bandwidth	1.2GB per second	159MB per second
Line size	32 Bytes or 8 words	N/A
Miss penalty	11 cycles to D8 cache	3 cycles to actual branch result

3.11.2.4 Integer Register File

The integer register file consists of 32 64-bit entries and contains the data for integer instructions and address generation. It has nine *read ports* and four *write ports* to allow for parallel execution of integer instructions and for parallel access to the on-chip data cache. The four write ports define the number of operations that can be completed in parallel (in this case, two integer instructions and two loads every cycle).

Four of the read ports and two of the write ports are dedicated to simultaneous integer ALU usage. Four more read ports are used for address generation during integer or floating-point memory operations; two write ports are used to load the register file from the data cache. The ninth read port, a bypass port, is used for data cache stores and unaligned loads.

3.11.2.5 Arithmetic Logic Units

Two 64-bit arithmetic logic units, one shifter, and one multiply/divide unit execute all R8000 integer arithmetic operations. All arithmetic operations occur in the execution stage (E-stage) of the integer pipeline. Results are written back to the register file previously described. Figure 3-12 diagrams the R800 ALU.

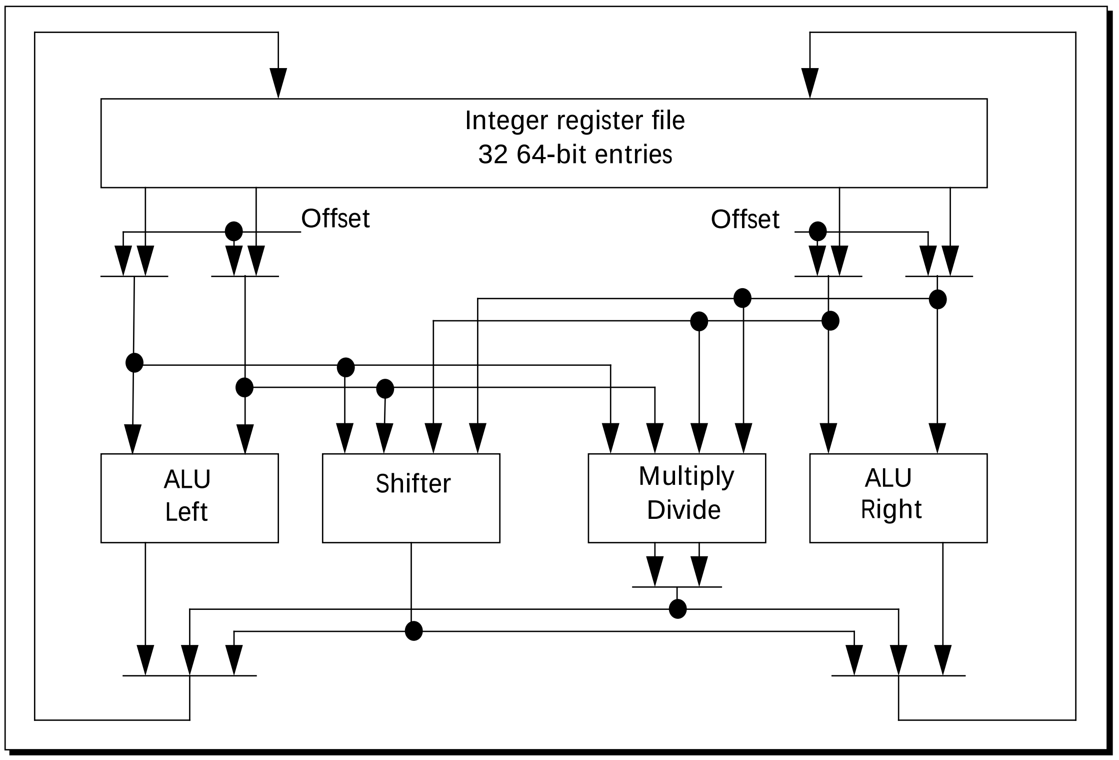


Figure 3-12 R8000 Arithmetic Logic Unit Block Diagram

3.11.2.6 Translation Lookaside Buffer (TLB) Organization

The translation lookaside buffer converts virtual addresses to physical addresses. A single TLB in the IU handles instruction references when there is an I-cache miss, and all data references. The TLB contains 384 entries to reduce TLB misses when processing large matrices, and is three-way set associative to maintain high performance.

The TLB is dual ported to allow for parallel references. It is split into two sections—one section contains virtual tags (VTAGS), the other section contains the actual physical address (PA) corresponding to each virtual tag. Table 4: compares the POWER CHALLENGE and CHALLENGE TLBs.

Table 4: Translation Lookaside Buffer Comparison

Parameter	POWER CHALLENGE TLB/ R8000	CHALLENGE TLB/R4000
Contents	**Virtual-to-Physical Address Translation75**	
Size	384 entries: one translation/entry	48 Entries: two translations/entry (even–odd pages)
Mapping	3-way set associative random placement	Fully associative random placement
Index	Virtual address	Virtual address
Ports	Two	One
Access	Single cycle	Single cycle
Kernel page size	16K	4K
User page size	16K	4K
Size	384 entries: one translation/entry	48 entries: two translations/entry (even–odd pages)

3.11.2.7 Data Cache and Data Cache Tag

The 16KB data cache in the IU is dual-ported and is arranged as 2048 entries by 64 bits. The direct-mapped D-cache is virtually indexed and physically tagged; it implements a write-through protocol to maintain coherency with the data streaming cache. The D-cache allows either two loads or one load and one store to occur simultaneously. Loads and stores of data on a byte (8-bit) resolution are also supported.

The D-cache tag RAM in the IU contains 512 entries—one entry for every 32-byte D-cache line. The physical address tag stored in the D-cache tag RAM is compared to the translated address from the TLB to determine if the requested data is in the data cache. If the tags are equal, the requested data is in the data cache; if the tags are not equal, a D-cache miss has occurred and a memory cycle to the data streaming cache is initiated.

Data cache access and TLB lookups are performed in the execution stage (E-stage) of the pipeline. This simultaneous activity determines if requested data resides in the D-cache. TLB is checked to determine if a valid virtual to physical address translation exists and the D-cache is checked to determine if the data resident in the D-cache is valid. Figure 3-13 diagrams the data cache and TLB.

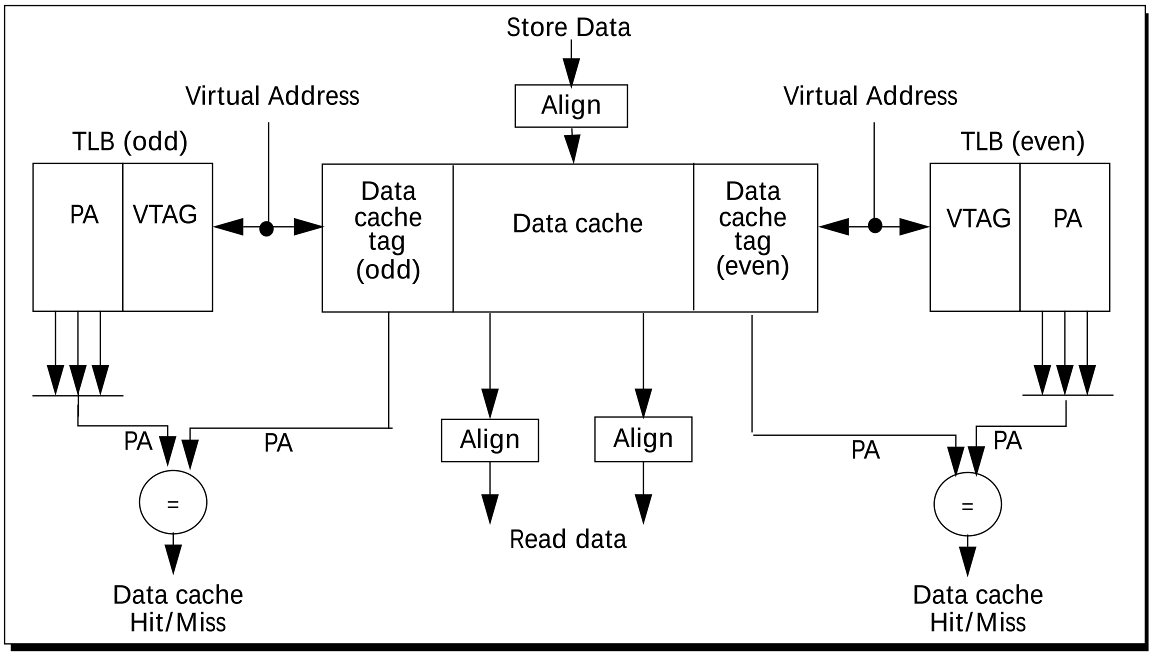


Figure 3-13 Dual-Ported Data Cache and TLB

3.11.3 Integer Operations

The IU performs several types of integer operations. Each operation requires a functional unit for some number of cycles (referred to as staging) and has some fixed time before the results are available (referred to as latency). Table 5: summarizes integer latencies.

Table 5: R8000 Integer Latencies

Integer Unit Operation	Latency (Cycle Count)
Add, shift, logical	1
Load, store	1
Multiply	4 (32-bit operands) 6 (964-bit operands)
Divide	21 (quotient ≤15 bits 39 (quotient 16-31 bits) 73 (quotient 32-64 bits)

Most integer operations execute in one cycle, but some operations, such as integer multiply and divide, require additional cycles. The R8000 integer multi-ply operation is one of the fastest implementations available, greatly improving execution of loops that require integer multiplication. A clever approach was used for the integer division operation: the time required for an integer divide operation is a function of the quotient size.

3.11.4 R8000 Floating-Point Unit Organization

The FPU chip, a 591-pin device, performs all floating-point functions for the R8000. The FPU has two fully-pipelined execution units, allowing two floating-point mathematical operations and two floating-point memory operations every cycle. The FPU register file contains 32 64-bit entries and has eight read ports and four write ports. Load and store data queues provide a pipelined inter-face between the IU and FPU, streamlining data flow and minimizing unused cycles. The FPU offers peak performance of 300 double-precision MFLOPS, at a clock frequency of 75 MHz. Figure 3-14 is a floating-point unit block diagram.

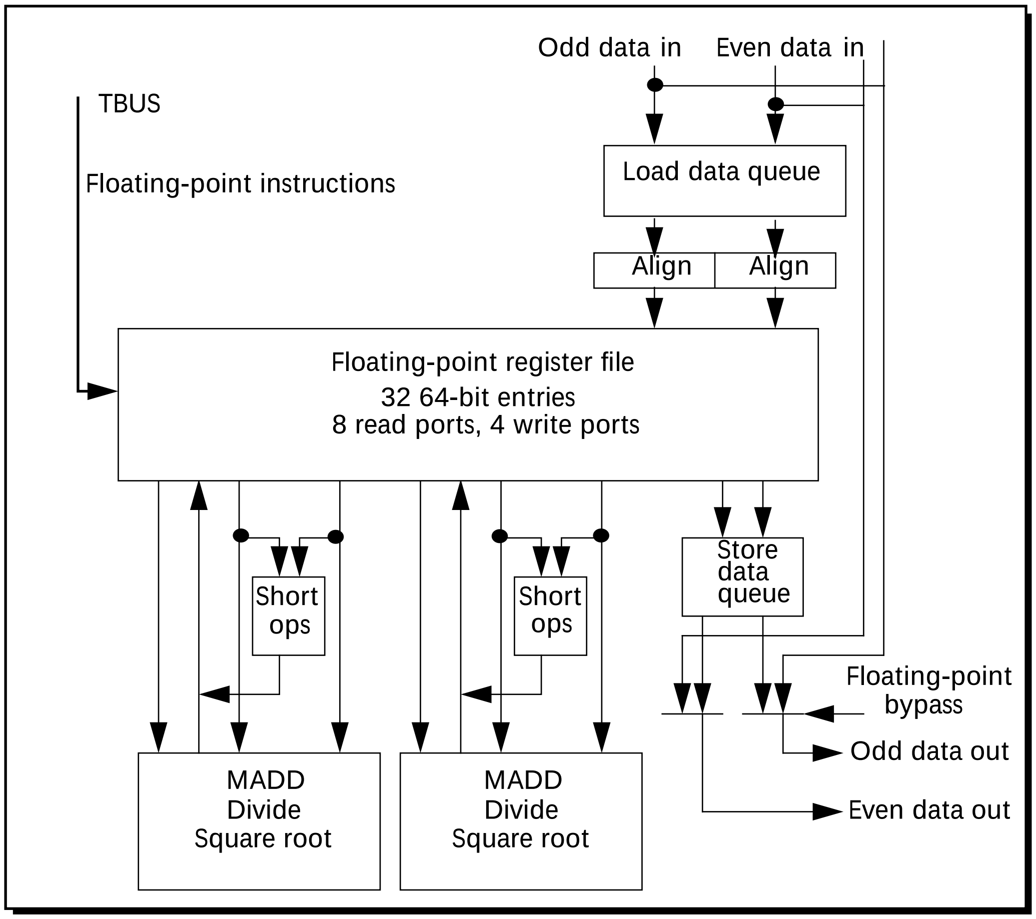


Figure 3-14 R8000 Floating-point Unit Block Diagram

The IU places floating-point instructions in the floating-point instruction queue in the IU. Each entry in the floating-point instruction queue is arranged as a quad word. The dispatch mechanism is similar to that of the IU: from zero to four floating-point instructions can be dispatched by the IU to the FPU each cycle. Instructions are dispatched only when the FPU has adequate resources available to execute the instructions. The floating-point instruction queue provides temporary storage for floating-point instructions while any dependencies that might prevent the floating-point instructions from being executed (such as waiting for dependent loads to complete, etc.) are cleaned up.

Floating-point instruction dispatches and floating-point loads and stores to the data streaming cache are controlled by the IU. Accesses that miss in the data streaming cache and require interfacing to main memory are handled by the off-chip cache controller. During these miss cycles, the FPU continues to execute floating-point instructions already in the queue.

Once floating-point data is retrieved from the data streaming cache, it is placed in the load data queue. For store operations, the result is placed in the store data queue. As soon as the corresponding address information from the tag RAM is available, the data is written out to the data streaming cache. Figure 3-15 diagrams the floating-point data path.

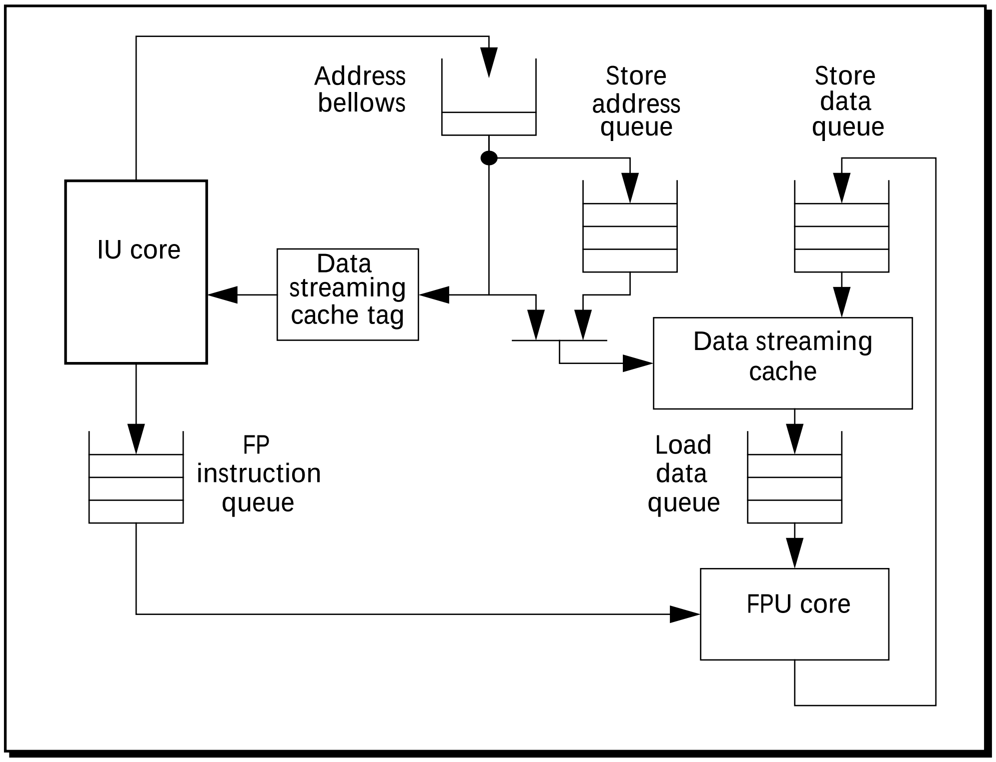


Figure 3-15 R8000 Floating-Point Data Path

3.11.5 Data Streaming Cache and Tag RAM

The R8000 chip set employs a very large 4MB, four-way set associative external data streaming cache. The large cache size, coupled with blocked data, greatly improves the performance of engineering and scientific applications with large data sets. The set associativity reduces the thrashing common with direct-mapped caches and increases the effective size of the cache. Studies show that four-way set associative caches exhibit miss rates similar to those of non-associative caches that are twice the size of the associative caches.

The data streaming cache is set as separate load and store data buses, eliminat-ing the bus turnaround time. Load and store operations can be fully pipelined, making it possible to issue two memory operations to the data streaming cache every cycle. The 4MB data streaming cache is split between even and odd 2MB banks; one memory operation can go to each bank every cycle.

The data streaming cache tag RAM is four-way set-associative and is organized as 2048 entries by 128 bits. Each 128-bit entry contains 32 bits of information for each of the four sets. Each bank of the data streaming cache employs a dedicated custom tag RAM to address the cache. The dual tag RAMS also allow simultaneous and independent operation of the data streaming cache banks every cycle. Figure 3-16 diagrams data-streaming cache organization.

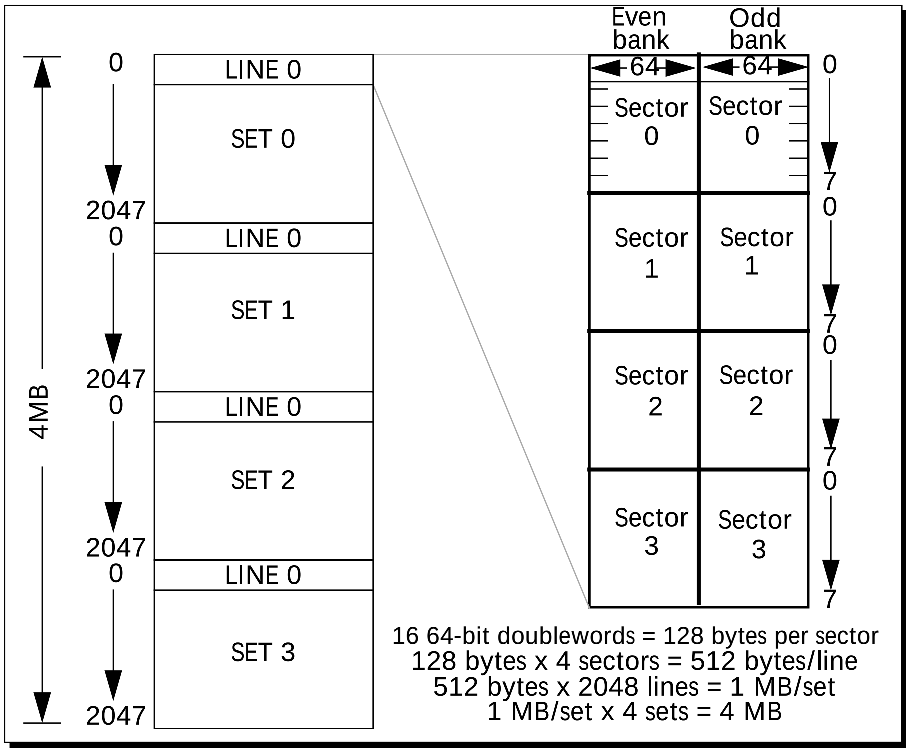

4MB
0
2047
LINE 0
SET 0
SET 1
SET 2
SET 3
Even bank
Odd bank
64
Sector 0
Sector 1
Sector 2
Sector 3
0
7
16 64-bit doublewords = 128 bytes per sector
128 bytes x 4 sectors = 512 bytes/line
512 bytes x 2048 lines = 1 MB/set
1 MB/set x 4 sets = 4 MB

Figure 3-16 R8000 Data Streaming Cache Organization

Table 6: compares the data streaming cache and level-two secondary cache.

Table 6: Data Streaming Cache and Level-2 Secondary Cache Comparison

Parameter	POWER CHALLENGE Data Streaming Cache (R8000)	CHALLENGE Secondary Cache (R4400)
Location	Off-chip; 4 SIMMs w/SSRAMS (12 nanoseconds)	Off-chip; 4 SIMMs w/ SRAMS (10 nanoseconds)
Contents	FP data/integer data instructions	FP data/integer data instructions
Size	4MB	1MB
Mapping	4-way set associative random placement	Direct-mapped, no hashing
Index	Physical address	Physical address
Tag	Physical address	Physical address
Coherency policy on writes	Write back	Write back
Coherency protocol	MESI	MESI
Coherency maintenance	In hardware	In hardware
Data Protection	Parity on 16-bit quantities	SECDED-Single-error correction, double-error detection
Ports	Single-ported to even bank single-ported to odd bank	Single-ported
Interleaving	Two-way: even/odd double words	N/A
Data access	5-stage fully pipelined	Asynchronous
Data transfer	2 64-bit double words per cycle (1 even double word per cycle, 1 odd double word per cycle)	128-bit per two cycle (one double word per cycle)
Latency	5 cycles initially, 0 cycles when pipeline is full	4 –11 cycles
Cache bandwidth	600MB per second even; 600MB per second odd; data streaming cache to/from registers	400MB/s–L2 cache t o L1 cache
Total cache bandwidth	1.2GB/s	400 MB/s
Line size	512 bytes (128 Words) split into 4 sectors; 128 bytes per sector (32 words)	128 bytes (32 words)
Miss penalty	53 cycles to main memory (128 Bytes); 80 cycles to DS cache of another CPU (128 bytes)	~105 cycles to main memory (R4400 100MHz) ~158 cycles to main memory (R4400 150MHz)
Access pattern	2 loads/cycle or 2 stores/cycle or 1 load and 1 store/cycle	1 load/cycle or 1 store/cycle
Dirtiness recorded	On a sector basis	On a line basis

In addition to being set associative, the data streaming cache is also two-way interleaved. This provides sufficient memory bandwidth to effectively use both floating-point functional units simultaneously, each cycle.

The design has two 64-bit operands to the FPU each cycle: an effective 1.2GB per second transfer rate. This lets each floating-point functional unit to execute a multiply/add every cycle. Figure 3-17 diagrams the data-streaming cache interleaving.

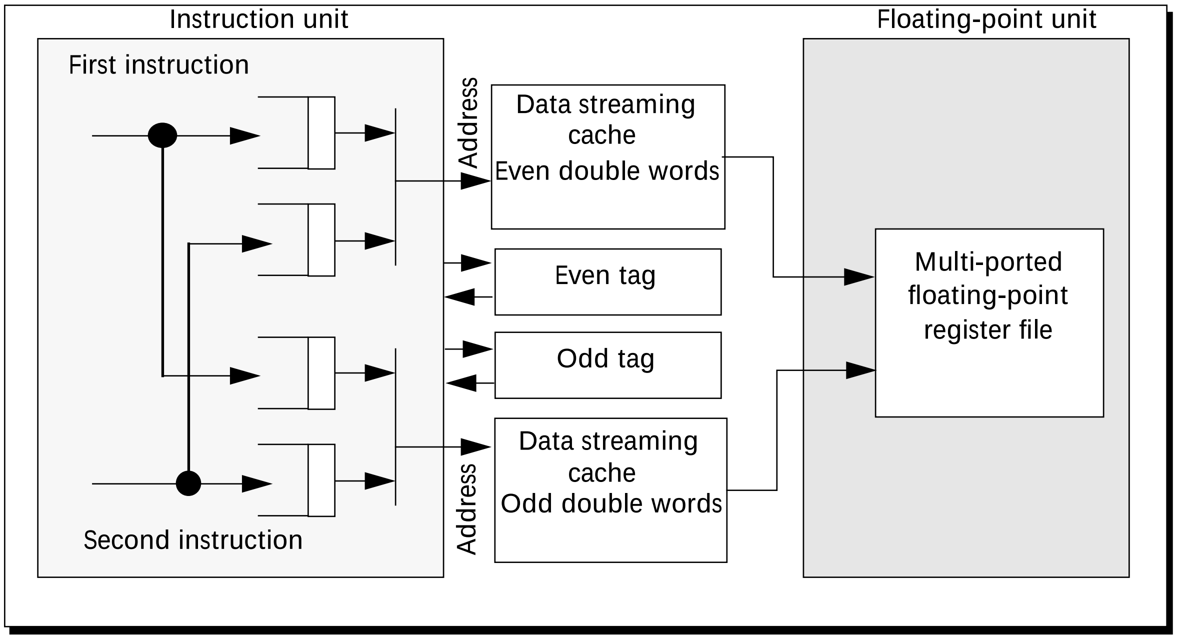

Instruction unit
Floating-point unit
First instruction
Second instruction
Address
Data streaming cache Even double words
Even tag
Odd tag
Address
Data streaming cache Odd double words
Multi-ported floating-point register file

Figure 3-17 R8000 Interleaved Data Streaming Cache

3.11.6 FPU Operations

The FPU performs three types of floating-point arithmetic operations: short, regular and long. Each operation requires a functional unit for some number of cycles (staging) and has a fixed time before the results are available (latency):

- Short operations (MOV, NEG, ABS) need one staging cycle and have a one-cycle latency.
- Regular operations (ADD, SUB, MADD, MSUB, MUL, CONVERT) require one staging cycle and have a four-cycle latency. They require one, two or three source operands from the FP register file and take four cycles to complete. On the next cycle the result is written back to the FP register file. On-chip bypass logic facilitates execution of data-dependant floating-point instructions.
- Long operations have variable latency and staging times.

Additionally, floating-point load and store operations are defined. When pipelined, these operations have an apparent latency of one cycle even though it takes five cycles to go through the data streaming cache. Table 7: summarizes latencies and staging associated with various FPU operations.

Table 7: R8000 FPU Operations and Associated Latencies/Staging

Floating-Point Unit Operation	Latency (Cycle Count)	Staging (Cycle Count)
Move, Negate, Absolute Value	1	1
Add, Multiply, MADD	4	1
Load, Store	1	1
Compare, Move, Conditional Move	1	1
Divide	14 (single precision) 20 (double precision)	11 (single precision) 17 (double precision)
Square Root	14 (single precision) 23 (double precision)	11 (single precision) 20 (double precision)
Reciprocal	8 (single precision) 14 (double precision)	5 (single precision) 11 (double precision)
Reciprocal Square Root	8 (single precision) 17 (double precision)	5 (single precision) 14 (double precision)

The R8000 microprocessor chip set delivers the floating-point performance and bandwidth necessary to accommodate large numerically-intensive applications. The floating-point structure and innovative caches provide capabilities not previously available from a high-volume RISC microprocessor.

3.12 POWERpath-2™ Coherent Interconnect

The heart of any multiprocessor system is its interconnect. The interconnect provides each processor with a path to memory and I/O, and may also support cache coherence and other features. POWER CHALLENGE multiprocessors use the POWERpath-2 as a coherent interconnect. The POWERpath-2 is a fast-and-wide split-transaction bus providing high-bandwidth, low-latency, cache-coherent communication between processors, memory, and I/O. Additional features include special transactions for efficient processor synchronization and I/O DMA transfers. Highlights include:

- 1.2GB per second sustained transfer rate
- 9.5 million transactions per second, sustained
- Snoop write-invalidate cache-coherence maintained in hardware
- Multiple outstanding, variable-duration split-read transactions
- Independent 256-bit data bus and 40-bit address bus
- 47.6MHz synchronous signaling (21-nanosecond cycle)
- ECC-protected memory and caches, parity-protected data and address buses

This section explains:

- POWERpath-2 protocol
- Cache coherency
- Bus timing
- Minimizing read latency
- Memory read response latency
- MP system latency reduction

3.12.1 POWERpath-2 Protocol

The POWERpath-2 protocol is designed with a RISC philosophy. The types and variations of transactions are small, and each transaction consumes exactly five cycles to transfer a single cache block of 128 bytes. Read transactions are split: independent address and data transactions can occur simultaneously, creating a pipeline effect. Thus bus performance can be optimized for common trans-actions, without requiring additional hardware complexity to handle rare cases.

3.12.2 Cache Coherency

The cache coherency protocol is identical to the Illinois Protocol [1], except that cache to cache transfers are only used for dirty data. Each cache has four states: invalid, exclusive, dirty exclusive, and shared. Transition between cache states is caused by actions initiated by the processor or by coherent transactions appearing on the bus. Figure 3-18 diagrams cache coherency state transactions.

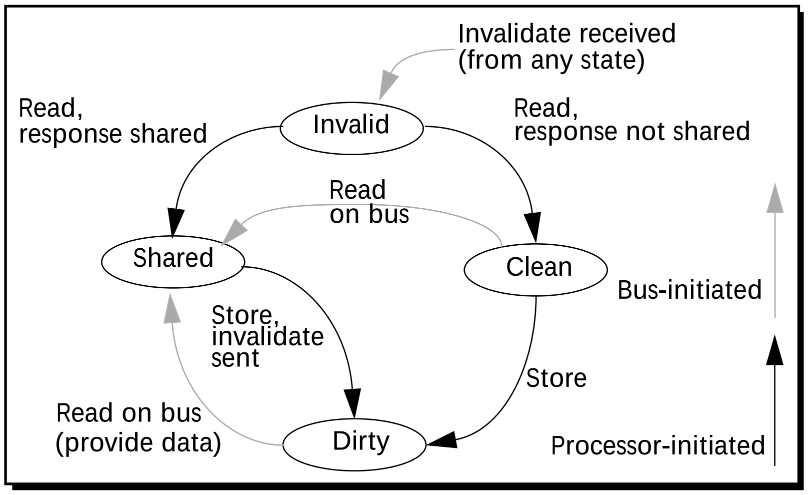


Figure 3-18 Cache Coherency State Transactions

To eliminate unnecessary cache contention between the processor and the bus snoopy mechanism, the process-bus interface maintains a duplicate set of cache tags[*]. A processor cache is accessed only for coherency reasons if the data in question actually resides in that cache; bus traffic targeting lines not cached by the local processor do not affect the processor or its cache.

Whenever a read request is satisfied by data from a processor's cache, the memory accepts the cache read response as a sharing writeback. This mechanism eliminates the need for a shared dirty state and reduces the bus bandwidth needed for write-backs, as data is cleaned as it passes across the bus from cache to cache. [†]

Pending reads are tracked by being associated with read resources. Read resources are associative memories which link the address of a pending read to a data response signal. The use of these read resources makes it possible to track and maintain coherency on read responses which may return out of order. When a read request is issued, it occupies the first available read resource. A pending read occupies a read resource until a corresponding read response appears on the bus. POWERpath-2 implements eight read resources, so that up to eight reads can be outstanding at any time. If all eight read resources are filled, future read requestors must wait until a read resource becomes available.

* Archibald, J., and Baer, J.L., "Cache Coherence Protocols: Evaluation Using Multiprocessor Simulation Model," ACM Transactions on Computer Systems, Vol. 4, No. 4, Nov 1986, pp. 273-298.

† Goodman, J. R., "Using Cache Memory to Reduce Processor-Memory Traffic." In Proceedings of the 10th International Symposium on Computer Architecture. IEEE, NY, 1983, pp. 124-131.

3.12.3 Bus Timing

Every POWERpath-2 bus transaction consists of five clock cycles. System wide, bus controllers execute the same five machine states synchronously: *arbitration, resolution, address, decode,* and *acknowledge*. This RISC approach to bus protocol simplifies controllers and lets them operate at maximum frequency with minimal design risk. Figure 3-19 diagrams a bus state transaction.

3.12.3.1

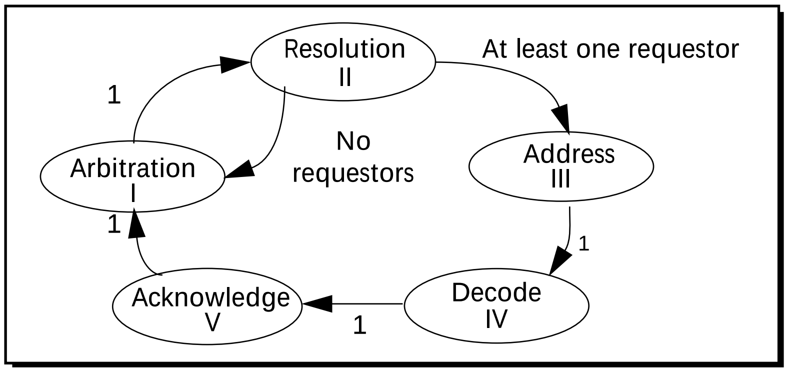


Figure 3-19 Bus State Transaction

When no transactions are occurring, each bus controller drops into a two-state idle machine. Thus, new requests appearing on an idle bus can arbitrate immediately, instead of having to wait for the arbitration cycle to arrive. Two states are needed to prevent different requestors from driving the arbitration lines on subsequent cycles.

Address and data buses are arbitrated separately to accommodate split-read transactions in a highly pipelined fashion. While one node is using the address bus to issue a new read request, another node can use the data bus to provide a read response to one of several pending reads. Under normal system loads, the memory system can provide a read response on the data bus two transactions after the read request appears on the address bus. For write requests, address and data appear simultaneously, while invalidate requests occupy only the address bus. With a sufficient number of requestors, the bus can sustain the peak transfer rate of 1.2GB per second.

Bus transactions are grouped into three categories... in order of priority:

- Read responses
- Address operations, which include read requests, invalidates, and interrupts
- Write operations

These categories are granted the bus in descending order of priority. Prioritizing read responses and address operations over writes reduces the stall time associated with reads and invalidates; since write-backs do not stall the processor, they are held off until no other operations are requested. Within each category, fair round-robin arbitration is used. Whenever possible, requests for the address and data buses are combined for greater bus efficiency.

POWERpath-2 employs a distributed arbitration scheme that reduces the overall time required to determine a bus winner, and thus reduces latency. A central arbitration scheme requires that all requests be issued to a single chip, which determines a winner and grants the bus. This system requires one additional transaction over the distributed arbitration scheme, where the knowledge of which node won the bus is immediately available to all nodes. Although distributed arbitration does require more logic to implement, higher gate counts available in today's integrated circuits make this cost low compared to the benefits of saving clock cycles.

3.12.3.2 Minimizing Read Latency

Memory read latency has one the most direct impacts on system performance of any design parameter. To achieve scalability in multiprocessor systems, it is particularly important that latencies remain low while larger numbers of processors and I/O devices saturate the bus with memory requests.

Latency reduction techniques used in POWERpath-2 can be divided into two broad categories: techniques that minimize latency for an single processor to achieve high performance for an individual node; techniques that minimize overall system latency to support scalable multiprocessing.

3.12.3.3 Memory Read Response Latency

Read latency to an individual processor is minimized by a number of design features in the processor interface implementation, in the bus protocol design, and in the memory system design. The numbered features in Figure 3-20 are explained below.

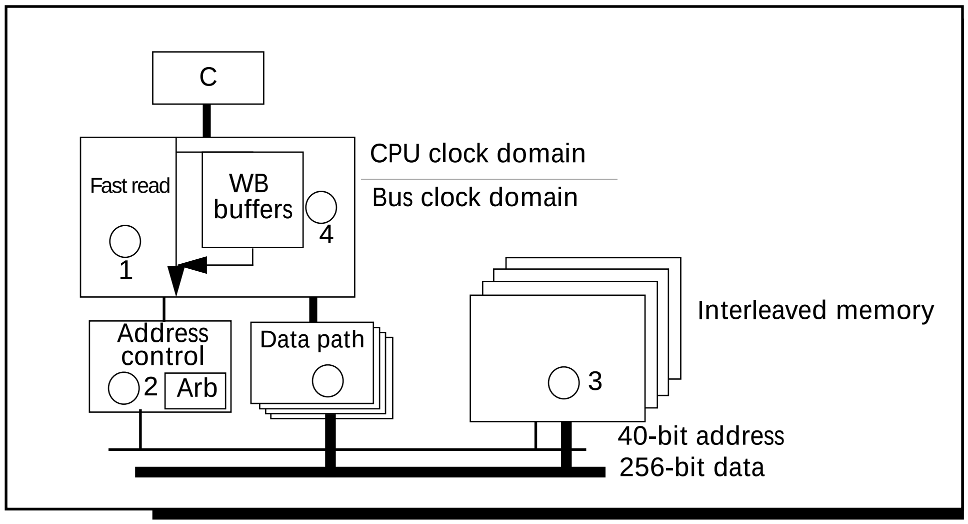


Figure 3-20 Read Latency Reduction Techniques

1. The logic in the processor interface chips is designed to anticipate and accelerate read requests. Other transactions, such as writes, suffer increased latency in this design, but have minimal impact on performance.
2. Arbitration for the POWERpath-2 bus favors read requests over other requests with less sensitivity to latency. Since the POWERpath-2 is designed with split address and data buses, read requests and read responses for independent transactions can occur simultaneously, as these paths share no common resource.
3. The POWER CHALLENGE memory system uses high-speed buffers to fan out addresses to a 576-bit wide DRAM bus. Fast page-mode accesses allow an entire 128-byte cache line to be read in two memory cycles, while data buffers pipeline the response to the 256-bit wide POWERpath-2 data bus. Twelve clock cycles after a read address appears on the address bus, response data appears on the data bus.
4. The processor interface contains eight writeback buffers. When the processor encounters a cache miss, it initiates a read request and stores the replaced cache line in a writeback buffer if the replaced line is dirty. The writeback buffers are emptied to the bus only when the processor interface runs out of reads to perform or when all buffers become full. In this way, reads are almost never held up by pending write-backs.

3.12.3.4 MP System Latency Reduction

The POWERpath-2 protocol implements a number of design features to maintain low latency across a large system under heavy load.

- Each processor interface maintains a complete set of duplicate cache tags, which hold state information for each cache line. When a coherent request appears on the bus, the processor interface checks the state in the duplicate tag store. If the state is invalid or shared, a proper response is made on the bus and no request need be sent to the local processor. The duplicate tags not only prevent processors from receiving unnecessary external requests, but also lower the overall read latency, because duplicating tag lookups to remote processors is much faster than issuing external snoop requests to those processors.

- When a read request appears on the bus, memory immediately initiates a fetch of that cache line. Concurrently, any processor interfaces that discover that their duplicate tags indicate an exclusive cache block initiate an intervention request to that processor’s cache. Simultaneously initiating both memory and, if necessary, remote cache accesses guarantees that the response arrives in the shortest amount of time possible, regardless of where the data resides.
- Whenever a read request is satisfied by data from a processor’s cache, the memory system accepts the processor’s read response as if it were a write request. Sharing write-backs causes dirty cache blocks to be cleaned as they move between processor caches. Any processor receiving a read response from another processor’s cache loads the data in a clean state, eliminating the need for additional write-backs of that block.
- The operating system can mark text pages with a special attribute. When a processor encounters a miss to a text page, the processor interface sees the text attribute in the read request. Upon receiving the read response to a text miss, the processor interface always loads the data in the shared state. This scheme eliminates the exclusive state for text pages, and avoids future intervention requests to the text line.

3.13 Synchronization

The POWER CHALLENGE architecture provides hardware support for fast synchronization operations to allow for efficient fine-grain parallel processing. This section explains:

- Semaphores and locks
- Fine-grain parallel dispatch
- Tolerating high bus loads

3.13.1 Semaphores and Locks

Semaphores and locks are normally implemented on POWER CHALLENGE using the load linked and store conditional primitives provided by all MIPS processors. Using these primitives allows construction of complex locks, such as counting semaphores. Standard POWER CHALLENGE software includes an extensive library of MP-safe synchronization functions.

3.13.2 Fine-Grain Parallel Dispatch

Silicon Graphics parallel compilers perform parallel dispatch by having a master processor write a parameter block and then writing a *start bit* in this block, which tells the slave processors to read their parameters and start the parallel computation. This is known as a *fork*. At the conclusion of the parallel computation, all processors must rendezvous at a barrier and wait for all of the threads to signal completion. This is known as a *join*.

All that is required to correctly implement fork and join is a coherent shared-memory system. However, without a special hardware assist, these operations are slow and consume a large amount of bus bandwidth. To perform forks efficiently, POWER CHALLENGE implements *piggyback reads*. To perform joins efficiently, a *synchronization counter* is implemented.

3.13.2.1 Piggyback Reads

If two or more processors issue read requests for the same cache block, the POWERpath-2 bus protocol allows them to piggyback on the read response. This means that even though a single read request is issued to the bus and a single read response is provided, any number of processors may participate in the transaction by accepting the read response as their own and indicating that the cache block should be treated as shared. When a fork occurs, all of the processors simultaneously attempt to read the start bit and parameter block. Since they all do this at once, they can all piggyback on the same read instead of each issuing its own. The fork then requires a single read access time instead of N read access times in the N processor case. Overall bus bandwidth is also conserved, since several requests are serviced by a single transaction.

3.13.2.2 Synchronization Counter

Each processor interface contains a special count register accessible by the user. This register responds to broadcast increment transactions. It is used to acceler-ate processor barrier synchronization. As each process reaches the barrier point, it issues a single broadcast transaction on the address bus, which increments all of the counters in the system. It then begins reading its own counter, waiting for it to reach the number of processors involved in the barrier. Only N bus transactions occur, for N processors. Without the counters, each processor must reread the cache line containing the count each time another processor writes it, and each processor must read the line exclusive in order to write it, thus using approximately N^2 memory transactions.

3.13.3 Tolerating High Bus Loads

In a large system, even the fastest memory bus approaches saturation under heavy loads. In these situations, it is important that the bus handle saturation gracefully. Performance should degrade in a linear fashion versus exponentially, while forward progress must be guaranteed and starvation avoided.

Independent address and data buses, multiple outstanding variable latency reads, and a high-bandwidth memory system allow the POWERpath-2 bus to maintain its peak data transfer rate. Other special features designed to encourage graceful performance under heavy loading are *interleaving* and a *programmable urgent timer.*

- The POWERpath-2 memory system supports interleaving on cache line addresses. Parallel access to SIMMs across a 576-bit wide DRAM bus makes it possible for a single two-way interleaved memory board to supply the full 1.2GB per second bus bandwidth. This occurs, however, only when read addresses alternately target even and odd cache blocks. With additional memory boards, memory interleaving can be increased up to eight-way.

Combining high interleaving with protocol tolerance for out-of-order read responses provides full bandwidth for almost any memory reference pattern. In addition to increasing memory interleaving, up to 16GB of memory can be added, and will perform at full speed even in the presence of single-bit errors due to in-line ECC correction.

- To prevent starvation and guarantee forward progress for all processes, each node is equipped with a programmable urgent timer. When a node cannot issue a request for a specified amount of time, its bus arbiter automatically raises its priority to urgent. Once set to urgent, the node has high-priority access to bus resources, such as memory and I/O.

3.14 Memory Subsystem

POWER CHALLENGE main memory can be configured in sizes ranging from 64MB to 16GB, allowing you to select the appropriate memory size for even the largest applications. The memory system includes many features that improve reliability while maintaining high bandwidth and low latency.

The POWER CHALLENGE memory subsystem consists of one to eight memory modules. Each module contains two leaf controllers, each of which can control up to four banks of memory, for a total of eight banks per module. Memory banks can be populated with either high-density SIMMs in increments of 64MB, or super density SIMMs in increments of 256MB. Depending upon the configur-ation selected, a memory module can therefore contain anywhere from 64MB to 2GB of memory.

Each leaf controller is capable of delivering one half of the POWERpath-2 bus bandwidth, or 600MB per second. A leaf controller can deliver data to the bus with a latency of 252 nanoseconds and a repeat rate of 210 nanoseconds. Memory is interleaved on cache-line (128-byte) boundaries, provided two or more leaves are present. Two leaves are sufficient to sustain full bus bandwidth for sequential accesses; adding more leaves allows the memory system to approach full bandwidth for randomly distributed accesses.

POWER CHALLENGE memory is ECC-protected, and can correct all single-bit and detect all double-bit errors. To diagnose developing problems early, the operating system logs all single-bit errors during system operation. In addition, the memory modules contain built-in self-test (BIST) circuitry, which tests the entire memory array and reports errors whenever the system is reset. The boot firmware automatically disables any memory banks that fail BIST, providing a “fail-soft” capability.

3.15 I/O Subsystem

The POWER CHALLENGE I/O subsystem is designed for large, high-bandwidth I/O configurations, while supporting simple low-bandwidth devices at a low base cost. It consists of one to four POWERchannel-2 boards, together with associated HIO modules and peripherals. Each POWERchannel-2 board contains a 320 MB per second HIO bus, to which various I/O adapters are attached. This bus was originally designed to meet the extremely large bandwidth requirements of ultra-high-performance Silicon Graphics 3-D graphics hardware. Although it might seem excessive for ordinary I/O needs, the HIO bus has proved to be the ideal interconnect for such supercomputer peripherals as HiPPI and disk arrays. Figure 3-21 diagrams I/O system architecture.

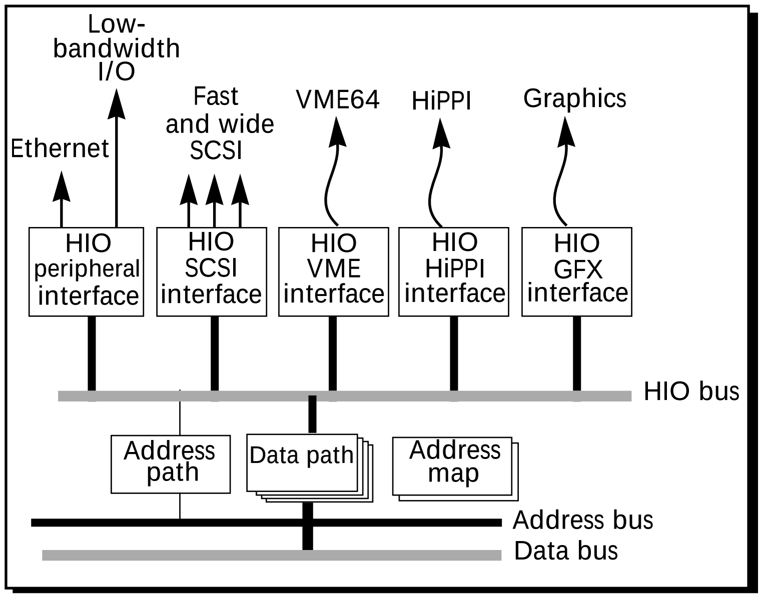


Figure 3-21 I/O System Architecture

All I/O devices connect to the HIO bus through I/O adapters. The POWERchannel-2 board contains I/O adapters to connect a basic I/O complement consisting of an Ethernet transceiver, two 16-bit SCSI-2 channels, three serial ports, and a parallel port. In addition, it includes connectors for a VCAM VME adapter and two connectors for HIO modules for customizing and expanding the I/O system to meet user requirements.

The HIO bus is a 64-bit multiplexed address/data bus running off the same clock as the system bus. It supports split-read transactions, allowing up to four outstanding reads per device.

All DMA I/O devices on the POWERchannel-2 perform their transfers in a virtual address space. This offers the following advantages:

- Large DMAs (greater than the page size) can be performed as a single contiguous operation, dramatically reducing OS overhead and interrupt load.
- I/O can be performed directly into the memory of a user process, without copying through system buffers.
- User-supplied drivers can have memory access restricted to specific address spaces, simplifying driver design and reducing the system’s vulnerability to driver bugs.

3.16 HIO Modules

Each POWERchannel-2 board provides connection slots for two HIO modules, allowing up to 12 HIO modules in a maximally configured system. These cards configure the I/O system to meet individual needs at the lowest possible cost. Currently available HIO modules include:

- *Triple 16-bit SCSI-2 Adapter*, which provides three 20MB per second SCSI-2 channels. Two of these channels are differential; the third may be configured as either single-ended or differential.
- *HiPPI* (occupying one VME slot in addition to the HIO slot): POWER CHALLENGE provides a full-bandwidth HiPPI interface, with a sustainable transfer rate in excess of 90MB per second.
- *Multi-Ethernet*: this card provides eight 10Base-T Ethernet connections.
- *F Interface*: this Silicon Graphics-proprietary interface allows connection of additional VME64 buses or graphics pipes.
- Dual FDDI
- OC3 ATM interface

3.17 VME Bus

POWER CHALLENGE comes standard with a VME64 bus that fully implements revision C.1 of the VME specification. The VME bus allows connection of a wide variety of third-party or customer-supplied peripheral controllers. Direct memory access is available in both directions between the POWERpath-2 and the VME bus, thus supporting boards that act as either VME masters or VME slaves. Using VME64 D64 transfers, DMA bandwidths in excess of 50MB per second can be achieved by VME masters.

For systems requiring more VME slots, up to four additional VME64 buses can be configured. The high bandwidth of the HIO bus allows all VME buses and other I/O adapters to operate at full speed without mutual interference.

3.18 Peripherals

POWER CHALLENGE offers a broad range of peripheral options, including:

- 2GB and 4GB differential SCSI-2 disk drives
- 4mm and 8mm DAT
- SCSI-2 RAID Array
- DLT tape

3.19 References

1. Papamarcos, M., and Patel, J. “A Low Overhead Coherent Solution for Multiprocessors with Private Cache Memories.” In Proceedings of the 11th International Symposium on Computer Architecture. IEEE, NY, 1984, pp. 348-354.
2. Archibald, J., and Baer, J.L., “Cache Coherence Protocols: Evaluation Using Multiprocessor Simulation Model,” ACM Transactions on Computer Systems, Vol. 4, No. 4, Nov 1986, pp. 273-298.
3. Goodman, J. R., “Using Cache Memory to Reduce Processor-Memory Traffic.” In Proceedings of the 10th International Symposium on Computer Architecture. IEEE, NY, 1983, pp. 124-131.

Chapter 4 64-bit IRIX™ Architecture and Standards

IRIX 6.2 is the first desktop-to-datacenter, all-platform 64-bit IRIX release. Based on UNIX System V Release 4.1 and conforming fully to major industry standards, IRIX 6.2 includes multiprocessing scalability, network performance, and valuable desktop and filesystem features not available in competitive UNIX offerings. IRIX 6.2 leads the industry in XPG4 Base 95 Profile Branding.

64-bit IRIX preserves customers' software investments with binary compatibility for 32- or 64-bit applications developed on IRIX 5 or IRIX 6. In addition, developers can recompile 32-bit applications to take full advantage of all the most recent MIPS processors.

IRIX 6.2 is supported on Silicon Graphics' Crimson*, Indy, Indigo R4000, Indigo2, POWER Indigo2, Onyx, POWER Onyx, Challenge, and POWER CHALLENGE systems.

Highlights of the IRIX 6.2 release include:

- Advanced 3D Internet/multimedia user and developer desktop environment
- 1TB virtual memory addressing, 9TB files, and 18TB filesystems on 64-bit platforms
- 32-bit ABI compatibility with IRIX 5 releases (applications can be recompiled for full R4x00/R5000/R8000/R10000 performance)
- Complete software environment for industry-leading graphics workstations and high-performance supercomputer, database, and network servers

4.20 Standards

IRIX 6.2 conforms to the following industry standards:

4.20.1 X/Open

IRIX 6.2 is among the first operating systems from a major systems vendor to receive the X/Open Base 95 Profile branding from the X/Open Company Ltd. All vendors who claim to support the evolution of UNIX to a single, common specification must achieve this milestone.

4.20.2 SVID

IRIX 6.2 provides a substantial subset of the API as described in the System V Interface Definition, Issue 3 (SVID3), the defining document for System V Release 4 (SVR4). Compliance with the SVR4 API is tested using a variety of commercial and public domain test

* IRIX 6.2 does not support Crimson systems with GTX graphics.

packages. One of the most comprehensive of these is the Generic ABI Test Suite (gABI) developed by UNISOFT. The gABI is a comprehensive test of the compliance of the system consisting of more than 7,800 tests.

4.20.3 MIPS ABI

IRIX 6.2 will be the reference platform for the MIPS ABI 1.2.1 and will be compliant with the MIPS ABI 1.1. IRIX 5.3 remains the reference platform for the MIPS ABI 1.1. The MIPS ABI is defined in the Black Book (*MIPS ABI Conformance Guide*), and is determined by the member companies.

Previous releases of IRIX have been selected as the reference platform by the MIPS ABI group. The member companies—all the vendors that offer SVR4 on MIPS processors—agreed that a suitably compiled binary from IRIX would operate correctly on their platforms.

4.20.4 POSIX

IRIX 6.2 meets the standards established by the POSIX 1003.1 Specification. IRIX has been certified as complying with the FIPS 151-1 interpretation of P1003.1 since IRIX 4.0.5, and as complying with the 1990 version, corresponding to FIPS 151-2, since IRIX 5.2.

IRIX 6.2 incorporates SVR4 commands and utilities that are fully compliant with the POSIX 1003.2 Specifications. Compliance was certified by the X/Open Base 95 Profile branding.

IRIX 6.2 provides many of the interfaces specified by the IEEE POSIX 1003.1b-1993 (formerly 1003.4 Draft 14) standard for real time.

IRIX 6.2 contains kernel enhancements for the support of real-time programming and guaranteed interrupt latencies. POSIX 1003.1c threads (*pthreads*) the industry standard for multithreaded programming, are supported as part of the REACT/pro™ product for IRIX 6.2.

4.20.5 Single UNIX® Specification (SPEC 1170)

Silicon Graphics provides most of the Application Programming Interfaces (APIs) defined in the X/Open “Single UNIX Specification” UNIX 95 (formerly SPEC 1170). Silicon Graphics is committed to complying fully with this X/Open standard, which will enable software developers to produce products portable to any version of the UNIX operating system that supports the specification.

4.21 User Interface and Graphics

Silicon Graphics provides a single, advanced user interface across its product line, plus a broad range of standard and optional graphics and visualization development tools.

4.21.1 X11 and Motif™

Leading-edge technology continues to be the focus for the developer user interface libraries for IRIX 6.2. IRIX 6.2 supports Release 6 of the X11 Window System for both client and server and IRIX IM, Silicon Graphics' enhanced version of OSF Motif 1.2.4. The X11R6 libraries and IRIX IM libraries have been updated to support 32-bit and 64-bit addressing. In addition, there is support for CID keyed fonts and the X double buffering extension, and features for font installation and management have been added.

4.21.2 Indigo Magic™ User Environment

From desktop utilities to digital media applications and collaborative tools, Indigo Magic provides everything necessary for both the novice and the experienced user to take advantage of the power of visual computing.

Indigo Magic offers far more benefits than generic common desktops. Compared to CDE, Indigo Magic offers major value in the areas of system administration, network awareness, and multimedia support. Implemented using the visually-enhanced SGI version of the Motif™ toolkit rendered using the X Window System™, this tightly-integrated environment includes:

- Icon catalogs for easy access to software tools
- Window overview for reducing screen clutter
- Multiple desks for switching between tasks
- IRIS Insight for online help and documentation
- Graphical system administration tools
- Network searching for printers, peripherals, systems, and users
- Proactive system monitoring tools
- Bundled multimedia, collaborative, and Web-based tools
- PC compatibility

4.21.3 Bundled Multimedia Products

A spectrum of useful multimedia products are bundled with each system:

- Digital Media Tools

  A tools suite that enriches your communications with all forms of digital media. Tools such as MovieMaker, SoundEditor, and ImageWorks make manipulating movies, audio, and images as natural as working with text. This package of capture, edit, playback, and translation tools supports many formats, including:
  - QuickTime™ for movies
  - AIFF for audio
  - GIF, TIFF, and JPEG for images
- MediaMail

  A comprehensive SMTP/MIME-compliant electronic mail package that helps you send and receive multimedia messages and manage your mail efficiently.

- IRIS Showcase™

  Create digital media-rich, interactive presentations using audio, video, 3D models, and more. Showcase also exports PostScript™, EPS, Inventor, and VRML 3D.

- InPerson™

  Real-time multiway conferencing from your desktop with interactive video, audio, 3D models, images, text, and more.

- Mindshare™ OutBox

  The first drag-and-drop Web publishing tool that enables every desktop to become a Web-based collaboration system in a heterogeneous environment. Drag and drop files to share them with any platform that supports Web browsing...PC, Macintosh, Sun, or other.

- WebMagic™

  A powerful WYSIWYG authoring tool for creating multi-datatype World Wide Web pages that eliminates the need to learn HTML.

- Netscape Navigator™

  The Web's leading browser for global information retrieval.

- WebSpace™ Navigator

  Browse 3D environments on the Web with the industry's first commercially available VRML browser.

- Adobe Acrobat™ Reader Version 2.1

  View information in Adobe's Portable Document Format (PDF), an emerging industry standard for electronic exchange of documents.

- Insignia SoftWindows™

  Run "off the shelf" DOS and Windows PC software. SoftWindow is an X Windows application for UNIX that emulates an Intel personal computer in software.

4.21.4 OpenGL®

OpenGL is an industry-standard graphics development environment embraced by more than 30 companies, including the major leaders in the PC and work-stations industries. OpenGL is a cross-platform, portable API that enables 2D, 3D, and imaging applications to be developed once for deployment to a variety of hardware platforms with different operating systems and windowing environments. All OpenGL platforms conform to the OpenGL standard, insuring application portability across heterogeneous platforms.

OpenGL includes operations for:

- Geometric and raster primitives
- Viewing and modeling transformations
- Lighting
- Shading

- Blending
- Fog
- Hidden surface removal
- Texture mapping

OpenGL gives developers a consistent graphical interface for each platform.

For Silicon Graphics workstations, OpenGL is bundled with IRIX 6.2. Other OpenGL licensees include:

- 3Dlabs
- AccelGraphcs
- Cray Research
- Digital Equipment Corporation
- Evans & Sutherland
- Harris Computer
- Hitachi
- IBM
- Intel
- Intergraph
- Microsoft
- NCD
- NEC
- Portable Graphics
- Samsung
- Sony
- SunSoft
- Template Graphics

4.21.5 Open Inventor™

Open Inventor is an object-oriented 3D graphics toolkit offering a comprehensive solution to interactive graphics programming problems. It presents a programming model based on a 3D scene database that dramatically simplifies graphics programming. Open Inventor offers a rich set of objects that speed up your programming time and extend your 3D programming capabilities beyond OpenGL, X11, and Motif. The objects include:

- Cubes
- Polygons

- Text
- Material
- Camera
- Lights
- Track balls
- Handle boxes
- 3D viewers and editors

Open Inventor follows in the footsteps of OpenGL and is on its way to becoming a industry-standard 3D graphics toolkit. Like OpenGL, Open Inventor is a cross-platform, portable API that has been ported to all major PC and workstation platforms. While the Open Inventor developer environment is not bundled with IRIX 6.2, the end-user environment for Open Inventor *is* being distributed today with IRIX 6.2.

4.21.6 ImageVision Library®

The ImageVision Library (IL) is an extensible, layered, object-oriented toolkit for creating, processing, displaying, and managing images on all Silicon Graphics workstations. The core set of robust image processing functions includes:

- Color conversion
- Arithmetic functions
- Radiometric and geometric transforms
- Edge, line, and spot detection

The ImageVision Library was designed to reduce complexity for development of image processing applications. It provides a consistent interface regardless of changes or new releases of hardware or foundation software. The ImageVision Library end-user environment is bundled with IRIX 6.2

4.21.7 Shells

IRIX 6.2 provides the following shells:

- C Shell (*csh: /bin/csh /usr/bin/csh /sbin/csh*)
- Bourne Shell (*sh: /bin/sh /usr/bin/sh /sbin/sh*)
- Korn Shell (*ksh: /bin/ksh /usr/bin/ksh /sbin/ksh*)

All shells are programmable and allow for a tailorable character-user environment.

4.22 filesystems

IRIX 6.2 provides an advanced filesystem with leading-edge features. It also supports a range of alternate filesystems for distributed and standards-based data access.

4.22.1 File Services

The IRIX file subsystem supports multiple physical disks and gives them the appearance of a single, logical filesystem with a hierarchical arrangement. Additionally, XFS filesystems may include plexed and non-plexed components.

IRIX 6.2 uses the Virtual filesystem interface which facilitates the inclusion of several filesystem types into IRIX 6.2, including:

- EFS
- XFS
- Network filesystem (NFS™)
- ISO 9660 (CDFS)
- DOS (floppy only)
- *swap*
- */proc* Filesystem

4.22.2 EFS

Ten years ago, Silicon Graphics replaced the standard UNIX filesystem with EFS, a higher-performance and higher-capacity filesystem needed by the most demanding customers of Silicon Graphics. EFS increases throughput by supporting 64KB extents instead of the 8KB or smaller blocks in some UNIX implementations. EFS also implements 8GB filesystems in 2GB filesystems found on traditional UNIX systems.

IRIX 6.2 still supports EFS for compatibility, but SGI recommends rapid conversion to XFS, one of the most advanced filesystems available from any vendor, and now a standard part of IRIX.

4.22.3 XFS™

Designed from the ground up, XFS is an advanced 64-bit journalled filesystem with integrated volume management and guaranteed rate I/O. XFS supports files as large as 9 million TB (2^{63-1}). With systems that have a 32-bit address space, the filesystem limit is one TB on files and filesystems. filesystems can grow to 18 TB or one TB with a 32-bit kernel, and individual directories have successfully held 67 million files in internal testing. Aggregate read/write performance exceeds 500 MB per second; single-descriptor performance is over 400 MB per second.

The XFS filesystem has a full range of graphical system administration tools to create, delete, mount, export, and modify filesystems.

XFS uses database journalling technology to provide very fast recovery. No filesystem checking utility is needed—the system quickly references a small log of recently updated filesystem transactions after a system crash. Logging filesystems scale as disks and CPUs are added; traditional UNIX filesystem checking utilities do not.

The XFS filesystem supports logical block sizes ranging from 512 bytes to 1GB. Filesystem extents, which provide contiguous data within a file, are configurable at file creation time and are multiples of the filesystem block size. The contiguous data control features of XFS greatly increase I/O throughput because there is no delay due to disk seeks or rotational latency in accessing data stored on disk.

Guaranteed rate I/O makes XFS the only filesystem available that allows applications to reserve specific bandwidth to or from the filesystem.

4.22.4 NFS™

Distributed filesystems simplify the way data is accessed by making remote files appear local. NFS, with an installed base of more than 3 million seats, is the de facto UNIX standard for distributed filesystems.

Until recently, virtually all NFS implementations used the NFS Version 2 protocol (NFSv2). NFSv2 was not designed with today’s high-performance networks or large file sizes in mind.

Silicon Graphics implemented NFS Version 3 (NFSv3) for the first time in IRIX 5.3. The Version 3 protocol also allows 64-bit file addressing, which makes it possible to use NFS to transfer files larger than 2GB over the network for the first time. Since V2 compatibility is preserved in ONC+ (in a more efficient implementation) customers get the best of both worlds: the performance of NFSv3 with the backward-compatibility of NFSv2.

To enhance file-sharing performance and reduce network load, NFS clients can use the Cache filesystem (CFS) function, retaining a copy of frequently accessed remote files locally and updating the cached copy only when the original has been changed.

The Network Information System (NIS) is provided for centralized filesystem management. The automounter service automatically mounts and unmounts NFS filesystems. The NFS locking service, *lockd*, allows locks to be used with remotely mounted files.

4.22.5 ISO 9660

ISO 9660 is a filesystem type used to mount CD-ROM discs in the High Sierra or ISO 9660 formats (with or without the Rock Ridge extensions) formats.

4.22.6 DOS and Macintosh Floppies

The DOS filesystem driver supports 5 1/4” floppy drives in three standard formats when used with the freestanding SCSI floppy drive. The standard single- and double-density, dual-sided 3 1/2”drive and the 3 1/2” 20.1MB floptical drive are also supported, as are HFS/MAC formats.

4.22.7 swap

The swap allows either a file or block device to be used as a swap resource.

4.22.8 /proc

The */proc* filesystem allows running processes to be accessed and manipulated as files by ordinary system calls, such as *open, close, read, write, seek,* and *ioctl*.

4.23 Volume Management

The XFS volume manager *xlv* is a superset of the *lv* volume manager. It supports striping, concatenation, and, optionally, disk mirroring. The system administrator can make online dynamic changes to volumes, including extending the size of a mounted filesystem, with a full set of graphical system administration tools.

4.24 Networking Standards

IRIX 6.2 implements the following Internet RFC (Request for Comment) and non-RFC standards, among others:

Table 8: RFC Standards

RFC	Protocol	Name
678		Standard File Format
768	UDP	User Datagram Protocol
791	IP	Internet Protocol
792	ICMP	Internet Control Message Protocol
793	TCP	Transmission Control Protocol
821	SMTP	Simple Mail Transfer Protocol
822	MAIL	Format of Electronic Mail Messages
826	ARP	Address Resolution Protocol

Table 8: RFC Standards

RFC	Protocol	Name
854	TELNET	Telnet Protocol
959	FTP	File Transfer Protocol
1014	XDR	External Data Representation
1042	IP-IEEE	Internet Protocol for IEEE 802
1055	SLIP	Serial Line Internet Protocol
1057		Portmapper
1058	RIP	Routing Information Protocol
1084	BOOTP	BOOTP Protocol
1094	Sun-NFS	Network filesystem Protocol
1119	NTP	Network Time Protocol
l122	TCP/IP	Internet Hosts Communication Layers
1123	TCP/IP	Internet Hosts Communications Layers
1156	MIB	Management Information Base
1157	SNMP	SNMP
1213	MIB-II	Management Information Base II
1247	OSPF	Routing protocol (not supported)
1323	TCP-HIPER	Transmission Control Protocol: Large windows, etc.
1532	DHCP	Dynamic Host Configuration Protocol
1533	DHCP	Dynamic Host Configuration Protocol
1534	DHCP	Dynamic Host Configuration Protocol
1541	DHCP	Dynamic Host Configuration Protocol

Table 9: Non-RFC Standards

Non-RFC	Protocol	Name
r-commands		4.3 “r-commands” (rsh, rlogin, rcp, rexec)
ONC/ONC+		Sun ONC+ 1.2 and Sun ONC 4.2 but not NIS+
NFS V3		NFS V3 Protocol Specification dated 2/16/94
HiPPI		HiPPI-PH (X3.183-1991)
HiPPI	HiPPI	-SC (X3.222-1983)
HiPPI	HiPPI	FP (X3.210-1992)
HiPPI	HiPPI	-LE (X3.218-1993)
ATM	UNI 3.0 and 3.1	ATM
DCE		OSF/DCE 1.1 Core Services (DCE 1.1 with DFS in 1H 1996)
AFS		AFS 3.4 via Transarc Corporation (Q1 1996)

4.25 Memory Management

The IRIX virtual memory subsystem is based on SVR4 and supports *mmap*() and *mprotect*(). The region structure has its roots in SVR3 with major modifications to support MP cache flushing and TLB management. Virtual memory is managed with a classic demand-paged model.

IRIX supports low-cost forks, also known as Copy On Write (COW) or lazy replication. Unique pages are created only when the child process writes to a particular page.

The size of swap partitions can be reduced or augmented on a running system. Applications can swap to NFS partitions, to mirrored partitions, and to regular files. Memory management policies and performance can be tuned using variables documented on the systune man page.

On IRIX 6.2 physical memory can be up to 16GB, allowing more tasks and/or larger tasks to be in memory simultaneously. A 64-bit user process can grow to a virtual address space of 2^{40} bytes (1TB) if the physical system has sufficient memory and swap space.

4.26 Scheduling

An IRIX process is a collection of resources and a thread of execution within a Silicon Graphics computer system. The virtual address space of a process, the contents of its user structure and proc table entry, and the values contained in machine registers when the process is running constitute the context of the process.

To support multiple processes, IRIX implements a process-scheduling algorithm that assures a fairly equitable division of processor time among all processes. This algorithm is non-preemptive, that is, the running process cannot be preempted by another process (but can be preempted by the kernel).

The running process can yield to another process “voluntarily,” by making a system call (such as an I/O request) that causes it to sleep, in which case another process is selected to run. The running process can also be preempted by the kernel to handle an exception, in which case the process is rescheduled to resume immediately after the exception handler is finished.

The kernel also enforces limits on the amount of time a process can monopolize the processor (time slicing).

While the process management model discussed represents an apparently equitable resource sharing model, multiple CPUs make the task more challenging. Further complexities are added when the scheduler must handle batch and real-time queues in addition to timesharing users.

The scheduler in IRIX 6.2 is designed to handle all these difficult hardware and application environments individually and in combination (recognizing that the goals of timesharing, batch, and real-time are often contradictory).

There are four major classes of priorities in IRIX 6.2. Two of these priority types have multiple subclasses:

Table 10: Scheduling Priorities

Class	Subclasses
Kernel	
User-Mode Real-time	Frame Scheduler
Timesharing	Gang, General
Background	Gang, General

Kernel tasks have the highest priority, background processes the lowest. Subclasses within a type can have any priority within the range assigned to that type.

Scheduling options include:

- Gang Scheduling
- Deadline Scheduling
- Processor Sets
- Processor Cache Affinity

4.26.1 Gang Scheduling

The user can schedule related processes or threads to run as a group. Threads that communicate with each other using locks or semaphores will be assured of running in parallel, avoiding waiting while the receiving or sending process is swapped in and then associated performance penalty.

4.26.2 Deadline Scheduling

The system administrator can specify that a particular process will receive a specified allocation of CPU time in any recurring period. The primary beneficiaries are applications where constant delivery of a video or audio stream or constant acquisition of real-time data must be assured. The process can block waiting for its time allocation or yield the processor if the required work is completed before its allocation has been used. This implementation is more efficient and easier to use than the real-time timers and signals used in other UNIX implementations.

4.26.3 Processor Sets

The administrator can reserve sets of processors to run specific tasks, partitioning the system into groups of CPUs that run batch jobs, real time, and general timesharing separately. This forces certain classes of users to run applications on a particular set of processors, and can provide assurance to batch users that the optimal number of processors needs to run a parallel batch task will be available.

4.26.4 Processor Cache Affinity

The scheduler can be instructed to consider the likelihood that data a process may have fetched into local cache the last time it was running may still be in cache on that processor. Waiting a few microseconds to run again on the original CPU may improve overall throughput by avoiding the penalty of reloading a new cache.

For more information on these schedule models and options, see the Silicon Graphics white papers, *Processor Segmentation: A Resource Management Facility for Shared Memory Multiprocessors* and *Parallel Throughput Performance of IRIX 5.x.*

4.27 Symmetric Multiprocessing and Multitasking

IRIX 6.2 currently supports up to 36 RISC processors on CHALLENGE systems and 18 on POWER CHALLENGE systems, providing unparalleled performance on compute-intensive programs. Silicon Graphics pioneered multiprocessor RISC systems running UNIX. MP versions of IRIX have been available since 1988, so IRIX 6.2 MP support is based on significantly more mature MP technology than is available from other RISC system vendors. The features of multiprocessors from Silicon Graphics allow demonstrable parallel speedups on many technical and commercial applications, further differentiating IRIX.

IRIX employs sophisticated synchronization techniques that allow multiple CPUs to execute in the kernel simultaneously, resulting in a multithreaded kernel. Within the IRIX kernel are more than 5,000 distinct MP-safe entry points. This allows very fine-grained locking; access to an individual data structure is synchronized rather than access to an entire subsystem, reducing the likelihood that Processor B must wait for Processor A to release a locked resource. As a result, IRIX offers exceptional scalability and performance as processors are added.

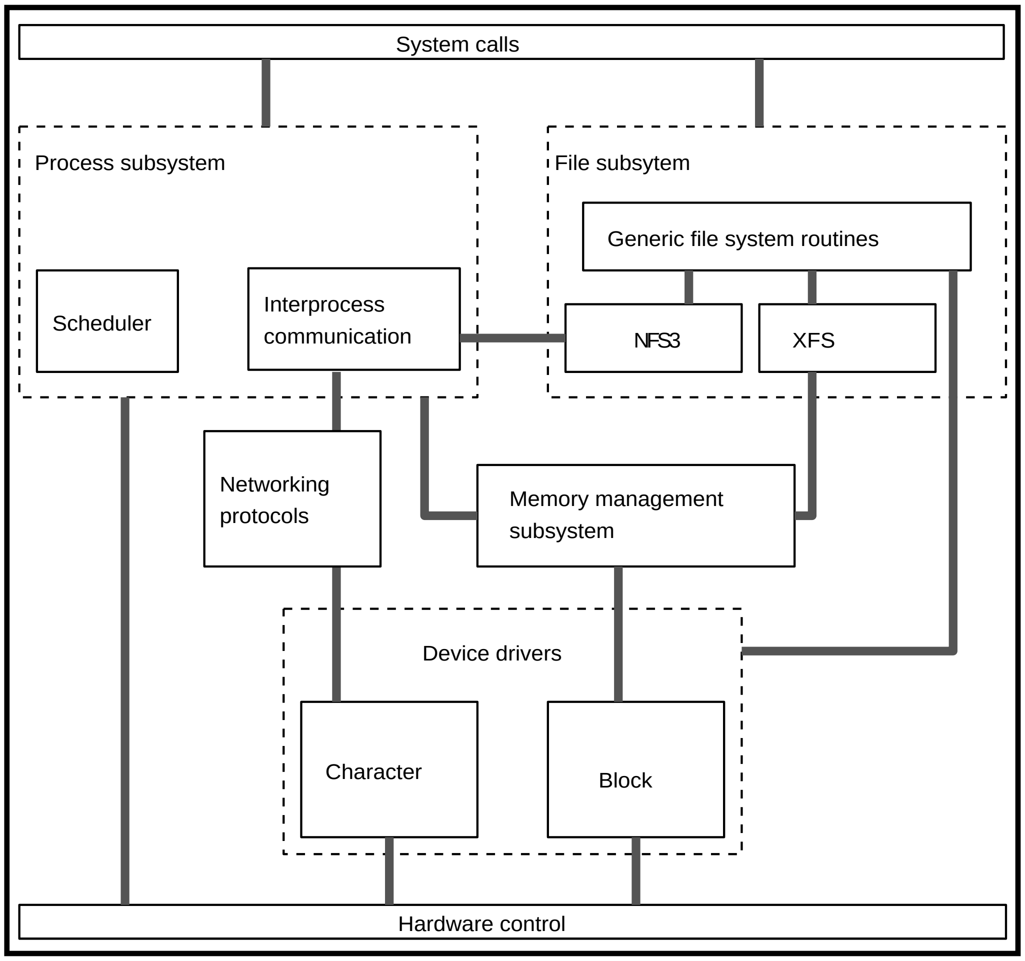


Figure 4-1 IRIX 6 Kernel Architecture

4.27.1 Synchronization Primitives

Silicon Graphics process synchronization primitives, such as spin locks, barriers, and semaphores, are allocated from a shared-memory arena. These are memory-mapped between processes and permit pointers to be shared. Synchro-nization primitives are in the MIPS instructions themselves, allowing for much faster synchronization than in traditional System V mechanisms.

4.27.2 Lightweight Processes

The *sproc* interface permits users to create lightweight processes that share the virtual address space (and potentially other attributes) of the parent process. The parent and child both have their own program counter value and stack pointer, but all text and data space is visible to both processes. This scheme is one of the basic mechanisms upon which parallel programs can be built.

4.28 Enhancements to Support 64-Bit Computing

The IRIS Development Option (IDO) for IRIX 6.2 includes MIPSpro compilers technology to generate either:

- 32-bit code for MIPS 2, MIPS 3 or MIPS 4.
- 64-bit code for MIPS 3 or MIPS 4

Over the last several releases, key components of the operating system have been progressively enhanced to support 64-bit binaries as well as existing 32-bit binaries. These include (but are not limited to):

- Kernel structures
- Include structures
- Extended linking format
- Dynamic shared objects

4.28.1 Kernel Structures

The header files listed in Table 11: contain structure definitions that have been extended to support 64 bits:

Table 11: IRIX System Structures Extended to Support 64 Bits

direct.h	fcntl.h	ftw.h	grp/h
locale.h	math.h	netconfig.h	netdir.h
nl_types.h	poll.h	pwd.h	rpc.h
search.h	setjmp.h	sigaction.h	signal.h
stddef.h	stdio.h	stdib.h	stropts.h
sys/ipc.h	sys/msg.h	sys/procset.h	sys/resource.h
sys/sem.h	sys/shm.h	sys/siginfo.h	sys/stat.h
sys/statvfs.h	sys/time.h	sys/times.h	sys/tiuser.h
termios.h	ucontext.h	sys/uio.h	sys.utime.h
sys.utsname.h			

4.28.2 Include Structures

Include structures have been modified to provide the correct 32- or 64-bit sizes. Since both 32- and 64-bit environments are supported under IRIX 6.2, multiple sets of libraries reside in different directories. 32-bit compilations, targeted at the R4000 and MIPS 2 ISA, are the default. The library path must be changed for other targets.

Each header file under /usr/include will support both 32-bit and 64-bit compilation. The following directory structure is used for IRIX 6.2 libraries:

Table 12: IRIX 6.2 Library Directory Structure[a]

Directories	Contents
/usr/lib/*.so	32-bit DSOs
/usr/lib/*.a	32-bit static libraries
/usr/lib/mips2/*.so	32-bit MIPS 2 DSOs
/usr/lib/mips2/*.a	32-bit MIPS 2 static libraries
/usr/lib/mips2/non-shared/*.a	32-bit MIPS 2 non-shared libraries
/usr/lib32	32-bit MIPS 3 and 4 libraries
/usr/lib64	64-bit libraries and tools
/usr/lib64/*.so	64-bit DSOs (also mip3 and mips4 subdirectories))

a. Any of these directories may have nonshared sublibraries.

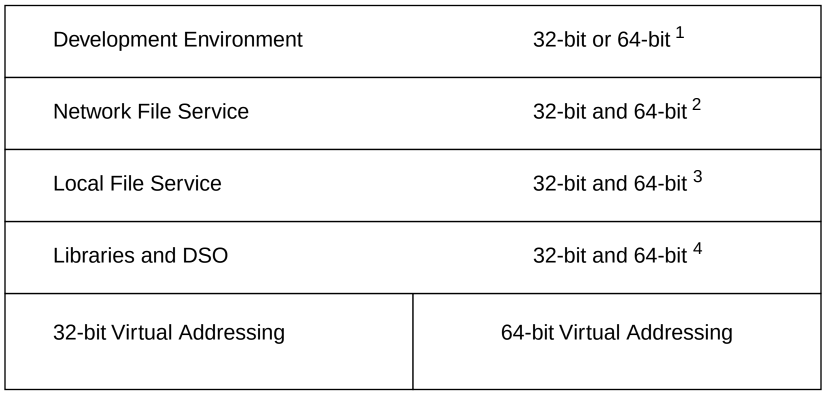
4.28.3 Extended Linking Format

The IRIX 5.x/SVR4 standard Extended Linking Format (ELF) for object modules has also been extended to 64-bits. ELF-64 is supported in all the tools that work with object modules.

4.28.4 Dynamic Shared Objects

The Dynamic Shared Object (DSO) architecture has been extended to support 64-bit addressing. DSO technology is used to provide upward compatibility between IRIX 5 and IRIX 6, and by dynamically binding applications to the appropriate shared libraries at execution time enables a single “shrink-wrapped” version that can operate without change on a wide variety of MIPS hardware platforms.

Figure 4-2 on page 91 summarizes changes for 64-bit and the resulting benefit—both 32-bit and 64-bit kernels can dynamically link 32-bit or 64-bit applications as needed.



1. Develop 64-bit applications anywhere
2. Use >2GB NFS V3 files anywhere while retaining fast access to NFS V2
3. Use large local files to at least 1TB
4. 32-bit IRIX V5 and 64-bit IRIX 6 applications run on the same systems

Figure 4-2 IRIX 6.2 64-bit Enhancements

4.29 IRIX 6.2 Application Binary Options

Developers can target one of three ABIs, depending on their compatibility and performance goals.

4.29.1 O32 for 32-Bit Applications

32-bit applications from IRIX 5 generally “just run” on IRIX 6.2.

4.29.2 64: Full Addressability and MIPS 4 Performance

Applications should be ported to 64-bits if they can benefit from such 64-bit features as:

- Faster logical/arithmetic operations
- Larger virtual address space
- Very large file sizes

Performance can increase on inner loops by a factor of two with 64-bit logical/arithmetic operations. Large models used in applications such as computational fluid dynamics or finite element analysis frequently require greater than 2GB virtual address spaces. Similarly, seismic and reservoir simulation applications in geophysics commonly use data files far larger than 2GB.

4.29.3 n32: Speed without Porting

Applications that already exist were often written with one or more target platforms in mind, and that target was often a 32-bit system. As a result, the developer may have consciously or unconsciously “wired in” 32-bit data structures, pointer lengths, or byte alignments in ways that make a port to 64- bits difficult. ISVs and others may also want to keep memory images small (for example) to run on as wide a size range of platforms as possible.

The n32 Software Developers Kit brings two major benefits:

- Existing 32-bit applications can take full advantage of the performance features of the MIPS 4 ISA without a 64-bit port
- Developers can compile one binary (-n32 -mips3), which runs on all current Silicon Graphics platforms

Developers who simply recompile on IRIX 6.2 with the -n32 -mips4 flags set will generate binaries that use all 32 floating-point registers, eight argument registers, and the MADD instruction of the MIPS 4 ISA. With -n32, fundamental data types do not change: all addressing is 32-bit. Please note, however, that R4x00 systems do not support MIPS 4 ISA.

Developers who compile on IRIX 6.2 with the -n32 -mips3 flags set will have binaries that run faster across the entire current Silicon Graphics product line.

Execution and compilation options for IRIX 6.2 and the platforms they run on are shown in Table 13:

Table 13: IRIX 6.2 Execution/Compilation Table[a]

	o32		n32		o64	
	MIPS I	MIPS II	MIPS III	MIPS IV	MIPS III	MIPS IV
Flags		-32	-n32 -mips3	-n32 -mips4	-64 -mips3	-64 -mips4
Indigo, Indigo^2, Indy, Crimson™						
Execution	✓	✓	✓			
Development		✓	✓	✓	✓	✓
CHALLENGE, Onyx						
Execution	✓	✓	✓		✓	
Development		✓	✓	✓	✓	✓
Power Indigo2™, POWER CHALLENGE, POWER Onyx™						
Execution	✓	✓	✓	✓	✓	✓
Development		✓	✓	✓	✓	✓

a. n32 IRIS GL, n32 OpenGL, and 64 OpenGL are not available on Crimson platforms.

4.30 Development Environment

IRIX 6.2 provides components to help developing applications to run in a Silicon Graphics environment. With all necessary execution-only environments needed to run applications on an IRIX 6.2 system, the development products and development libraries are available as layered components.

The base product offering, IRIS Development Option, (IDO), provides the basic tools and libraries needed for developing IRIX applications. These include:

- C compiler that supports both 32-bit and 64-bit development
- Pertinent code generator
- OpenGL
- X11, Motif, and Digital Media libraries
- Development tools such as *dbx*, *pixie*, and *prof*
- Online reference manuals

For high-performance computational applications, IRIX delivers the highly-optimized CHALLENGEcomplib for access from either FORTRAN or C. IRIX 6.2 provides the additional capability of Power FORTRAN and Power C compiler preprocessors for automatic parallelization of applications for execution on multiprocessor systems.

Additional compiler options are available, including FORTRAN 77, FORTRAN 90, and C++, capable of producing 32 or 64-bit code. IRIX 6.2 provides 32-bit support for Pascal, Ada83, and Ada95, as well as providing support to the additional high functionality libraries, such as Open Inventor, Performer, ImageVision, and ViewKit.

IRIX 6.2 also provides access to the ProDev application tools. ProDev WorkShop, an awarding winning X1- based visual development environment, offers unmatched debugging and performance tuning capabilities, including visual debug tools, graphical visualization of structures and arrays, source code analysis and browsing, test coverage, debugging of parallel and single stream codes, performance analysis, memory leak detection, and build analysis. Tightly integrated into a cohesive development environment, ProDev WorkShop provides support for C, C++, FORTRAN, and Ada95.

RapidApp is a rapid application development tool for creation of media-rich, visual applications for C++. Providing a drag-and-drop mechanism for building applications with objects, RapidApp offers access to X11, Motif, and ViewKit visual components as well as providing access to Open Inventor, and ImageVision Library.

ProDev/Ada provides the most complete development environment for developing Ada95 applications, including extending ProDev WorkShop to understand task based-parallel applications. It provides a validated Ada95 compiler and Ada95 bindings to all SGI libraries.

4.31 Performance Services

IRIX 6.2 includes interfaces that can be used in systems demanding the maximum possible I/O performance. These features require source changes, but when used appropriately, the performance gains can be dramatic. These I/O-enhancing features are:

- Asynchronous I/O
- Memory-mapped I/O
- Direct I/O

4.31.1 Asynchronous I/O

Traditionally, UNIX systems have allowed a given process to have only one I/O operation in progress at a time. This situation forced applications such as database servers that required several concurrent I/O operations to adopt complex, multiprocess architectures. IRIX 6.2 includes support for multiple concurrent I/O operations within a single process. The interfaces comply with the POSIX 1003.1b-1993 specification, allowing a user to queue read(2) and write(2) requests to a device, and receive an optional queued signal when the request completes. A process can simultaneously queue a number of requests without having to wait for any of them to complete. Since the interface is standards-compliant, applications using it are portable.

4.31.2 Memory-Mapped I/O

IRIX 6.2 supports the SVR4 interfaces for memory-mapped I/O. Using these interfaces, disk files are visible in the address space of the program, and can be accessed without explicit I/O operations. The kernel can then bring in pages as it would for an executable, using the demand paging features of the virtual memory subsystem.

4.31.3 Direct I/O

Direct I/O enables direct transfers between disk and the user address space. It bypasses the kernel buffer cache by using DMA to transfer data directly between the disks and the user address space. Memory-mapped files, by contrast, make the data in the kernel buffer cache directly accessible from the user address space.

Direct I/O requires substantially fewer changes to an existing application than memory-mapped I/O. Using both direct and asynchronous I/O, the program can have complete control of its own I/O operations, yielding performance close to that of raw disk, while retaining the benefits of the filesystem. The interfaces for direct I/O were defined by Silicon Graphics.

4.32 Real Time

IRIX 6.2 includes extensive real-time facilities and interfaces, collectively referred to as REACT™. These allow the configuration of systems that interact with external events in a fast, guaranteed, and deterministic manner. The key elements of support for real time in IRIX 6.2 include:

- Non-degrading (fixed priority) scheduling
- Asynchronous I/O (discussed earlier)
- Deadline scheduling (discussed earlier)
- Facilities for memory page locking, to insure that deterministic behavior is never disrupted by a page fault
- Bounded, guaranteed interrupt latency on a properly configured multiprocessor system
- Processor isolation designates one or more CPUs to handle all kernel overhead

- Processor restriction allows users to specify which processors run on which real-time CPUs and to exclude all others
- Guaranteed rate I/O from the XFS filesystem
- Queued signals delivered in priority order
- Real-time processors can be exempted from receiving system clock interrupts and running the IRIX round-robin schedule

These facilities allow IRIX, with its rich feature set and comprehensive applications support, to be used in the most demanding real-time environments. For further information on real-time support in IRIX 6.2, refer to the *REACT in IRIX 6.2 Technical Report*.

4.33 Supercomputing Features

IRIX 6.2 offers features to support unbundled POWER CHALLENGEarray functionality, including the Fair Share Scheduler (based on SHARE II™), kernel level check-point/restart (based on Hibernator II™), and extended accounting, all available from Silicon Graphics. Additional solutions are available from third parties.

4.33.1 Fair Share Scheduler

The Fair Share Scheduler allows system administrators to create a custom compute-resource allocation policy and apply it to an IRIX-based computing environment. Administrators can use Fair Share Scheduling to allocate CPU time, disk space, system memory, connect time, printer pages, and other resources to those users or departments who most need them.

4.33.2 Checkpoint–Restart

Hibernator II is a kernel-level job execution management tool. It allows system administrators to suspend jobs in mid-execution, restarting them later on the same system or a different machine of the same architecture. Checkpoint– Restart allows processes to continue across an administrative shutdown or system crash, and can also be used to load balance applications across a number of Silicon Graphics systems with compatible architectures (IP19 and IP21) and software environments.

4.33.3 Extended Accounting

IRIX accounting has traditionally been process-oriented. Once activated, information logged for each user login included programs run, CPU and wall clock time, and character and block I/O resources used. IRIX Extended Accounting adds to these capabilities via projects and array sessions. “Projects” allows users to bill multiple projects separately. “Array sessions” group processes together across the nodes of a PC array under a single identifier for unified accounting, job control, and more. Both forms of extended accounting also add monitoring of data commonly tracked in a supercomputing environment, such as swaps, bytes read/written, read/write requests, and time spent waiting on the run queue and for raw and block I/O.

4.33.4 Other Batch Scheduling Options

LSF is a load sharing and distributed batch computing product available on Silicon Graphics platforms from Platform Computing Corporation in Toronto. It supports transparent, heterogenous, remote execution and can interoperate with remote NQS systems.

4.34 Security

IRIX 6.2 incorporates technology originally developed for Trusted IRIX/B for identification, authentication, and auditing. Shadow password support places encrypted user passwords in a file that is inaccessible to non-privileged programs, thwarting brute force attacks on the password data base. Auditing logs security-relevant events, enabling analysis of attempts at unauthorized entry and changes made to system files and configurations. IRIX 6.2 also supports automatic password aging, password quality and reuse controls, and optional login restrictions.

These features are designed to meet the U.S. Department of Defense “Orange Book” C2 level of trust. An optional Trusted IRIX/B 6.2 layered product is available for customers who need a C2 or B1 Level of Trust. Trusted IRIX/B assures a B1 level security environment while providing the features from IRIX 6.2 that are required for compatibility with the other Silicon Graphics products and third-party application software. Multiprocessing, graphics, real time, the X Window System and networking are all included.

4.35 Internationalization (I18n) and Localization (L10n)

The IRIX operating environment supports multibyte character sets and local conventions, and servers as the foundation for the WorldView™ family of language modules. WorldView merges international character sets and local conventions into the Indigo Magic user environment on Silicon Graphics workstations. Users can customize the desktop to recognize the language interface of their choice. Developers can design applications to support multiple localized versions.

Silicon Graphics recognizes that it is essential to provide native support for each language. All documents can be created, edited, and printed in the native language. Many systems and utilities in the Silicon Graphics user interface are localized as well. These include key user interface and system messages. Not only is text localized, but formatting of lists, text, and data—such as date, time, number, and currency—follow native conventions for each language.

European-language and Asian-language modules are available from Silicon Graphics. WorkView/ELM, the European-language module supports:

- Czech for Czech Republic
- Danish for Denmark
- Dutch for Netherlands and Belgium
- English for Australia, Canada, UK, and US
- Finnish for Finland

- French for France, Belgium, Switzerland, and Canada
- German for Germany, Austria, and Switzerland
- Greek for Greece
- Hungarian for Hungary
- Icelandic for Iceland
- Italian for Italy and Switzerland
- Norwegian for Norway
- Polish for Poland
- Portuguese for Portugal and Brazil
- Russian for Russia
- Serbo-Croatian for Former Yugoslavia
- Slovak for Slovakia
- Spanish for Spain, Mexico, and Argentina
- Swedish for Sweden
- Turkish for Turkey

In addition, for French-language and German-language modules, there are localized versions of the Indigo Magic desktop interface, MediaMail, and many elements of online documentation, including IRIS Essentials, IRIX System Messages, and IRIS Showcase.

The Asian-language modules are:

- WorldView/JLM, Japanese Language Module for Japanese input via Romanji-kana and kana-kanji conversion
- WorldView/KLM, Korean Language Module
- WorldView/CLM, Chinese Language Module for traditional and simplified Chinese forms
- WorldView/TLM, Thai Language Module

For all Asian-language modules, the Indigo Magic desktop interface, Showcase, and printing support are localized. MediaMail and IRIX system messages are localized specifically for JLM and CLM.

A new addition for IRIX 6.2 include the *iconv* International Codeset Conversion library.

4.36 System Management

IRIX offers three different levels of administrative tools, aimed at users with various needs and professional experience. Cadmin is a set of simple visual tools for desktop users. IRIXpro is a suite of tools for system administrators. Performance CoPilot is a tool suite for monitoring local and remote network performance.

Silicon Graphics also has developed *swmgr*, a graphical interface to *inst* and *gendist*, to ease software and patch installation, handle installation problems, and provide software version tracking.

A variety of ISVs—including Tivoli, Bull, CosMos/Computer Associates, Legent, and Raxco—sell system and network-management solutions for Silicon Graphics platforms.

4.36.1 Cadmin

Cadmin is a simple visual system administration tool for software, performance, volume, and filesystem management. Users can easily add machines, new users, and simple devices, monitor status, and perform other low-level, routine tasks.

4.36.2 IRIXpro™

IRIXpro is a suite of tools for the professional systems administrator in the technical computing market. IRIXpro provides two applications:

- Propel, *rdist*-based software and file distribution (integrated with *swmgr*)
- ProVision, a centralized SNMP-based system event-monitoring tool

4.36.3 Performance Co-Pilot™

Performance Co-Pilot is a Silicon Graphics tool suite that delivers distributed, integrated performance monitoring and performance management across a range of network platforms, databases, and applications. The product is targeted at performance analysts, developers, and system administrators. Performance Co-Pilot is designed to deliver integrated solutions for Silicon Graphics systems. Optional agents include Oracle, Informix, and POWER CHALLENGEarray.

4.37 Backup

Users and administrators have the option of using traditional tape archiving tools (*cpio*, *tar*, *dump*) or using the graphical backup and restore utility that come standard with our systems. We also offer the IRIX NetWorker product that is a full-featured data management tool for multi-system data backup and recovery. NetWorker allows for fully unattended backups with superior performance and scalability.

4.38 Data Migration

Data Management API (DMIG) allows implementation of hierarchical storage management software with no kernel modifications as well as high-performance dump programs without requiring raw access to the disk and knowledge of filesystem structures.

4.39 Compatibility

Every effort has been made to maintain or expand compatibility between IRIX versions.

4.39.1 COFF Obsoleted

IRIX 5 and IRIX 6 binaries are produced using the SRV4 Executable and Linking Format (ELF) which replaces the ECOFF object format in IRIX 4 and enables Dynamic Shared Objects. Development and execution of COFF (IRIX 4 and earlier) binaries is not supported on IRIX 6.2. A utility to find/identify COFF-dependent binaries is provided with the release and installed on the system. It may be executed before IRIX 6.2 is installed.

4.39.2 Binary Compatibility between IRIX 5.3 and IRIX 6.2

Nearly all binaries built on IRIX 5 can be run under IRIX 6. It may be necessary to recode or recompile to take advantage of IRIX 6 enhancements. There may be rare cases—such as applications that examine internal operating system data structures—in which application code must be recompiled.

4.39.3 Object Compatibility between IRIX 6.1 and IRIX 6.2

Except for n32 C++, user object code is compatible between IRIX 6.1 and IRIX 6.2; however, device drivers and any other applications that access the hardware directly must be recompiled.

Chapter 5 MIPSpro Compiler Technology

This chapter consists of three sections:

- The MIPSpro Compiler
- Optimization technology in the MIPSpro compilation system
- Porting Cray code

The chapter concludes with a brief reference section.

5.40 The MIPSpro™ Compiler

The MIPSpro compilers for IRIX 6.2 are the fourth-generation family of optimizing and parallelizing compilers from Silicon Graphics. The compilers generate 32- and 64-bit optimized code for the new MIPS R10000 and MIPS R8000-based POWER CHALLENGE systems running IRIX 6.2, the 64-bit operating system. Supported Instruction Set Architectures (ISA) include both the MIPS III and MIPS IV Instruction Set Architecture (ISA) and full complaint with the new 32-bit (n32) and new 64-bit (n64) ABI. Compiler highlights include

- Support for FORTRAN 90, FORTRAN 77, C, and C++.
- State-of-the-art optimization and automatic parallelization technology
- New enhancements for Inter-Procedural Analysis (IPA)
- Comprehensive support for parallel application development
- Complete support for 32- and 64-bit development and execution

The MIPSpro compilers for IRIX 6.2 take advantage of all the performance oriented features of the MIPS R10000, MIPS R8000, MIPS R4400 and MIPS4000 microprocessors, including:

- Full access to all the hardware features
- Improved calling convention
- Usage of all 32 64-bit floating-point registers
- Usage of all 32 64-bit general-purpose registers
- DWARF debugging format

The MIPSpro compilers will not support MIPS I and MIPS II ISA for old 32-bit applications that are compliant with the old 32-bit ABI (o32). The compilers can be installed on IRIX 6.2-based Silicon Graphics workstation or server systems including POWER CHALLENGEarray, POWER CHALLENGE, POWER Onyx, Onyx, CHALLENGE, POWER Indigo^2, Indigo^{2,} and Indy systems.

Figure 5-1 illustrates the various components of the MIPSpro compilation system. Components include the front end and common back ends that perform all scalar and parallel optimizations and transformations. The compilers support value-added extensions to:

- Enrich the basic functionality of the compilers
- Ease the porting of applications written for other platforms
- Boost run-time performance of applications

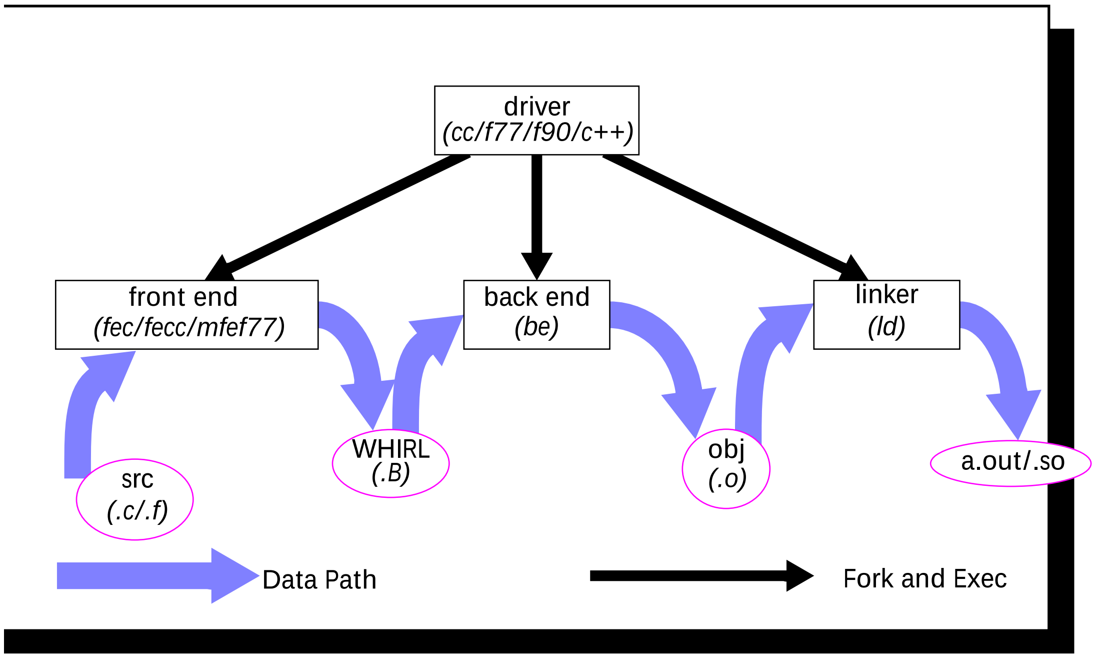


Figure 5-1 MIPSpro Compilation Structure

This section explains:

- Optimizations
- Memory hierarchy optimizations
- Automatic and user-assisted parallelization
- High-performance scientific libraries
- Software development support

5.40.1 Optimizations

Compilers perform a range of general-purpose and architecture-specific optimi-zations to improve application performance by reducing the number of instruc-tions executed. This fully utilizes the CPU’s instruction set, maximizes register use, minimizes memory references, and eliminates unused or redundant code.

New optimization techniques take maximum advantage of the new processor features such as on-chip and off-chip caches, pipelining, and superscalar chip architecture. The optimizations are applicable to a wide range of scientific and engineering applications and benefit both scalar and parallel performance.

Assorted command-line options can leverage different combinations of optimi-zations. In general, optimizations are spread across the compilation system for better efficiency. For instance, high-level optimizations like loop interchange and loop unrolling are performed in the compiler front ends, whereas architec-ture-specific optimizations like software pipelining and automatic blocking are done in the common back end. All optimizations are fine-tuned to take advan-tage of the new system. Key optimizations:

- Architecture-specific optimizations
  - software pipelining
  - instruction scheduling
  - automatic blocking
  - register blocking
  - array padding
  - global instruction distribution
- Statement level optimizations
  - array expansion
  - common subexpression elimination
  - global constant propagation
  - dead code elimination
  - Global Code Motion (GCM)
- Loop-level optimizations, including combinations of:
  - loop unrolling
  - loop interchange
  - unroll-and-jam
  - loop distribution
  - loop fusion
  - loop invariant code motion
  - sum reduction

- Procedure-level optimizations
    - procedure inlining
    - interprocedural analysis (IPA)

5.40.2 Memory Hierarchy (Cache) Optimizations

Memory hierarchy optimizations play a key role in matching the performance capabilities of the fast superscalar processor with the relatively slower main memory system. The primary function of the cache subsystem is to bridge the gap between processor and main memory speed. In this sense, the cache architecture of a modern superscalar microprocessor like the MIPS R10000 or MIPS R8000 is similar to the vector register sets of traditional vector supercomputers that used vector registers to keep the processors busy with computation without having to reference main memory frequently.

Caches are based on the observation that most application programs exhibit some degree of locality of reference: Programs access pieces of data that are “near” already requested data in space and time. A program that accesses memory without regard to locality of reference might perform poorly because of a large number of cache misses. The compiler plays a crucial role in restructuring programs to reduce cache misses by interchanging loops, or by tiling or blocking loop nests so that data is consumed most efficiently by the processor. This is similar to traditional vectorizing compilers that restructured programs to fit in vector memory (registers) in pieces. The compiler restructures programs so that a useful subset of the problem can fit into the cache. Thus, the processor works on patches of the original code and data from the cache memory, avoiding main memory references, before moving on to the next patch.

Figure 5-2 illustrates the cache misses per FLOP as a function of different cache sizes for a broad range of scientific and engineering applications.

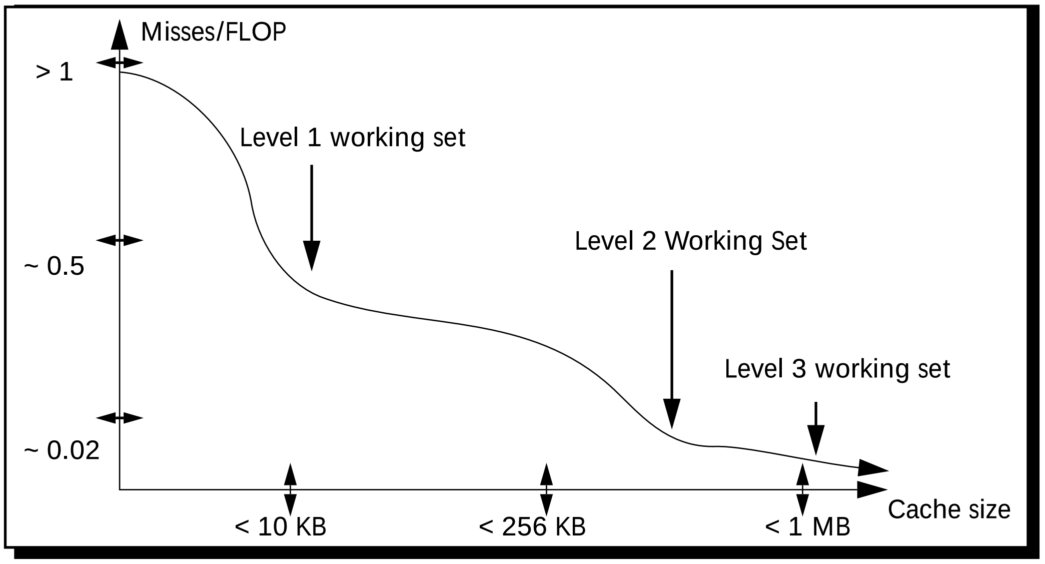


Figure 5-2 How a Working Set Affects Scientific Applications*

For cache sizes of less than 1MB, the miss ratio becomes negligible. This experiment illustrates how a combination of a moderately large cache size and good compiler technology can reduce cache misses to a negligible amount for a large (but not all-inclusive) class of big scientific and engineering problems.

5.40.3 Automatic and User-Assisted Parallelization

The MIPSpro Power compilers (MIPSpro Power FORTRAN 77, MIPSpro Power FORTRAN 90, and MIPSpro Power C) support automatic and user-directed parallelization of FORTRAN and C applications for multiprocessing execution. The compilers employ automatic parallelization techniques to analyze and restructure user applications for parallel execution, as preferred by users who rely on the compilers to parallelize their applications.

The compilers also provide a comprehensive set of standards-based comment directives that enable users to assist the compiler in the parallelization process. Users can use these directives to provide additional information to the compiler to boost parallel performance.

The parallelization technology is fine-tuned to take advantage of the POWER CHALLENGE system architecture. A combination of automatic and user-assisted parallelization can lead to substantial improvements in the performance of many programs.

* From "Working Sets, Cache Sizes, and Node Granularity Issues for Large-scale Multiprocessors" by Jaswinder Pal Singh, Anoop Gupta, and Edward Rothberg, in *Proceedings of the 20th International Symposium on Computer Architecture*

5.40.4 High-Performance Scientific Libraries

The compilers are complemented by CHALLENGEcomplib, a comprehensive collection of scientific and math subroutine libraries that provide support for mathematical and numerical algorithms used in scientific computing. CHALLENGEcomplib is similar to scientific libraries provided by other super-computing vendors like the Cray SCILIB, IBM ESSL and the Convex VECLIB.

The key motivation for creating CHALLENGEcomplib is to provide standard library functionality and to improve the runtime performance of applications. CHALLENGEcomplib is available in sequential and parallel form. The library consists of complib.sgimath that includes support for:

- Basic Linear Algebra Subprograms (BLAS)
- Extended BLAS (Level 2 and Level 3)
- LAPACK, FFT, Convolutions
- Direct Sparse Linear Equation Solvers

5.40.5 Software Development Support

The MIPSpro compiler family includes basic debugging and program runtime analysis tools including *dbx*, *pixie* and *prof*.

- *dbx* is the source level debugger that facilitates debugging of FORTRAN 90, FORTRAN 77, C and C++ code. *dbx* also supports debugging of parallel FORTRAN 90, FORTRAN 77 and C code.
- *prof* is the standard profiling tool that provides “program counter (PC) sampling” of an application’s execution. This information identifies the compute intensive portions of the application and forms the basis for performance tuning of programs.
- *pixie* is also a profiling tool that provides statement-level execution profile by using a technique called basic-block counting. *pixie* provides much finer resolution than prof.

  Note: Both *pixie* and *prof* can be used to profile parallel programs.
- *ProDev Workshop*, a suite of software development tools that includes:
  - Workshop Pro MPF, the parallel program development tool
  - WorkShop Debugger, the parallel debugger
  - WorkShop Performance Analyzer, the parallel program profiling and performance tuning tools
  - Other static and dynamic analysis tools

5.41 Optimization Technology in the MIPSpro Compilation System

Optimization technology is an integral part of the MIPSpro compilation system. Supercomputing microprocessors like the MIPS R10000 or MIPS R8000 offer huge performance potential that needs to be harnessed by advanced optimi-zation techniques. The optimizer considers processor, system, and program characteristics when restructuring programs for performance.

Figure 5-3 illustrates three important parameters that influence the effective-ness of the optimization phases of the compilers.

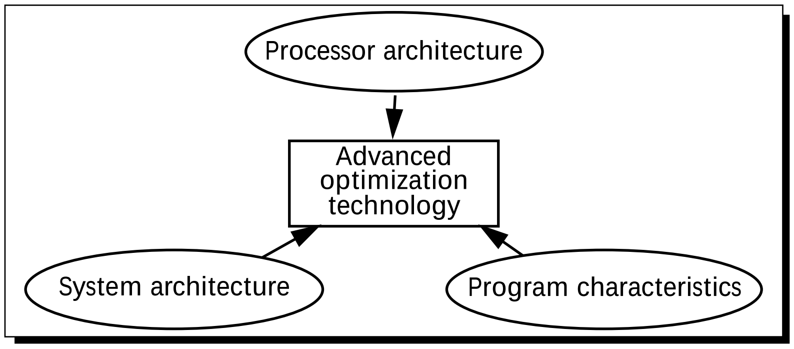


Figure 5-3 Optimization Phase Parameters

This section discusses:

- Processor architecture
- System architecture
- Application characteristics
- Optimization technology
- Basic block optimizations
- Global optimizations
- Advanced optimizations
- Floating-point optimizations
- Global code motion
- Software pipelining
- Pointer optimizations
- Loop optimizations
- Parallelization
- Procedure inlining
- Interprocedural analysis

5.41.1 Processor Architecture

The optimizer has a sound understanding of the architecture and performance characteristics of the MIPS R10000 or MIPS R8000 64-bit superscalar processor and it uses this information in generating optimized code for applications. The MIPS R10000 and MIPS R8000 processors implement the MIPS IV Instruction Set Architecture (ISA).

The MIPS R10000 is a two-way superscalar microprocessor with the following features:

- A large fast-access, high-throughput on-chip instruction and data cache subsystem
- 32 64-bit integer and 32 64-bit floating-point registers, virtual addressing capabilities
- Latency tolerance capabilities including hardware branch-prediction, in-order instruction decoding and out-of-order instruction execution facilities for speculative execution
- ANSI/IEEE-754 standard floating-point coprocessor with imprecise exceptions
- Full compatibility with earlier 32-bit and 64-bit MIPS microprocessors

The MIPS R8000/90MHz is a four-way superscalar microprocessor with the following features:

- A large fast-access, high-throughput cache subsystem specially designed for moving floating-point data
- 64-bit integer and floating-point operations, 32 floating-point registers, and virtual addressing capabilities
- On-chip instruction and data caches
- Branch-prediction capabilities
- ANSI/IEEE-754 standard floating-point coprocessor with imprecise exceptions
- Full compatibility with earlier 32-bit and 64-bit MIPS microprocessors

5.41.2 System Architecture

POWER CHALLENGE system architecture is also considered when restructuring programs for fast execution. For example, the compilers can generate parallel code to take advantage of the multiprocessing capabilities of the POWER CHALLENGE systems. Similarly, the compilation system considers the system bus and memory performance characteristics when implementing efficient synchronization primitives for parallel execution of programs.

POWER CHALLENGE system characteristics important to the optimizer include:

- Shared-memory multiprocessing support with up to 36 MIPS R10000 or 18 MIPS R8000 microprocessors
- 1.2GB system bus
- Up to 16GB of two-way, four-way, or eight-way interleaved main memory

5.41.3 Application Characteristics

The compilers perform extensive analysis of application programs to perform appropriate optimizing transformations. Each application has different charac-teristics that determine the degree of optimizations performable. For example, compute-intensive applications that regularly reference data may be very amen-able to an optimization known as loop blocking. Similarly, applications that ref-erence different sections of data simultaneously can be parallelized by the com-piler for concurrent execution. In short, application characteristics play an im-portant role in determining the degree and effectiveness of the optimizations that can be performed.

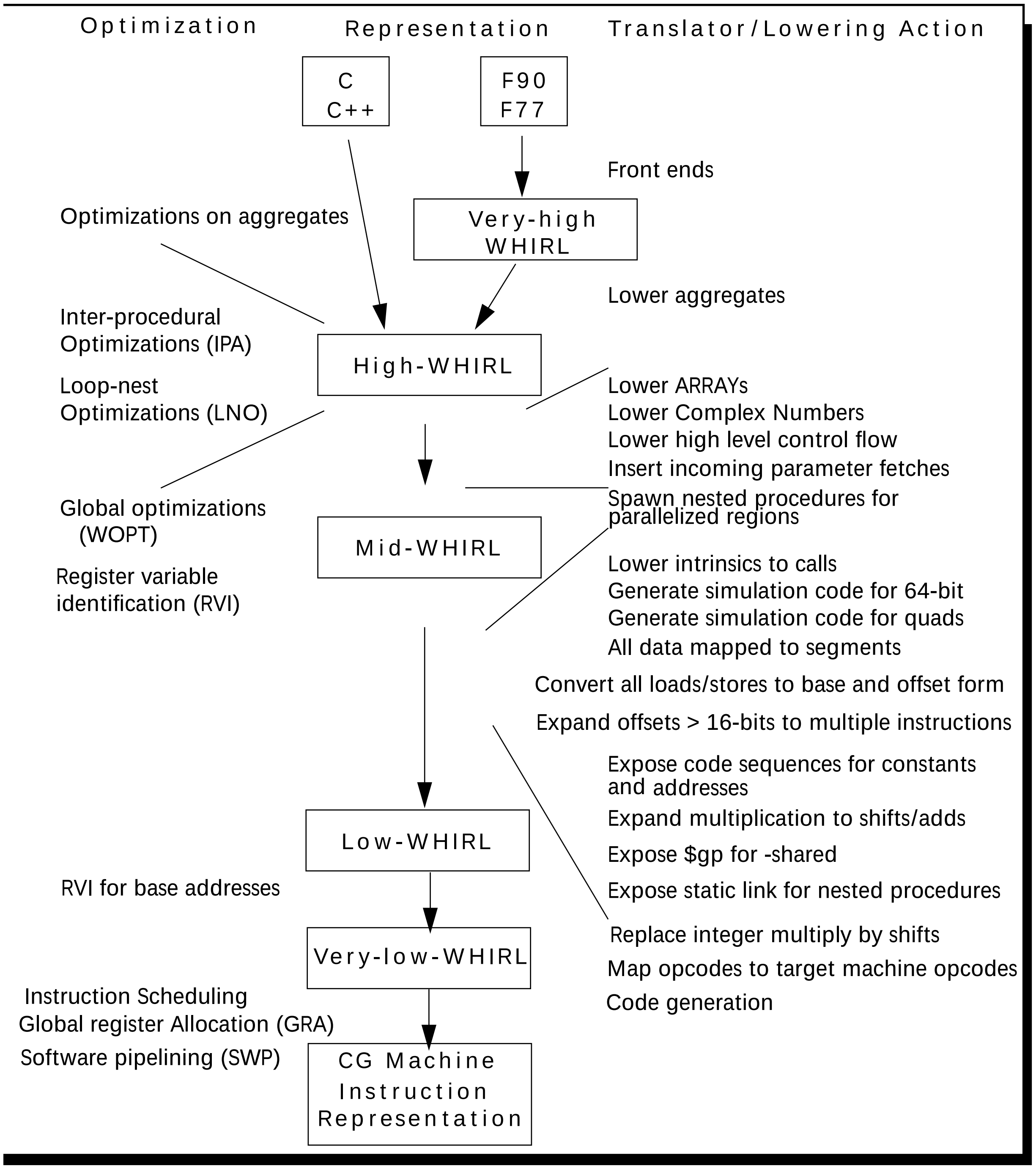
5.41.4 Optimization Technology

MIPSpro compilers perform a hierarchy of optimizations, ranging from fine-grained instruction-level optimizations to coarse-grained parallelization through loops and tasks to reduce application execution time. The optimization phases are spread across the compilation system. Figure 5-4 on page 112 illustrates the different kinds of parallelism exploited by the compilers.

WHIRL, the new intermediate language (IR) for the MIPSpro compilation system, is designed to support C, C++, FORTRAN 77 and FORTRAN 90. WHIRL supports compilation and optimization of program code for MIPS architectures. All high-level and low-level optimizations are performed on WHIRL at the IR level. High-level optimizations are performed in the early stages of the compil-ation process through analysis and transformation of High-Level WHIRL IR. Key high-level optimizations include loop-nest optimizations and interproced-ural analysis. Automatic loop blocking and loop interchange are memory hier-archy optimizations that key into the cache architecture of the machine. Similarly, loop unrolling attempts to expose more instruction level parallelism to the optimizer for fine-grained parallelism.

Instruction-level optimizations are performed mostly in the common back end to get the most performance out of the MIPS R10000 or R8000 superscalar processor. Common instruction-level optimizations include software pipelin-ing, instruction scheduling, global instruction movement and register alloca-tion. Other optimizations such as loop distribution and loop fusion are import-ant for efficient parallel execution. The compiler uses extensive analysis and transformation techniques to detect parallelism in programs. The compilation system supports a complete runtime environment for parallel execution. This runtime library is common to all MIPSpro compilers.

Figure 5-4 Oprtimization Technology in the MIPSpro Compilation System



Optimization
Representation
Translator/Lowering Action
C C++
F90 F77
Front ends
Optimizations on aggregates
Very-high WHIRL
Lower aggregates
Inter-procedural Optimizations (IPA)
High-WHIRL
Loop-nest Optimizations (LNO)
Lower ARRAYs
Lower Complex Numbers
Lower high level control flow
Insert incoming parameter fetches
Spawn nested procedures for parallelized regions
Global optimizations (WOPT)
Mid-WHIRL
Register variable identification (RVI)
Lower intrinsics to calls
Generate simulation code for 64-bit
Generate simulation code for quads
All data mapped to segments
Convert all loads/stores to base and offset form
Expand offsets > 16-bits to multiple instructions
Expose code sequences for constants and addresses
Expand multiplication to shifts/adds
Expose $gp for -shared
Expose static link for nested procedures
Low-WHIRL
RVI for base addresses
Very-low-WHIRL
Replace integer multiply by shifts
Map opcodes to target machine opcodes
Code generation
Instruction Scheduling
Global register Allocation (GRA)
Software pipelining (SWP)
CG Machine Instruction Representation

5.41.5 Basic Block Optimizations

Basic optimizations are performed at optimization level 1 (-O1) and are meant to create efficient scalar code at the basic-block level for both C and FORTRAN programs. A basic-block is a sequence of statements ending with a condition or unconditional branch. Many scalar optimizations are performed at the basic-block level to improve the efficiency of the generated code. The following are some of the most common optimizations performed at the basic-block level:

- Algebraic simplification
- Common-subexpression elimination
- Constant propagation and constant folding
- Dead-code elimination

The compiler also performs the full range of scalar optimizations, including:

- Invariant IF floating
- Loop unrolling and loop rerolling
- Loop fusion and loop peeling
- Array expansion

5.41.6 Global Optimizations

The compilers perform extensive global optimizations at optimization level 2 *(-O2)* that are usually beneficial to most applications, and are conservative in nature to ensure integrity of the results of floating-point computations. The compiler performs memory hierarchy optimizations to maximize reuse of data in the cache. Optimizations performed at this level include:

- Loop blocking and loop interchange
- Global constant propagation to propagate and fold constants across basic blocks
- Control flow optimizations to remove redundant statements, delete unreachable sections of the program, and combine different basic blocks into large basic blocks
- Strength reduction to replace expensive operations with simple ones
- Induction variable simplification
- Fenceposting
- Backward motion of region invariants
- Forward motion of stores
- Collapsing *if* statements
- Global copy propagation
- Arithmetic expression folding
- All -O1 level optimizations

5.41.7 Advanced Optimizations

Compilers perform aggressive optimizations at level 3 (-O3), focusing on best code quality. Great flexibility is provided to enable combination of optimiz-ations at this level for maximum floating-point performance. Important super-scalar optimizations such as software pipelining are performed at this level.

5.41.8 Floating-Point Optimizations

Normally, compilers generate floating-point code that conforms to the IEEE 754 floating-point standard. However, many floating-point-intensive codes that were not written with careful attention to floating-point behavior do not require precise conformance with the source language expression evaluation standards or the IEEE 754 arithmetic standards. It is therefore possible to relax conformance restrictions in favor of better performance. MIPSpro compilers provide a number of different command-line options to accomplish this goal:

- Roundoff options
- IEEE floating-point options
- Reciprocal and reciprocal square-root
- Fast intrinsics

5.41.8.1 Roundoff Option

The -OPT:roundoff=n flag is available to determine the extent to which optimizations are allowed to affect floating-point results, in terms of both accuracy and overflow/underflow behavior.

5.41.8.2 IEEE Option

The *-OPT:IEEE_arithmetic=n* flag specifies the extent to which optimizations should preserve IEEE floating-point arithmetic.

5.41.8.3 Reciprocal and Reciprocal Square Root

The flexible floating-point options provide users with a range of alternatives to trade off accuracy for speed. Thus applications can take advantage of fast MIPS IV instructions like *recip* and *rsqrt.* This is particularly significant for applications that were running on Crays, which have several fewer bits of precision than IEEE 64-bit. Heavy users of Cray and other non-IEEE compliant vector machines who have a need for speed can use these options.

In short, optimizing divides into multiplies by using reciprocal and lifting the inverse calculation outside the loop can give rise to superior performance improvements. For example, if IEEE conformance is required, the generated code must do the *n* loop iterations in order, with a divide and an add in each iteration. Alternatively, if IEEE conformance is not required, the implementation of *x/y* as *x * recip (y)*, or *sqrt(x)* as *x * rsqrt(x)* can be used to treat the *divide* as *a(i) * (1.0/divisor)*. On the MIPS R8000 processor, the reciprocal can be calculated once before the loop is entered, thereby reducing the loop body to a much faster multiply and add per iteration, which can be a single *madd* instruction.

```
INTEGER i,n
REAL sum, divisor, a(n)
sum = 0.0
do i = 1, n
   sum = sum + a (i) / divisor
enddo
```

For example, a loop encountered in the Computational Chemistry Application, WESDYN, developed at Wesleyan University is representative of loops that are frequently encountered in computation-intensive portions of chemistry applications. The loop contains reciprocal as well as square-root operations that are candidates for higher performance.

```
do 200i = 1,n

   r2( i ) = 1 / ( xx( i3 ) ** 2 + xx( i3 + 1 ) ** 2 + xx( i3 + 2 ) ** 2 )
   r1( i ) = sqrt( r2( i ) )
   i3 = i3 + 3

200 continue
```

recip and *rsqrt* are also important to graphics applications that use the reciprocal and reciprocal square root operations in important computational parts.

5.41.8.4 Fast Intrinsics

The MIPSpro compilation system supports a fast version of intrinsic library functions. Selected mathematical functions from the standard mathematical library are hand-coded in assembly language to take maximum advantage of the MIPS IV instruction set of the MIPS R10000 or MIPS R8000 architecture. Specifically, frequently used intrinsics such as the transcendental functions (*log, exp, power, sin, cos, cis and tan)* are hand-coded in assembly and are part of a separate fast mathematical library.

The fast library can be invoked with the -*lfastm* command-line flag. The accuracy level of all the hand-code transcendental functions (except for *tan*) is better than 2 ULPS (units in the least significant place).

5.41.9 Global Code Motion

Global Code Motion (GCM) is a general-purpose optimization that redistributes the instructions among basic blocks along an execution path to improve instruction-level parallelism and make better use of machine resources.

Traditional global optimizers avoid moving instructions in cases that might cause them to be executed along control flow paths where they would not have been in the original program. The MIPSpro global optimizer does perform such code motion, called *speculative code motion*, because the instructions moved are executed based on speculation that they will actually prove useful. This kind of aggressive code motion is unique to the MIPSpro compilers. By default, GCM is very conservative in its speculations. However, a number of options are available to control the degree of speculation.

Figure 5-5 illustrates the different kinds of available speculations .

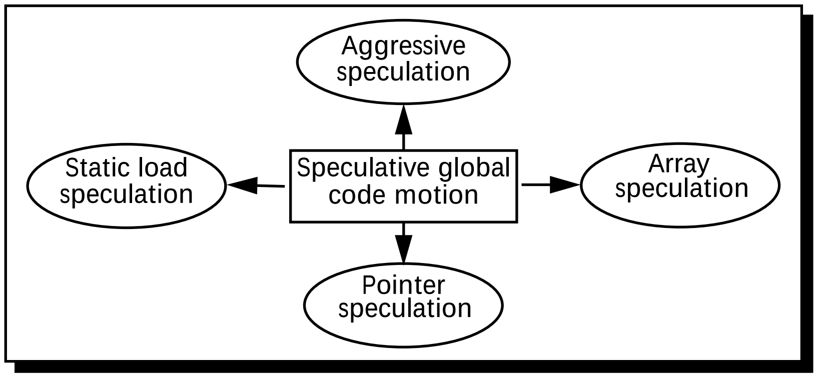


Figure 5-5 Speculation Types

Valid speculative code motion must normally avoid moving operations that might cause runtime traps. As a result, turning off certain traps at runtime enables more code motion. Thus, applications that can ignore *floating-point exceptions* in certain segments of the program can take advantage of this optimization. Similarly, applications that can ignore *memory access exceptions* can also take advantage of this feature. For example, in the *SPECfp92 benchmark 013.spice2g6*, speculative code motion can be enabled for a critical loop by turning off floating-point and memory exceptions around that loop, resulting in a healthy performance gain.

What follows is a brief description of each of these optimizations (with appropriate user-level flag control):

- *Aggressive speculatio*n (-GCM:aggressive_speculation [= (ON | OFF)])
  GCM normally does not move instructions to basic blocks that are already using most of the instruction execution resources available, since doing so will likely extend the execution time of the block. This option minimizes that bias, which often helps floating-point-intensive code.
- *Array speculation (-GCM:array_speculation [= (ON | OFF)])*
  A form of speculation that is often very effective is called bottom loading. It moves instructions from the top of the a loop's body to both the block before the loop (for the first iteration) and to the end of the loop body (which executes them at the end of one iteration so that they will be ready early for the next iteration). Doing this, however, means that the instructions are executed one or more extra times in the last iteration(s) of the loop. If the

instructions moved are loading elements of an array, extra accesses might occur beyond the end of the array. This option permits such out-of-bounds array references by padding the arrays to prevent the out-of-bounds references from causing memory exceptions.

- *Pointer speculation* (-GCM:pointer_speculation [= (ON | OFF)])
  This option allows speculative motion of loads of two kinds, both involving pointer usage. The first allows motion of loads through pointers that may be NULL. The second form moves a reference like **(p + n)*, for a small integer *n*, to a block that already contains a reference to **p*. The assumption is that if *p* is a valid address, *p+n* will be, too. In the example below, the load of *p->next->val* can be moved before the if through a potentially NULL pointer, as can the load of
  p->final_val, which is offset by a small amount from the *p->next* reference.

  ```
  if ( p -> next ! = NULL ) {
  sum = sum + p -> next -> val ;
  } else {
  sum = sum + p -> final_val ;
  }
  ```

- *Static Load Speculation* (-GCM:static_load_speculation [= (ON | OFF)])
  This option allows the speculative motion of loads from static data areas.

5.41.10 Software Pipelining

Software pipelining is arguably the most important architecture-specific optimization implemented in the MIPSpro Compilers. It is a practical, efficient and general-purpose scheduling technique for exploiting fine-grained, instruction level parallelism available in modern-day superscalar processors such as MIPS R10000 and R8000.

In software pipelining, iterations of loops are continuously initiated at constant intervals without having to wait for preceding iterations to complete. That is, multiple iterations, in different stages of computation, are in progress simultaneously. The steady state of this pipeline constitutes the loop body of the object code.

Consider this simple DAXPY loop for execution on a MIPS R8000 microprocessor:

```
do i = 1, n
   v (i) = v (i) + X * w (i)
enddo
```

On the MIPS R8000 architecture, this loop can be coded in assembly language as two *load* instructions followed by a *multiply-add* instruction and a *store.* Figure 5-6 shows this execution order constraint.

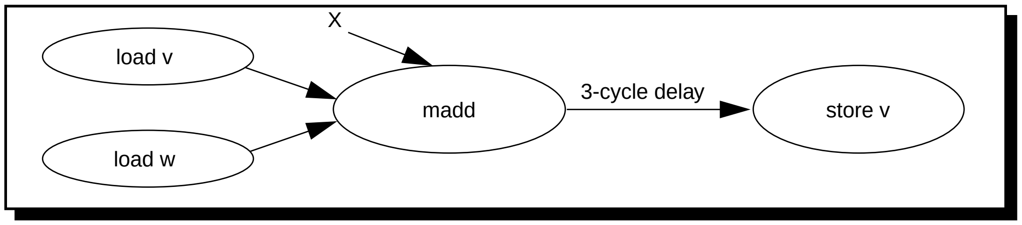


Figure 5-6 Execution Order Constraint for the MIPS R8000 microprocessor

This *simple* schedule completes one iteration of the loop body in five machine cycles. Considering that the MIPS R8000 processor allows up to two memory operations and two floating-point operations in the same cycle, the instructions above are initiated by the machine as outlined in Table 14:. (This schedule completes one iteration of the loop in five machine cycles):

Table 14: Simple Schedule for Loop Body of DAXPY

Cycle Count	Memory Operations	Floating-point Operations
0	load &v(i); load &w(i)	madd X, w(i), v(i)
1		
2		
3		
4	store &v(i)	

If the same loop were unrolled by a factor of four, the loop body would look like:

```
do i = 1, n, 4

   v (i) = v (i) + X * w (i)
   v (i + 1) = v (i + 1) + X * w (i + 1)
   v (i + 2) = v (i + 2) + X * w (i + 2)‘
   v (i + 2) = v (i + 2) + X * w (i + 2)

enddo
```

The unrolled loop allows for instructions from four independent iterations to execute in parallel, thereby increasing the instruction level parallelism and hence the performance of this loop. Table 15: illustrates the sequence of *loads, madds* and *stores* for this unrolled loop. (This schedule completes four iterations of the loop in eight machine cycles.)

.

Table 15: Schedule for Loop Body of Four-Way Unrolled DAXPY

Cycle Count	Memory Operations	Floating Point Operations
0	load &v (i); load &w (i)	madd X, w(i), v(i)
1	load &v (i + 1); load &w (i + 1)	madd X, w (i+1), v (i+1)
2	load &v (i + 2); load &w (i + 2)	madd X, w (i+2), v (i+2)
3	load &v (i + 3); load &w (i + 3)	madd X, w (i+3), v (i+3)
4	store &v (i)	
5	store &v (i + 1)	
6	store &v (i + 2)	
7	store &v (i + 3)	

The schedule above completes four iterations of the loop body in eight cycles, thereby improving the performance to one quarter of the peak MFLOPS of R8000. However, this schedule still leaves room for improvement. Each store has to wait three cycles for its corresponding *madd* instruction to complete, thereby forcing the four store operations to be initiated in different cycles. In other words, the schedule above does not take advantage of the ability of R8000 to do two stores in one cycle.

By using software pipelining, the loop instructions can be initiated at constant intervals such that *each iteration executes a combination of loads and stores from different iterations*. In the DAXPY example, this can result in a schedule that would complete two iterations every three cycles to realize significant performance improvements over the two previous schedules.

Table 16: illustrates the machine schedule for the software pipelined version of the above loop. To prepare properly for entry into such a loop, a prologue section of code is added that sets up the registers for the first few stores in the main loop. Similarly, to exit the loop properly, an epilog section is added that performs the final stores. Any preparation of registers needed for the epilog is done in the cleanup section of the code. (The main loop completes four different iterations in six machine cycles.)

Table 16: Schedule for Software-pipelined DAXPY.

Cycle Count	Memory Operations	Floating-Point Operations
PROLOG:		
	t1 = load &v (i); t2 = load &w (i)	t7 = madd X, w(i), v(i)
	t4 = load &v (i + 1); t5 = load &w (i + 1)	t8 = madd X, w (i+1), v (i+1)
MAINLOOP:		
0	t1= load &v (i + 2); t2 = load &w (i + 2)	t3 = madd X, w (i+2), v (i+2)
1	t4 = load &v (i + 3); t5 = load &w (i + 3)	t6 = madd X, w (i+3), v (i+3)
2	store t7; store t8	beq CLEANUP
3	t1 = load &v (i +4); t2 = load &w (i + 4)	t7 = madd X, w (i + 4), v (i + 4)
4	t4 = load &v (i + 5); t5 = load &w (i + 5)	t8 = madd X, w (i + 5), v (i + 5)
5	store t3; store t6	bne MAINLOOP;
EPILOG:	store t7; store t8	br ALLDONE
CLEANUP:	t7 = t3; t8 = t6	
	br EPILOG	
ALLDONE:		

In the main loop, the code completes four different iterations in six cycles, which is better than the previous two schedules. Table 17: illustrates the performance improvement in the three cases.

Table 17: DAXPY Speedup Factor for Simple Schedule

Scheduling Type	Performance	Speedup
Simple scheduling	1 iteration in 5 cycles	1.0
4-way unrolling	4 iterations in 8 cycles	2.5
Software pipelining	4 iterations in 6 cycles	3.3

As Table 17: suggests, software pipelining can make a huge difference in the performance of compute-intensive applications. Another advantage of software pipelining is its ability to generate compact code, compared to transformations like loop unrolling that can increase the program size by a noticeable amount. Compact code prevents instruction cache penalties resulting from increased code size. The most important aspect of software pipelining is its ability to generate near-optimal code for loops.

Software pipelining in the MIPSpro compilers is based on the concept of “modulo iteration-interval scheduling”, a technique for software pipelining innermost loops—a proven way to generate code with near-optimal perform-ance. In effect, this technique sets a performance goal for each loop prior to scheduling, then attempts to achieve this goal by taking into account resource constraints and program data reference constraints.

Typical benchmark programs, like the SPECfp92 benchmarks, illustrate substantial improvements in performance when compiled with software pipelining.

5.41.11 Pointer Optimization

For many compiler optimizations, ranging from simply holding a value in a register to the parallel execution of a loop, it is necessary to determine whether two distinct memory references designate distinct objects. If the objects are not distinct, the references are said to be *aliases*. When these references are pointers, there is often not enough information available within a single function or compilation unit to determine whether the two pointers are aliased. Even when enough information is available, the analysis can require substantial amounts of time and space. For example, it could require an analysis of a whole program to determine the possible values of a pointer that is a function parameter.

In short, compilers must normally be conservative in optimizing memory references involving pointers (especially in languages like C), since aliases (i.e. different ways of accessing the same memory location) may be very hard to detect. Consider the following example:

```
float x[100];
float *c;

void f4 ( n , p , q )
int n;
float * p;
float * q;

   for ( i = 0 ; i < n ; i ++ ) {
   p[ i ] = q[ i ] + c[ i ] + x[i] ;
   }
}
```

To be safe, the compiler must assume that the pointer references *p*, *q*, and *c* may all be aliased to each other. This in turn precludes the possibility of aggressive loop optimizations by the optimizer.

MIPSpro compilers alleviate this problem of aliasing by providing users with a number of different options for specifying pointer aliasing information to the compiler. The compiler uses this information to perform aggressive optimiz-ations in the presence of pointers for healthy performance improvements. Table 18: illustrates various user options for improving runtime performance.

Table 18: User-assisted Pointer Optimizations

Flag	Description
-OPT:alias=any	Compiler should assume that any pair of memory references may be aliases unless proven otherwise. This is the default setting in the compiler and reflects a safe assumption by the compiler.

-OPT:alias=typed	Compiler assumes that any pair of memory references that are of distinct types cannot be aliased. For example:
	void dbl (i , f) {
	int * i;
	float * f;
	
	*i = *i + * i ;
	*f = *f + *f ;
	}
	The compiler assumes that *i* and f point to different memory locations as they are of different types. This can result in producing an *overlapped* schedule for the two calculations.

-OPT:alias=unnamed	Compiler can assume that pointers will never point to named objects. In the following example compiler will assume that the pointer *p* cannot point to the object *q*, and will produce an overlapped schedule for the two calculations. This is the default assumption for the pointers implicit in FORTRAN dummy arguments according to the ANSI standard.
	float g;
	void double (p)
	float* p;
	{
	
	g = g * g ;
	*p = *p + *p ;
	
	}

Table 18: User-assisted Pointer Optimizations

Flag	Description

-OPT:alias=restrict	Compiler should assume a very restrictive model of aliasing, where no two pointers ever point to the same memory area. This is a rather restrictive assumption, but when applied for specific well-controlled, valid cases, can produce significantly better code.
	vopid double (p,q)
	int*p;
	int*q;
	{
	*p = *p + *p ;
	*q = *q + *q ;
	
	}

5.41.12 Loop Optimizations

Most compute-intensive programs spend a significant portion of their execution time in loops; the MIPSpro compilers spend a significant portion of their time in optimizing loops in the program. Various techniques optimize the performance of loop, many of them automatically enabled by the compilers at various levels of optimizations. This section explains important loop transformations performed by the MIPSpro compilers.

5.41.12.1 Loop Interchange

Loop interchange is a memory hierarchy optimization that modifies the data access pattern of nested loops to match with the way data is laid out in memory. For example, in a typical FORTRAN 77 loop nest, FORTRAN stores array elements in column-major order (not row-major like most programming languages). Each iteration of the *i* loop steps across contiguous elements of *A*, while each iteration of the *j* loop steps over an entire column of *A*. Assuming that *A* is dimensioned as *A (M, N)*, each iteration of the *j* loop steps across *M* elements of *A*. If *M* is larger than a page size of the machine, each iteration of the *j* loop steps on a new page, thereby exhibiting bad locality. As a result, the program may spend considerable portion of its time moving data between memory and the cache system, and exhibit poor performance.

```
do i = 1, m
  do j = 1, n
      a ( i , j ) = a ( i-1 , j ) + 1.0
  enddo
enddo
```

Original Loop

```
do j = 1, n
  do i = 1, m
      a ( i , j ) = a ( i-1 , j ) + 1.0
  enddo
enddo
```

Interchanged Loop

Figure 5-7 Loop Comparison

This problem of the innermost loop having a large stride is eliminated by interchanging the two loops, as shown in the right-hand example in Figure 5-8. Now the innermost loop runs across contiguous elements, minimizing page faults and cache misses. Depending on the dimensions of the arrays, the transformed loop can exhibit significantly better runtime performance.

Another advantage of loop interchange is its ability to move parallelism to outer levels of a nested loop. Before interchange, the innermost *j* loop can be parallelized by the compiler. However, the amount of work performed within the *j* loop may not be sufficient for efficient parallel execution. Once loop interchange is performed, the parallelism moves to the outer loop thereby increasing the amount of work in the loop. In effect the compiler is able to parallelize a larger region of code for better performance.

5.41.12.2 Loop Distribution

The compiler performs loop distribution to partition a single loop into multiple loops. Loop distribution has the advantage of making a loop's working set better fit the paging structure of the underlying machine. It can also expose more parallelism to the compiler. By distributing the loop into a sequential loop and a parallel loop, the compiler is able to efficiently execute parts of the original loop in parallel. The multiple loops are usually smaller (in body size) compared to the original loop and are more amenable to software pipelining. Figure 5-9 illustrates this transformation:

```
do i = 1 , m
    a ( i ) = b ( i ) + c ( i )
    d ( i ) = d ( i + 1 ) + e ( i )
enddo
```

```
do i = 1 , m
    a ( i ) = b ( i ) + c ( i )


do ii = 1 , m
    d ( ii ) = d ( ii + 1 ) + e ( ii )
enddo
```

Figure 5-8 Loop before (left) and after (right) Distribution

The original loop (left) cannot be parallelized because of the data dependency arising from the reference to array *D*. However, after distribution the first *i* loop can be parallelized and the *ii* loop software pipelined for performance.

5.41.12.3 Loop Fusion

Loop fusion, the inverse of loop distribution, involves "jamming" two originally separate loops into a single loop. Figure 5-10 illustrates this transformation.

```
do i = 1 , m
    a ( i ) = b ( i ) + c ( i )


do j = 1 , m
    d ( j ) = a ( j ) + e ( j )
enddo
```

```
do ij = 1 , m
    a ( ij ) = b ( ij ) + c ( ij )
    d ( ij ) = a ( ij  ) + e ( ij )
enddo
```

Figure 5-9 Loop before (left) and after (right) Fusion

Loop fusion can be used in many cases to combine two loops, each of which utilizes a large portion of the page space of the machine. The fused loop can have a working set that is smaller than the sum of the two individual loops, improving data reuse, and permitting better register allocation. In Figure 5-10, after loop fusion the elements of array *A* are immediately available for use by the second statement in each iteration. The optimizer recognizes the reuse of elements of *A*, and keeps them in registers for the operation.

Loop fusion can also increase the size of loops to improve the efficiency of parallel execution. By combining two small loops into a bigger loop, fusion sets the stage for profitable parallelization of the bigger loop. In Figure 5-10, the two individual loops may be too small to overcome the overheads of parallelization. However, the combined loop after fusion may be large enough to realize performance improvements from parallelization.

5.41.12.4 Loop Blocking

Loop blocking is an effective technique available in the MIPSpro compilers for optimizing the performance of the memory hierarchy for numerical algorithms. The reason for blocking is that entire matrices typically do not fit in the fast data storage (for example, the register file or cache) of the machine. Figure 5-10 shows the change in the memory access pattern as a result of loop blocking.

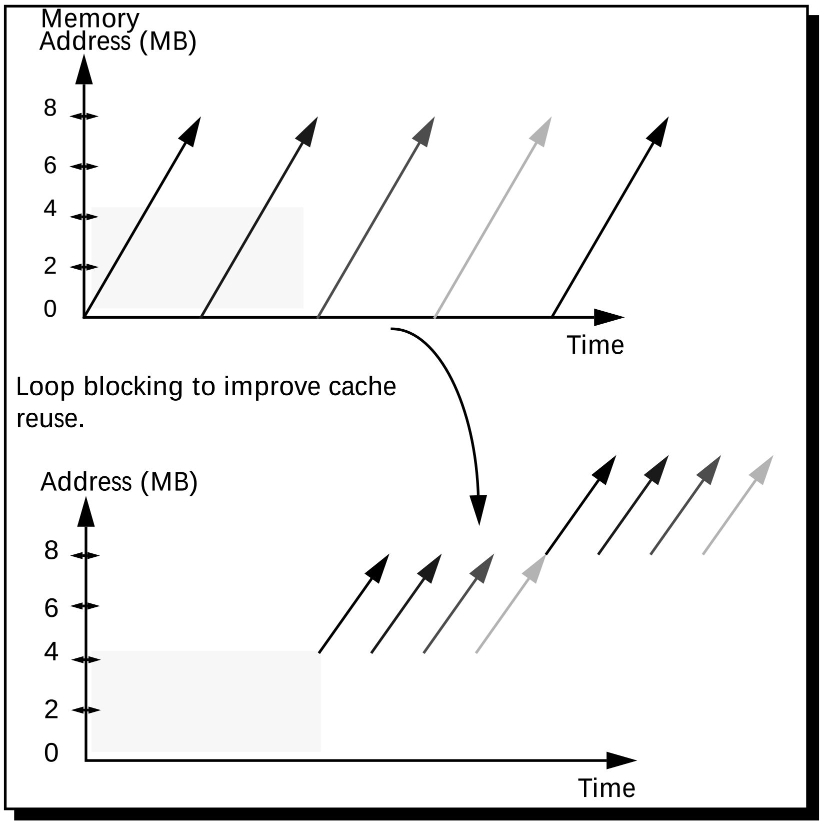


Figure 5-10 Memory Reference Pattern before (top) and after (bottom) Loop Blocking

Blocking decomposes matrix operations into submatrix operations, with a submatrix size chosen so that the operands can fit in the register file or cache. Since elements of a submatrix are reused in matrix operations, this reduces slow memory accesses and speeds up the computation.

The *before* picture in Figure 5-11 references four sets of consecutive addresses over a certain period of time before repeating the access pattern. Blocking restructures the loop to reflect the memory access pattern illustrated in the *after* picture. Here, subsets of all four data sets reside in cache and are accessed in a shorter period of time. This arrangement enables useful computation to be performed efficiently on a cache resident subset of the original dataset before moving on to the next subset. Performance improvements come from reduced processor-to-main memory traffic as a result of efficient cache utilization.

5.41.12.5 Loop Unrolling

Loop unrolling is a fundamental transformation that is a basic component of other restructuring techniques like software pipelining and unroll-and-jam. The unrolling of outer loops of nested loop regions are usually important for good use of the memory hierarchy. Unrolling of inner loops improves the usage of the floating-point registers and provides more room for instruction overlap. Unrolling decreases the trip count of loops, thereby reducing the loop's conditional branch overhead.

The number of times a loop should be unrolled *(unrolling factor)* is determined by the compiler, based on numerous considerations including the amount of data referenced in the loop body, the data access dependencies, the availability of registers, the size of data cache, and the purpose of unrolling. Figure 5-12 illustrates the process of unrolling.

```
do i = 1, n
  v ( i ) = v ( i - 2 ) + X * w ( i )
enddo
```

```
do i = 1 , n , 4

  v ( i ) = v ( i - 2 ) + X * w ( i )
  v ( i + 1 ) = v ( i - 1 ) + X * w ( i + 1 )
  v ( i + 2 ) = v ( i ) + X * w ( i + 2 )
  v ( i + 3 ) = v ( i + 1 ) + X * w ( i + 3 )

enddo
```

Figure 5-11 Loop before (left) and after (right) Unrolling

Here unrolling the loop exposes a lot of instruction level parallelism as the different assignments in the unrolled loop can be overlapped for performance.

5.41.12.6 Loop Multiversioning

Multiversioning is a technique employed by the compiler to improve the efficiency of parallel performance. Many loops, especially in FORTRAN, use symbolic bounds as trip counts which cannot be determined at compile time. However, the compiler can generate multiple versions of the original code at compile time. The resulting program will execute the appropriate path depending on the loops's trip count as determined dynamically at execution time.

```
do i = 1, n
  v ( i ) = v ( i ) + X * w ( i )
enddo
```

```
if ( n > 100 ) then
   /* run this loop in parallel */
     do i = 1 , n
       v (i) = v(i) + X * w(i)
     enddo
 else
  /* run this loop sequentially */
     do i = 1, n
         v(i) = v(i) + X * w(i)
     enddo

 enddo
```

Figure 5-12 Loop before (left) and after (right) Multiversioning

Multiversioning improves the overall efficiency of parallel execution by using accurate information at program execution time.

5.41.12.7 Pattern Matching

The compiler back end understands patterns of standard computational kernels such as SAXPY, DAXPY, LINPACK, and the like, and generates optimal code for such code sequences. Pattern matching is also used for performing basic dependency analysis on loops to improve the effectiveness of software pipelining.

5.41.13 Parallelization

The MIPSpro FORTRAN 90, MIPSpro FORTRAN 77 and MIPSpro C compilers fully support automatic and user-assisted parallelization. User-assisted parallelization is available in the form of comment-based directives to guide program parallelization, which is enabled by specifying the *--mp* flag for both FORTRAN and C.

The directives provide comprehensive support for specifying and controlling the degree and dynamics of parallel execution. For example, the directives can be used to specify conditional parallelism to ensure that parallelism occurs only under certain dynamic conditions. In the two examples shown in Figure 5-14, the *if* clause in the directive specifies the conditions for parallel execution. In the FORTRAN 77 example, the loop executes in parallel only if the value of *jmax* is greater than 1000. Similarly, the C example executes in parallel only if the value of max is greater than 1,000.

```
c$doacross if ( jmax > 1000) share ( pi2, jmax, imax, a, b, c )
c$local ( i, j, x ) lastlocal ( y )
   do 35 j = 1 , jmax
   do 30 i = 1, imax
   x = a ( i , j ) + b ( i , j )
   y = pi2 * x
   c ( i , j ) = y
35 continue
30 continue
C Example of a directive based parallel FORTRAN 77 loop.
```

```
for ( i = 0 ; i < max ; i++ )
      b [ i ] = const * a [ i ] ;
becomes
#pragma parallel if ( max > 1000) shared ( a , b )
#pragma local ( i ) byvalue ( max, const )
{
      #pragma pfor iterate ( i = 0 ; max ; 1 )
                   for ( i = 0 ; i < max ; i++ )
                        b [ i ] = const * a [ i ] ;
}
/* Parallelizing a simple C for loop */
```

Figure 5-13 FORTRAN and C Parallelization Examples

The compilers automatically detect program parallelism by employing the technique of data dependency analysis. Both FORTRAN 77 *(-pfa* flag*)* and C *(-pca* flag*)* compilers have this capability. Data dependence information is also used by a number of loop transformations listed in previous sections of this chapter.

5.41.14 Procedure Inlining

The compilers provide for automatic and user-directed *inlining* of functions and subroutines in FORTRAN and C programs. Inlining is the process of replacing a function reference with the text of the function. This process eliminates function call overhead and improves the effectiveness of numerous scalar and parallel optimizations by *exposing* the relationships between function arguments, returned values, and the surrounding code.

The compilers provide command-line options to direct the inlining of the specified list of subroutines. Flags are available to limit inlining to routines that are referenced in deeply nested loops, where the reduced call overhead or enhanced optimization is multiplied. Options exist to perform interprocedural inlining, whereby instances of routines can be inlined across different files.

One drawback of unlimited inlining is its tendency to increase the code size of the resulting program. Uncontrolled replacement of function or subroutine calls with the actual body of the called routine can cause “code explosion,” which in turn increases compile time and reduces the effectiveness of other optimizations. The technique of Interprocedural Analysis (IPA) provides the benefits of inlining without necessitating inlining the code.

5.41.15 Interprocedural Analysis (IPA)

MIPSpro Compilers for IRIX 6.2 has comprehensive support for IPA as an optional phase of the compilation system. Figure 5-14 illustrates the architecture of the IPA phase of the MIPSpro Compilation system.

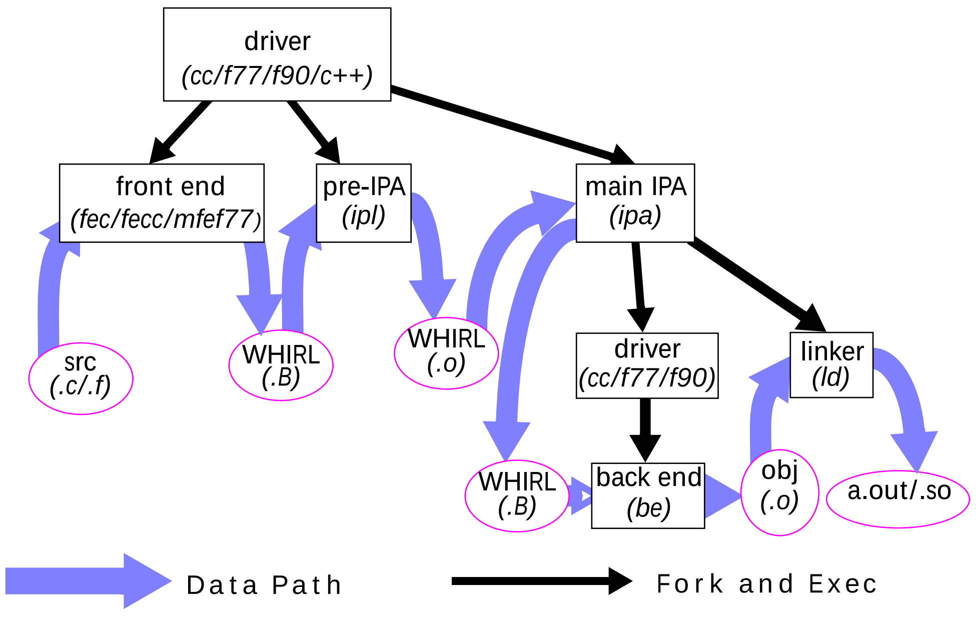


Figure 5-14 IPA Compilation Model

IPA analyzes the interactions between multiple program units (PUs) and passes critical cross-module (global) information to the common back end, which then proceeds to utilize this information when compiling individual PUs one at a time.

IPA tracks the flow of control and data across procedure boundaries and uses this information to drive the optimization process. IPA is particularly useful for performing interprocedural inlining and interprocedural constant propagation, which enables routines with incoming loop bounds information to be available at compile time. This ability can be useful for driving optimization decisions. Figure 5-15 shows an example.

IPA can be turned on by using the *-ipa[=list]* option to specify the degree of IPA to be performed.

```
 n = 1000
...
do i = 1 , 100

  call foo ( a, b, n )

enddo
```

```
subroutine foo ( a , b , n )
real a (*) , b (*)
integer n

do j = 1 , n
  a ( j ) = a (j) + X * b(j)

enddo

end
```

Figure 5-15 Loops with and without Interprocedural Analysis

In this example, the value of *n* is used as a loop bound within the subroutine *foo*. In the absence of IPA, the compiler assumes that the values of *n* is modified inside the call to subroutine foo. Moreover, the value of *n* on entry to subroutine *foo* will not be known at compile-time. As a result, the compiler must generate multiversion code when parallelizing the *j* loop within *foo*.

However, with IPA turned on, it is possible to know (at compile time) the value of *n* on entry to *foo* at the call site. This information is then used by the automatic parallelizer to decide how to parallelize the *j* loop in subroutine *foo*. If the value of *n* is small, the loop may not be profitably parallelized. On the other hand, if the value of *n* is large, the loop gets parallelized for profitable execution.

In short, IPA provides a mechanism to propagate information across procedure boundaries without having to inline calls, thereby increasing the effectiveness of all optimizations.

The main IPA optimization consists of 4 phases. The first phase is a quick pass through all the input files (including relocatable objects and shared objects) to read in all the summary information as well as performing global symbol resolution. The second phase performs the analysis. Based upon the results of the analysis, the third phase performs the optimization and actual code transformations. The fourth phase is optional. It performs recompilation analysis and decides if any already-compiled objects can be reused without recompilation.

5.42 Porting Cray™ Code*

The MIPSpro compilers have a number of value-added features and extensions to facilitate easy migration of existing Cray FORTRAN programs over to Silicon Graphics systems. Important Cray extensions are accepted as-is by the MIPSpro Compilers. A large portion of the remaining Cray features are handled by providing equivalent functionality in the MIPSpro compilation system.

Before examining the degree of Cray compatibility provided by the MIPSpro compilers, it is worthwhile to understand the key features of Cray's compilation system. Cray extensions fall into three categories:

- language extensions:
  - FORTRAN 77 datatype language extensions for high-precision floating-point computations
  - FORTRAN 77 language extensions to support elementary FORTRAN 90 style constructs
- compiler directives to boost scalar and vector performance of applications
- compiler directives to parallelize FORTRAN code on a Cray system

This section describes the Cray extensions in each category and the corresponding capabilities available in the MIPSpro compilers.

5.42.1 Language Extensions

Cray FORTRAN 77 supports the 64-bit programming model for most elementary data types. For instance, basic datatypes like REAL, INTEGER and LOGICAL are 64-bit quantities for Cray. Support also exists for the quadruple-precision COMPLEX data type. Table 19: illustrates the equivalent functionality supported by the MIPSpro compilation system.

* This section is based on work completed by David (Wei) Chen, systems engineer, Silicon Graphics.

Table 19: Cray FORTRAN 77 data type extensions

Cray FORTRAN Extension	Corresponding Silicon Graphics Functionality	Comments
real	real*8	Replace *real* with *real*8* or turn on the command-line option *-r8*, which promotes all *real* data types to 64-bit quantities.
integer	integer*8	Replace *integer* declarations with *integer*8*, or turn on the command-line option *-i8* to promote all integer declarations to become 64-bit quantities. The *-i8* flag provides a convenient way for users to selectively promote integers to be 64-bit quantities. Thus, in the absence of the *-i8* flag or the *integer*8* declaration, integer data type size remains as a 32-bit quantity, thereby preventing integer data cache conflicts that may result from larger integer quantities. Thus, the *-i8* flag provides a much more flexible, performance-oriented approach to manage 64-bit integer data types. Library routines called from user applications should conform to the integer*8 format.
logical	logical*8	Replace *logical* with *logical*8* if a 64-bit quantity is required.
complex	complex*32	Use *complex*32* to get quad precision support.

Cray FORTRAN 77 also supports FORTRAN 90 style indexed array syntax. Table 20: illustrates the equivalent SGI functionality for handling Cray FORTRAN 77 array extensions. The MIPSpro FORTRAN 77 supports the FORTRAN 90 style array syntax. MIPSpro FORTRAN 77 also supports dynamic allocation of arrays.

Table 20: Cray FORTRAN 77 Array Syntax

Cray Program with Indexed and Vector-valued Array Section Selectors	Equivalent Silicon Graphics Program
dimension A(10), B(10), C(5), TEMP(5) dimension X(5, 5), Y(2:10)	dimension A(10), B(10), C(5), TEMP(5) dimension X(5, 5), *Y(2:10)*

A = B	do i = 1, 10
	A(i) = B(i)
	enddo

C = A(3: 7)	do i = 1, 5
	C(i) = A(i + 2)
	enddo

B(1: 5) = X(3, 1: 5)	do i = 1 , 5
	B(i) = X(3, i)
	enddo

A = sin(B)	do i = 1, 10
	A(i) = sin(B (i))
	enddo

TEMP = A(C)	do i = 1 , 5
	TEMP(i) = A(C(i))
	enddo

TEMP = X(1, C)	do i = 1, 5
	TEMP(i) = X(1, C(i))
	enddo

TEMP = A (C + C)	do i = 1 , 5
	TEMP (i) = A (C(i) + C(i))
	enddo

5.42.2 Compiler Directives for Scalar and Vector Performance

Cray FORTRAN supports a few directives to improve the performance of scalar and vector code. Notable vectorization directives are listed in the following table with the equivalent SGI functionality.

Table 21: Cray FORTRAN 77 Vectorization Directives

Cray FORTRAN Vectorization Directive	Silicon Graphics Corresponding Functionality
cdir$ivdep	This directive informs the Cray compiler that there is no data dependency in the loop that follows this directive, thus ensuring complete vectorization of the loop. This directive is accepted as is by the MIPSpro compilers. It tells the compiler to be less strict when deciding whether it can get some sort of parallelism between loop iterations. By default, the compiler makes conservative assumptions about possible memory reference conflicts. The directive allows the compiler to be more aggressive about such assumptions. Thus, superscalar optimizations like software pipelining can benefit from recognizing this directive.
cdir$nextscalar	This directive informs the compiler to generate only scalar (nonparallel) instructions for the loop that immediately follows this directive. The MIPSpro compiler accepts this directive as is and adheres to the original meaning of this directive.

5.42.3 Cray Compiler Parallel-Processing Directives

The Cray compilers support three different techniques to parallelize FORTRAN code. These are:

- *autotasking* (Cray's version of the automatic parallelizer): The automatic parallelization capability of the MIPSpro Power FORTRAN 90 and MIPSpro Power FORTRAN 77 compiler is equivalent to this feature of Cray.
- *microtasking* (Cray's version of user-assisted parallelization): Users can insert comment-based directives in their original FORTRAN 90 or FORTRAN 77 code to specify loop-level and region-level parallelism to the compiler. MIPSpro Power FORTRAN 90 or MIPSpro Power FORTRAN 77 compiler supports a complete set of Parallel Computing Forum (PCF) standards-based comment directives. The PCF directives support loop-level and region-level parallelism and are similar to Cray's microtasking facility. By using the PCF directives users can replace the Cray microtasking directives with equivalent PCF directives to specify the same degree of parallelism.
- *macrotasking* (Cray's version of task-level parallelism): This was Cray's original facility for specifying coarse-grained process-level parallelism. Silicon Graphics compilation system provides a complete user-level threads library for users interested in taking advantage of process-level parallelism. In general, macrotasking is nonportable because of the dependency on vendor-specific libraries.

Table 22: maps Cray’s parallel processing capabilities with their equivalent SGI functionality.

Table 22: Cray Parallel Processing Functionality

Cray’s Parallel Processing Functionality	Silicon Graphics Equivalent Capability

The “*cmic$parallel*” directive is used to declare a parallel region in Cray FORTRAN.
 	
`cmic$ parallel [ if (exp) ]`
`cmic$ shared ( var, ... )`
`cmic$ private ( var, ... )`
Parallel code includes loops with independent iterations, adjacent independent blocks of code, critical sections, or a combination of all these cases.
`cmic$ endparallel`
	The PCF directive “*c*KAP* parallel region*” provides equivalent functionality.
	
	`c*KAP* parallel region [ if (exp) ]`
	`c*KAP* shared ( var, ... )`
	`c*KAP* local ( var, ... )`
	Parallel code includes loops with independent iterations, adjacent independent blocks of code, critical sections, or a combination of all these cases.
	`c*KAP* end parallel region`

The “*cmic$doall*” directive is used for specifying loop-level parallelism in Cray FORTRAN.
 	
`cmic$ doall [ if (exp) ]`
`cmic$ shared ( var, ... )`
`cmic$ private ( var, ... )`
`cmic$ [single |chunksize (n) |numchunks(n)|`
`cmic$ guided | vector] [ savelast ]`
 	
`      do i = n1, n2`
`      ....`
`      enddo`
 	
`cmic$ endparallel`
	This functionality can be replicated by using SGI’s FORTRAN MP “*c$doacross*” directive.
	
	`c$doacross [ if (exp) ]`
	`c$and shared ( var, ... ), local (var, ... )`
	`c$and lastlocal (var, ... )`
	`c$and [ mp_schedtype=type, chunk=n ]`
	
	`      do i = n1, n2`
	`      ....`
	`      enddo`

To summarize, the MIPSpro compilers have sufficient functionality built into them to facilitate smooth migration of Cray code onto SGI platforms. Important Cray directives like *c$dir ivdep* are recognized and processed to improve the performance of superscalar optimizations like software pipelining, while other vector specific directives like *c$dir vector* are essentially ignored by the MIPSpro compilers. In terms of general purpose optimizations, the MIPSpro compilers favor superscalar optimizations like software pipelining and global speculative code motion while the Cray compilers attempt vectorization for best performance.

5.43 References

1. *MIPSpro FORTRAN 90 Programmer's Guide*
2. *MIPSpro POWER FORTRAN 90 Programmer's Guide*
3. *MIPSpro FORTRAN 77 Language Reference Manual*
4. *MIPSpro POWER FORTRAN 77 Programmer's Guide*
5. *MIPSpro POWER C Programmer's Guide*
6. *MIPSpro Compiling, Debugging, and Performance Tuning M*

7. *FORTRAN 90 Handbook*, Walt Brainerd, McGraw-Hill. A complete introduction to ANSI FORTRAN 90 Language and Programming with examples.

8. *Practical Parallel Programming*, Barr E. Bauer, Academic Press, Inc. Presents the practical aspects of parallel programming on Silicon Graphics multiprocessor systems.

9. "Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors," Jaswinder Pal Singh, Anoop Gupta, and Edward Rothberg, in *Proceedings of the 20th International Symposium on Computer Architecture*, IEEE, New York, 1993, pp. 14-25.

Chapter 6 CHALLENGEcomplib: High-Performance Scientific and Math Libraries

CHALLENGEcomplib is a collection of scientific and math subroutine libraries that provide support for mathematical and numerical algorithms used in scientific computing. By incorporating *CHALLENGEcomplib* routines in compute-intensive portions of scientific and engineering applications, users can take advantage of the performance capabilities of the underlying system without having to rewrite their applications. *CHALLENGEcomplib* is comparable to scientific libraries provided by other supercomputing vendors, such as the Cray SCILIB, IBM ESSL, and the Convex VECLIB.

The *complib.sgimath* module, the most important portion of *CHALLENGEcomplib*, includes the following highly-optimized and parallelized routines for MIPS R10000 and R8000 systems:

- Basic Linear Algebra Subprograms (BLAS), level 1, 2, and 3
- 1D, 2D, and 3D Fast Fourier Transforms (FFT)
- Convolutions and correlation routines: LAPACK, LINPACK, and EISPACK
- SOLVERS: *pcg* sparse solvers, *direct* sparse solvers, *symmetric* iterative solvers, and solvers for special linear systems.

CHALLENGEcomplib supports both the MIPS III and MIPS IV Instruction Set Architecture (ISA) and is fully complaint with the new 32-bit (n32) and new 64-bit (n64) ABI. This enables *CHALLENGEcomplib* to take advantage of all the performance-oriented features of the MIPS R10000, R8000, and R4400/4000 microprocessors, including:

- Full access to all the hardware features
- Improved calling convention
- Usage of all 32 64-bit floating-point and all 32 64-bit general-purpose registers

CHALLENGEcomplib also supports MIPS I and MIPS II ISA for old 32-bit applications that are compliant with the old 32-bit ABI (o32).

Chapter 7 ProDev WorkShop: Application Development Tools

Writing programs that run on multiple processors and take maximum advantage of the hardware is a complex and difficult task. Debugging and tuning these programs is even more difficult. To address these problems, Silicon Graphics ProDev WorkShop programming environment is specifically designed to facilitate the development of parallel programs. WorkShop tools to assist the advanced developer include:

- Debugger
- Static Analyzer
- Performance Analyzer
- Build Manager
- Test Coverage Tool
- Pro MPF (optional) and Pro Ada (optional)

7.44 ProDev™ WorkShop

ProDev WorkShop is a complete programming environment with excellent support for parallel program development. WorkShop consists of multiple tools, including the Debugger, the Static Analyzer, and the Performance Analyzer.

- The Debugger is a suite of tools that support the debugging of parallelized programs
- The Static Analyzer analyzes source code and helps developers navigate and visualize their code structure
- The Performance Analyzer provides profiling capabilities for each thread of execution of a parallel program

WorkShop Pro MPF is an optional WorkShop module that cooperates with the MIPSpro Power FORTRAN 90 and MIPSpro Power FORTRAN 77 compiler to facilitate development, tuning, and execution of parallel FORTRAN 90 and FORTRAN 77 programs. Ada programmers will find Pro Ada particularly useful.

7.45 ProDev WorkShop Static Analyzer

The Static Analyzer is a visual source code navigation and analysis tool. It provides the ability to visualize program structure and allows easy navigation through code, which is vital for restructuring and re-engineering existing software. Its graphical presentation of code structure makes it easy to understand, even for someone who is not the original developer. It is helpful in porting situations, when code that is being ported to other platforms will not run or compile. It provides excellent performance on complex FORTRAN programs, and is useful for analyzing legacy code. It provides multiple queries into code structure, such as queries on functions, variables, and common blocks.

7.46 ProDev WorkShop Debugger

The WorkShop debugger is a state-of-the-art, source-level debugger featuring multiple graphical views that are dynamically updated during program execution. It was written from the ground up to support advanced technology, and it is tightly integrated with the performance analyzer, providing increased efficiency for overall program analysis.

- *Enhanced productivity through visualization*
  Tools such as the 3D Array Visualizer and the Structure Browser allow users to identify problems in their code by examining the visual representation of the expressions or data.The WorkShop debugger provides 15 different "views" into a program that are dynamically updated as the user steps through the program.
- *Multiple-process and distributed source-level debugging*
  The WorkShop debugger provides source-level debugging support for programs that have multiple processes or have been parallelized. It permits automatic or manual specification of process groups, and provides individual and group process control. Several processes can be debugged at one time. Traps, break points, and watch points can be set on a single thread or on all threads. The relevant source for each process and any related views, such as variables and expressions, are highlighted at any specified point during the debugging session. The WorkShop debugger is based on a client/server model, allowing distributed debugging.
- *Machine-level debugging*
  Three views provide powerful machine-level debugging capabilities: Register View, Memory View, and Disassembly View. Each view allows the modification of the values that it displays. Register View shows the current register field, register value, which can be modified, and provides a register display area. Memory View allows you to look at and modify memory. The memory display area shows the contents of individual byte addresses. The Disassembly View provides the ability to set break points in the disassembled source, and provides "Continue To" and "Jump To" options for machine-level instructions. Other options include the ability to disassemble a specified number of lines, starting from a specified source line address, from the beginning address of a specified function name, or starting from the address corresponding to a specified file.

7.47 ProDev WorkShop Performance Analyzer

Performance tuning is one of the most difficult programming tasks. Tuning multithreaded applications is even more difficult. The Performance Analyzer is an integrated collection of tools that measure, analyze, and help to improve application performance. Tightly integrated with the WorkShop debugger, it allows the user to visualize a program's performance over separate phases of execution, and correlate the information back to the source code. All of the views show performance statistics on a per thread basis, and provide the ability to correlate the performance of all threads.

- *Task-oriented data collection*
  The Performance Analyzer is designed on a task basis, facilitating the tuning process. Users can choose from a number of performance tasks:
  - Determine bottlenecks, identify phases

- Get Total Time per function and source line
- Get CPU usage per function and source line
- Get Ideal Time per function and source line
- Trace I/O activity, system calls, page faults
- Find memory leaks
- Find floating point exceptions

It also allows the creation of custom tasks, allowing the user to collect data on function counts, basic blocks, and do program counter (PC) profiling. A unique feature of the Performance Analyzer is that it allows the user to specify and collect data during different states of the program's execution through the use of a sampling paradigm. This allows the user to collect information at specified poll point intervals, through the use of sample break points, or interactively with manual sampling. These sample points can then be used to specify phases of program execution. These phases are visually indicated on an experiment timeline. The phases can then be analyzed individually to determine the resource that is causing the bottleneck.

- *Rapid identification of expensive functions*
  Among the multiple views in the Performance Analyzer is the function list, which displays all of the functions in the program, highlights expensive functions both graphically and at the source level, shows the associated performance usage, and suggests more efficient program ordering.
- *Multiple graphical views*
  The Performance Analyzer has an integrated set of graphical views that visually represent performance data. These views can be seen on a per-thread basis.
- *Resource Usage View* to analyze resource usage consumption of the program over different phases of execution. Graphical strip charts are used to display resources such as CPU time, page faults, and context switches.
- Call *Graph View* to display a call graph of the program that is annotated with user-specified profiling information for rapid understanding of program execution sequence and bottlenecks.
- *I/O View* to display read and write activity on a per file-descriptor basis in a strip chart.
- *Heap View/Leak View* to display a color coded map, and listing, of the dynamic memory of the program that clearly identifies memory leaks and erroneous frees, correlated back to the responsible source code.
- *Annotated source and disassembly views* to display relevant source for performance data, annot.ated with performance statistics. Where necessary, the disassembled code is shown instead.

7.48 ProDev WorkShop Pro MPF

ProDev WorkShop Pro MPF provides a powerful visual interface into MIPSpro Power FORTRAN 90 and MIPSpro Power FORTRAN 77 transformations to show which loops were parallelized, which were not, and why they were not. In all cases where a loop cannot be parallelized, WorkShop Pro MPF shows the obstacles to parallelization and allow the user to rearrange the algorithm to circumvent them. Where possible, WorkShop Pro MPF prompts the user for the additional information that will allow the parallel FORTRAN phase to parallelize that code section.

WorkShop Pro MPF also allows control over user-directed MIPSpro Power FORTRAN 90 and MIPSpro Power FORTRAN 77 assertions and directives, as well as parallelization and MP scheduling controls. Integration with the ProDev WorkShop Performance Analyzer allows the user to identify the most expensive loops in the program and concentrate on those, rather than waste effort tuning loops that do not use significant time during execution.

To ease development, debugging, and performance tuning of parallel code, WorkShop Pro MPF:

- Provides detailed listing of all loops in the program with parallelization status
- Allows filtering by loop state: unparallelizable, parallel, serial, or those for which the user has requested modifications
- Allows filtering of information by source file or subroutine
- Shows detailed information about each loop
- Displays source code, highlighting the selected loop
- Lists transformed loops coming from the original loop
- Correlates the original source with the transformed source.
- Shows obstacles to parallelization and other messages
- Shows actual performance cost for the loop
- Displays a list of all subroutines and files in the program
- Highlights obstacles to parallelization
- Shows source lines of where obstacles are in the code
- Shows relevant variable or subroutine names and their uses
- Works with the workshop performance analyzer
- Allows sorting of loops by performance cost
- Annotates source display with performance information
- Allows straightforward composition of a custom *DOACROSS* directive
- Allows editing of parallelization condition
- Shows variables with read/write status within the loop, allows selection of state and highlighting of uses of each variable within the loop