TECHNICAL SUMMARY

OUR BUSINESS AT KENDALL SQUARE RESEARCH IS TOTALLY FOCUSED ON THE OPERATIONAL, FUNCTIONAL, AND ECONOMIC REQUIREMENTS OF OUR CUSTOMERS WHO USE COMPUTING AS A STRATEGIC TOOL. TO THAT END WE HAVE DEVELOPED A NEW FAMILY OF PRODUCTION ORIENTED, HIGHLY PARALLEL, SHARED MEMORY COMPUTERS, UNIQUE IN THEIR ABILITY TO MEET BOTH THE NUMERICALLY-INTENSIVE COMPUTING NEEDS OF TECHNICAL USERS AND THE TRANSACTION AND DATABASE-INTENSIVE REQUIREMENTS OF BUSINESS. THE KSR1 FAMILY OFFERS NEW LEVELS OF PERFORMANCE AT DRAMATICALLY LOWER COST, AND FOR THE FIRST TIME, COMBINES THE SCALABILITY OF A DISTRIBUTED-MEMORY (PARALLEL) ARCHITECTURE, WITH THE PROGRAMMING EASE OF A CLASSIC SHARED-MEMORY COMPUTER.

"WHEN I AM WORKING ON A PROBLEM I NEVER THINK ABOUT BEAUTY. I ONLY THINK HOW TO SOLVE THE PROBLEM. BUT WHEN I HAVE FINISHED, IF THE SOLUTION IS NOT BEAUTIFUL, I KNOW IT IS WRONG."

BUCKMINSTER FULLER

FRONT COVER: FROM THE BEGINNING, MAN'S ART AND CRAFT HAVE BEEN INSEPARABLE AND CONTAINED A BEAUTY BORN OF SIMPLICITY. THE HAND PRINT FOUND IN CAVES AND ON CANYON WALLS ON FIVE CONTINENTS FUNCTIONED AS A SIGNATURE, PASSPORT, AND TERRITORIAL MARKER. THE GEAR, WHICH WAS THE CENTRAL TECHNOLOGY OF THE FIRST WORKING COMPUTER, HAS ULTIMATELY BECOME A UNIVERSAL SYMBOL FOR TECHNOLOGY AND PRODUCTIVITY. NOWHERE IS THE FUSION OF ART AND SCIENCE MORE EVIDENT THAN IN TODAY'S COMPUTERS WHERE THE BEST TECHNICAL SOLUTIONS ARE TERMED "ELEGANT" AND THE MOST RECENT ADVANCE IS CALLED "STATE OF THE ART."

ERRATA

There are several corrections to be made to the Technical Summary.

- On page 28, the first two instances of "GNUMAKE" should change to read "GNU EMACS."
- Page 67 includes a discussion of "quashing." In this discussion, there are two references to "quash_never." The second instance should be "quash_false."

Sorry for any inconvenience this may have caused.

KENDALL SQUARE RESEARCH

KENDALL SQUARE RESEARCH

TECHNICAL SUMMARY

CONTENTS

PART ONE

INTRODUCTION TO THE KSR1

INTRODUCTION

The high performance KSR1 family of general purpose, highly parallel, standards based computer systems runs a broad range of mainstream applications, ranging from numerically intensive computation, to on-line transaction processing (OLTP) and database management and inquiry. The KSR1 differs from conventional mainframes and supercomputers in that it incorporates the scalability and power of highly parallel processing. The KSR1 can be differentiated from the class of massively parallel processors (MPPs) because it uniquely provides the superior performance and ease of use of the shared memory programming model in a scalable, highly parallel computer. In essence, the KSR1 combines the scalability and lower cost of highly parallel processing — across an entire spectrum of computing power, mass storage, and I/O bandwidth — with the higher performance, ease of use and familiarity of the shared memory programming model that has been an industry standard for three decades. The use of shared memory enables a standards based open environment.

THE NEED FOR HIGHER PERFORMANCE COMPUTING

Since the 1970s, the demand for more computer power has increased significantly among large users of numerically intensive, OLTP and database applications. To date, this need has typically been met by mainframes for OLTP and database applications and by vector supercomputers for numerically intensive tasks.

The proliferation and complexity of the data generated by personal computers, workstations, electronic cash registers, and other high speed electronic devices has significantly exceeded the capabilities of the enterprise-wide computing resources of many organizations. For these companies to provide competitive products and services they must find new ways of keeping up with the increasing volume and use of data. The immediate processing of available and requested data requires organizations to move from historical batch processing environments to on-line computing. Current mainframe systems of most corporations have limited power and capability to support this substantial change in requirements and data usage.

In the fields of science and engineering, the demand for more computing power is driven by a greatly increased emphasis on computational modeling to develop and verify engineering solutions and by an increased focus on highly challenging basic and applied scientific problems requiring a seemingly unlimited amount of numerically intensive computation. The basic extension of human knowledge in many scientific areas can only be addressed by the application of high performance computing resources. Examples abound in both basic and applied sciences: innovations in material science (including semiconductor and superconductor design); the mapping of the human genome; pharmaceutical design; speech recognition; and advanced oil and gas recovery.

Visualization is applicable to both commercial and technical applications. The capacity of modern high performance computers to help people "see" or visualize things provides vivid and compelling examples of the increasing need for more computing power, whether the purpose involves intellectual investigation or entertainment. A single earth satellite can produce the requirement to organize, store, and analyze a Terabyte (1 billion bytes) of data each day. The spectacular special effects produced by contemporary movie-makers require computer systems to handle up to 1.8 Gigabytes (1,800,000,000) of visual data per second of finished film footage. The theme parks of the future will require high performance computers to conjure up virtual reality for their audiences.

In all these seemingly disparate applications, the common thread is the growing need for ever-higher levels of computer performance. Progress in the computational science laboratories has been rapid in recent years, but extensions of that progress from the lab to the real world of applications have not yet achieved success on a widespread basis. Several different computer architectures have demonstrated the ability to achieve very high speed and power, but only in narrow and specialized areas.

Large users, ironically, are most often limited in their options. An example may be seen in the area of fourth generation languages (4GL) software development tools. A small firm, whose business makes only limited demands on computer power and speed, can effectively select a 4GL and build a satisfactory application in weeks or months, at a relatively low cost of development and with a high expectation of good results. This option is not open, however, to the very largest companies, whose demands already tax the speed and power of their computer systems. Fourth generation languages are relatively inefficient and become an issue for large users whose systems are already at capacity. The ideal computer system for such large users is one that can provide so much power that the user is no longer constrained by the relative inefficiency of the 4GL and can thus benefit from the reduced costs of application software development and maintenance.

Over the past two decades, system designers have approached the development of high-powered computers in two basic ways.

TRADITIONAL MAINFRAMES AND SUPERCOMPUTERS

Mainframes have typically been the workhorses of the data processing departments of major corporations worldwide. They are primarily used for database intensive and on-line transaction processing applications during business hours and for batch processing large updates to the corporation's database at night. At the high end of the performance spectrum a number of mainframes working concurrently are required to meet the ever expanding needs of the world's largest users (airlines, banks, brokerage, insurance). Before the development of "vectors" in the 1970s, mainframes were also used for scientific computing.

The first supercomputers involved the use of one (or, at most, a few) of the fastest processors that could be obtained by increasing the packing density, minimizing switching times, heavily pipelining the system, and employing vector processing techniques, which apply a small set of program instructions repeatedly to multiple data elements. Vector processing has proven to be highly effective for certain numerically intensive applications, but much less so for more commercial uses such as OLTP or database. With the introduction of vector supercomputers began the somewhat artificial distinction between the uses of supercomputers and mainframes. The vector supercomputers were seen as useful primarily for numerically intensive applications, such as those found in the technical areas of science and engineering. Non-vectorized mainframes were seen as better for commercial applications.

This partly arbitrary distinction, which became conventional wisdom, was further bolstered by the comparative difficulty of programming supercomputers, whose sheer computational speed was achieved at substantial cost. By contrast, sequentially-processing mainframes traded off a portion of their own theoretically maximum computational speed through the adoption of virtual memory techniques to facilitate their programmability.

The development of the concept of virtual memory,[1] a landmark achievement in computer science, provided a way to free programmers from the unnecessary burdens of memory storage and allocation by separating the notion of address from *physical location in the memory.* Its incorporation into the design of mainframes in the early 1970s, can arguably be said to have helped create the computer revolution.

The sheer computational speed of sequential-processing vector supercomputers has led to the development of a considerable body of specialized program code, much of it written in Fortran, but virtually all of it confined to applications in the scientific and engineering communities. To date, traditional supercomputers have found no place in the compute intensive applications of the commercial arenas of OLTP, database management and decision support.

Moreover, even within the confines of science and engineering, incremental improvements in hardware speed and power have proven ever more costly. This class of machine is now widely recognized to be approaching fundamental limits, such as the speed of light, the laws of thermodynamics and architectural constraints on the number of processors.

THE MASSIVELY PARALLEL PROCESSORS (MPPs)

In the 1980s, the first massively parallel processors (MPPs) began to appear, with the single goal of achieving far greater computational power at greatly improved price/performance ratios. The concept behind massively parallel processing is to employ large numbers of low cost processors to provide performance far beyond that of mainframes and supercomputers.

However, in actual practice, MPPs have experienced limited market acceptance. The chief reason for the rapidly diminishing expectations has been that the distributed memory architecture of the MPPs, while very scalable, does not support conventional shared memory programming.

Because of the high cost of developing software for MPPs, most such systems have typically been used for only one or a few applications. Although some large-scale users have developed technical and engineering software for certain applications, the majority of third-party software companies have not found it practical to port applications to these computers.

In addition, MPPs have been basically designed to operate in batch processing mode. The use of such systems to support large networks of interactive terminals, workstations, or single-user desktop computers operating in OLTP or interactive mode has proven to be inefficient and expensive. Even at sites where MPPs have met with some computational success, it is common to find other conventional systems functioning as servers to handle I/O or communications.

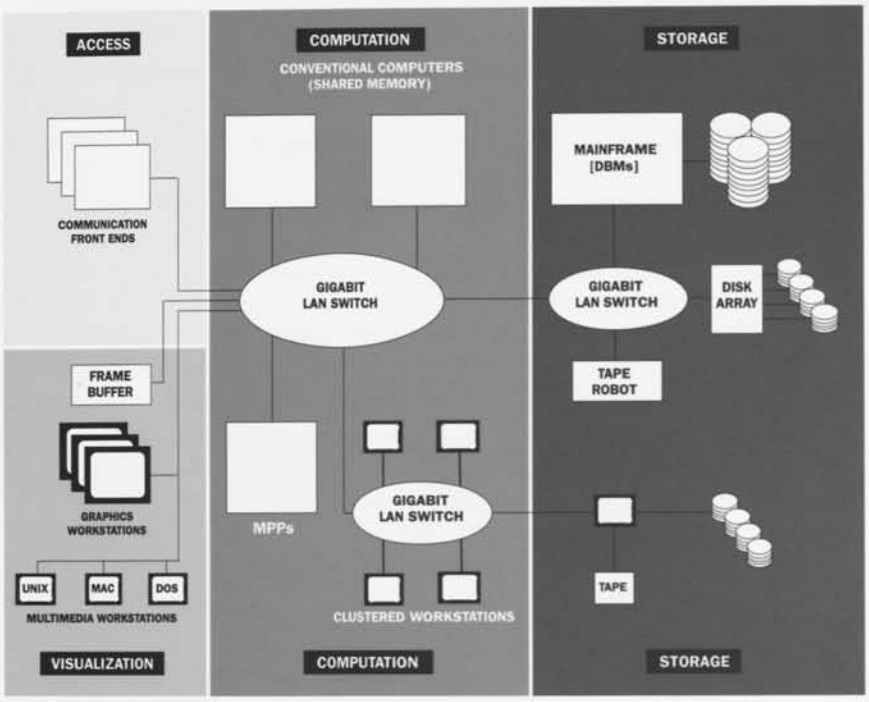
The difficulty of porting existing programs; the absence of a conventional, standards based and familiar software development environment; the lack of widely available third-party application programs; and the restriction in operating modalities — all these factors have combined to restrict acceptance of MPPs. Although a number of approaches have been tried by the various suppliers of MPPs, simple solutions to the seemingly intractable difficulties encountered with MPPs have eluded all but the most dedicated (primarily government) segment of the market.

1. Kilburn, T., Edwards, D.B.G., Lanigan, M.J., and Sumner, F.H. "One-level Storage System;" IRE Transactions, EC-11, Vol. 2, pps. 223-235, April, 1962.

COMPUTING IN THE CONTEMPORARY WORLD

Whether employed for science, engineering, or commercial purposes, most contemporary computer centers resemble, in basic ways, the hypothetical site described in Figure 1. Large computer centers today are characterized by their use of various systems functioning as servers to accomplish some specific purpose within the overall hardware/software assemblage.

FIGURE 1 *Typical Heterogeneous Computer Center*



Contemporary servers fall into four basic types, depending on the functionality assigned to them within the overall site architecture:

1. ***The compute server*** —The compute server is the core of the overall system, and it is responsible for the actual computational task. This is the sole function to which traditional supercomputers and traditional MPPs can be assigned.
2. ***The data/storage server*** — Another computer in the overall system, the data/storage server, is assigned to manage the movement and storage of large volumes of data across and through a multiplicity of disks and storage devices. Mainframes and dedicated servers are used for this purpose.
3. ***The access/communications server*** — Still another system, the access/communications server, is responsible for all user access and communications into and out of the overall configuration. At complicated, modern sites, the demands upon this server may range from handling the I/O of a variety of devices, each with its own specifications and requirements, to the management of sophisticated networks. Specialized access servers and sometimes minicomputers are used to manage user access.

4. ***The visualization server*** — A relatively new development, but increasingly common at contemporary computer centers is the visualization server. This server is tasked to allow the users to envision the data or computations in ways that allow the human eye and mind to perceive their significance. Clusters of workstations are most often used for this purpose.

INTRODUCING THE KSR1 HIGHLY PARALLEL PROCESSOR

The KSR1 is a highly parallel open standards based system specifically designed to function as any one of the servers described above, as any combination of them, or as all of them simultaneously, while providing a system-wide parallel processing capability. The KSR1 performs the multiplicity of jobs ordinarily assigned to the various servers, simultaneously and in the face of the vastly differing circumstances and requirements that the tasks themselves may exhibit from moment to moment in real time. The KSR1 provides production-level parallel computing power, all of it scalable, with the familiar shared-memory programming model that is the contemporary industry standard. The use of shared memory enables a standards based, open environment [O/S, communications, languages and applications].

See Part Five, Architecture and Theory of Operation, for full details of the architecture of the KSR1 and its capabilities.

INTRODUCING ALLCACHE™

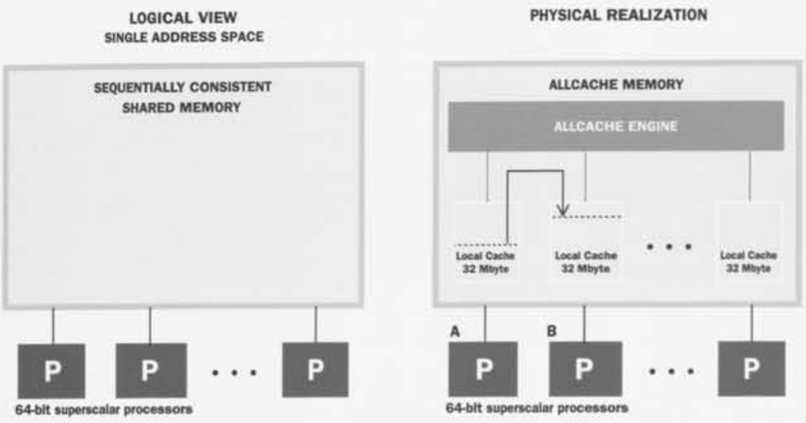
The patented ALLCACHE memory system is the enabling technology at the heart of the KSR1, a major innovation that allows the system to provide scalable, highly parallel processing power on a system-wide basis, whether the standard of measurement is sheer computational speed, I/O bandwidth, data throughput, or visualization. ALLCACHE merges the concept of virtual memory with the modern power of highly parallel processing to give the KSR1 unprecedented applicability.

ALLCACHE is precisely that — all cache memory that re-creates the time-honored standard of virtual memory for the contemporary parallel programmer. ALLCACHE returns the tasks of dynamic storage allocation and management to the hardware, relieving programmers of tasks they have not been compelled to do since the mid-1960s.

Figure 2 on the following page illustrates the ALLCACHE concept. When processor A first references the address X, hardware in the ALLCACHE memory system examines that processor's local cache to see if the requested address is already stored there. If processor B's local cache contains address X, the processor request is satisfied without any request to the ALLCACHE Engine. If not, the ALLCACHE Engine hardware locates another local cache (for example, local cache A) where the address and data exist. The ALLCACHE Engine moves addresses and their associated data to the point of reference on demand. There is no fixed physical location for an "address" within the ALLCACHE memory system, and this physically eliminates main memory.

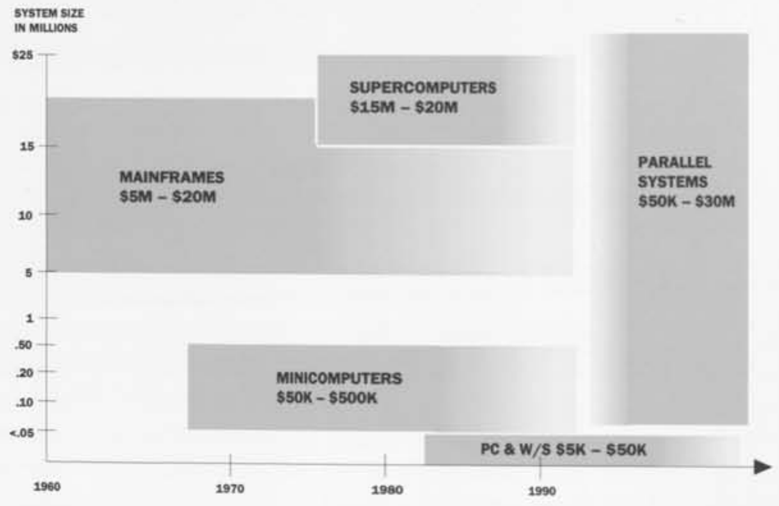
ALLCACHE is the first memory architecture to deliver the conventional, sequentially consistent shared memory programming model in a highly parallel computer. Thus it combines the memory model used by traditional mainframes and supercomputers with the scalability of highly parallel systems. Scalability allows users to add computer resources in incremental and cost-effective steps, without changes in software and without performance degradation. The entire KSR1 system implements sequential consistency to guarantee that a program will behave in a manner most intuitive to its programmer. The result of a program executed on the KSR1 is significantly faster but otherwise equivalent to the execution of the program on a conventional, multi-tasking single processor, which carries out its tasks in sequential fashion.

FIGURE 2 *ALLCACHE™ Memory System*



This fundamental improvement in the way that highly parallel processors can be programmed will serve as the basis for the next major wave in hardware architecture. Just as appearance of mini-computers in the late 1960s and supercomputers in the 1970s affected the history of mainframes, highly parallel processors that may be programmed as conventional systems will give rise to the computer industry depicted in Figure 3.

FIGURE 3 *Major Hardware Computing Waves*



The combination of the KSR1 highly parallel processor, its ALLCACHE memory system and a number of innovative features that take maximum advantage of the best qualities of both traditional programming methodologies and the cost-effective speed and power of highly parallel processing technology advances is described in detail in the following pages.

Patent 5,055,999

The following quotations are extracted from U.S. Patent No. 5,055,999, assigned to Kendall Square Research, 8 October 1991, for "MULTIPROCESSOR DIGITAL DATA PROCESSING SYSTEM." This patent, which describes the heart of the idea to cast the solution to the memory management problem of highly parallel processors in silicon for the first time, is the foundation concept underlying the architecture of the KSR1 system.

. . . . A system of the type provided by the invention does not require a main memory element, i.e., a memory element coupled to and shared by the system's many processors. Rather, data maintained by the system is distributed, both on exclusive and shared basis, among the memory elements associated with those processors.

. . . . the processing cells include central processing units coupled with memory elements, each including a physical data and control signal store, a directory, and a control element.

. . . . a controller coupled with each memory monitors the cell's internal bus and responds to local processor requests by comparing the request with descriptors listed in the corresponding directory.

. . . . a memory management unit facilitates. . . transfer of information.

. . . . Data movement between processing cells is governed by a protocol involving comparative evaluation of each access request with the access state associated with the requested item.

. . . . The caches of a KSR system can be used by system software as part of a multilevel storage system. In such a system, physical memory is multiplexed over a large address space via demand paging.

. . . . A system of the type described. . .

. . . . provides improved multiprocessing capability with reduced bus and memory contention.

. . . . The dynamic allocation of exclusive data copies to processors requiring exclusive access, as well as the sharing of data copies required concurrently by multiple processors reduces bus traffic and data access delays.

. . . . Utilization of a hardware-enforced access protocol further reduces bus and memory contention, while simultaneously decreasing software overhead required to maintain data coherency.

. . . . The interconnection of information transfer domain segments permits localization of data access, transfer and update requests.

. . . . These and other aspects of the invention are evident.

PART TWO

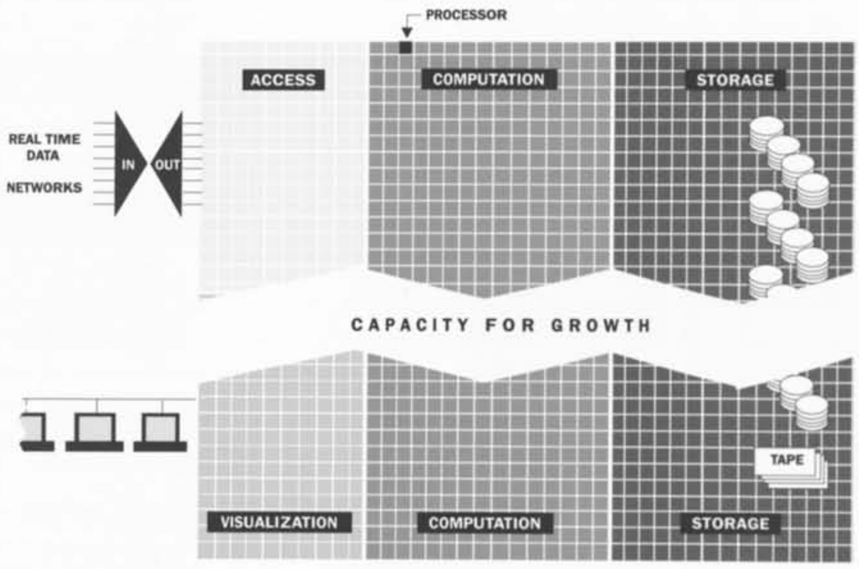
DEPLOYING THE KSR1

INTRODUCTION

Access, computation, data storage, visualization — these are the four functions performed by modern computer centers. Yet, at most modern centers, the four tasks are often assigned to different types of systems, creating the heterogeneous computer environments so common today.

By contrast, the KSR1, with its high performance capabilities and its patented ALLCACHE memory system, can serve in any of the four roles, in any combination of them, or in all of them at the same time. The KSR1 is truly general purpose in nature, and it provides a homogeneous computing environment with unlimited scalability.

FIGURE 4 *Homogeneous Computing Environment with Unlimited Scalability*



KSR1 AS A COMPUTE SERVER

In the specific arena of computational speed and power, the KSR1 provides users of high performance systems with an escape from the dilemma posed by the limited choice of architectures previously available. Prior to the introduction of the KSR1, users requiring high performance computing were restricted to the choice of supercomputers or mainframes on the one hand or massively parallel processors (MPPs) on the other. Mainframes and supercomputers permit a conventional programming environment, but are intrinsically limited in performance and bear a very high cost of computation. The MPP architectures, by comparison, offer high performance at a lower cost of computation, but have proven to be very difficult or impractical to program for most applications and have not been generally effective in multi-user, multi-application environments.

TABLE 1

PROCESSOR CONFIGURATIONS	PEAK MIPS	PEAK MFLOPS	MEMORY (MBYTES)	MAX. DISK CAPACITY (GBYTES)	MAX. I/O CAPACITY MBYTES/SEC
KSR1-8	320	320	256	210	210
KSR1-16	640	640	512	450	450
KSR1-32	1,280	1,280	1,024	450	450
KSR1-64	2,560	2,560	2,048	900	900
KSR1-128	5,120	5,120	4,096	1,800	1,800
KSR1-256	10,240	10,240	8,192	3,600	3,600
KSR1-512	20,480	20,480	16,384	7,200	7,200
KSR1-1088	43,520	43,520	34,816	15,300	15,300

The KSR1 offers an alternative to this dilemma by presenting a clear third path to high performance computing. The KSR1 provides users with a scalable family of computer systems (see Table 1 for configuration examples) which combine: the very high levels of performance and lower costs of computation inherent in parallel processing, with the high performance and ease-of-use of conventional shared memory programming, and the benefits of industry standards. These standards include OS, databases, communications, languages and applications.

On conventional MPPs the program must explicitly manage memory allocation and memory movement. On a KSR1, the ALLCACHE Engine embedded in hardware automatically manages memory allocation and movement relieving the program of these chores, delivering higher performance and ease of programming. Therefore programs explicitly written for message passing massively parallel computers will run faster on a KSR1 due to the higher efficiency of shared memory.

The ALLCACHE memory system architecture implements a sequentially consistent shared address space programming model, masking the physical distribution of local caches of memory. ALLCACHE automates the addressing and location of memory, so that programmers may be less concerned with the location of data while developing or porting programs to the KSR1.

ALLCACHE automatically moves an address requested by a processor to the 32 MByte local cache memory associated with that particular processor. Thus it exploits the "locality of reference" property of address reference sequences. (Programs and data, once referenced, are likely to be referenced again.) ALLCACHE keeps memory traffic close to the processor that is using the data, which is the key to the scalability of current and future products of Kendall Square Research.

Each KSR1 ALLCACHE Processor, Router and Directory cell (APRD) consists of a 64-bit superscalar processor, a 32 MByte local cache memory and a portion of the ALLCACHE Engine responsible for finding addresses and their contents and relocating them to the local cache of the processor requiring them, while maintaining sequential consistency among all the local caches within the system. (See Part Five, Architecture and Theory of Operation, for detailed information.) When taken as a whole, the collection of local caches behaves as a single shared address space.

The KSR1 processor employs 64-bit address, 64-bit integer, and 64-bit floating point data types, and it can perform arithmetic operations on IEEE standard 64-bit floating point numbers at a peak rate of 40 MFLOPS. To reduce memory traffic, each APRD has large register sets: 64 floating point registers, 32 integer registers and 32 address registers.

The KSR1 architecture allows it to perform work on a variety of jobs simultaneously, unlike traditional supercomputers and MPPs. Large scale users at a number of supercomputer centers have recently observed surprising patterns of workloads and user behavior/requests that indicate that a majority of user programs may require only a relatively short compile and execution runtime on the supercomputer, yet may be compelled to wait in queue for extended periods of time while larger programs are executed.

In these analyses of typical patterns of user demands upon CPU usage and performance, most requests are of a relatively low order of complexity or CPU time consumption. This has led to the use of clustered workstations to accommodate user demands. However, this approach has all the limitations of multicomputers, namely programming complexity, enormous operations complexity, poor load balancing and lack of scalability.

The KSR1 provides a far more expeditious and efficient solution, because it handles multiple jobs of widely varying sizes and requirements simultaneously ranging from uni-processing to massively parallel processing.

KSR1 AS A COMMUNICATIONS SERVER

The KSR1 is ideally suited to the communications management task. The high performance network, graphics and I/O connections of the KSR1 are both scalable and flexible. Peripheral capacity and effective bandwidth increase in step with the addition of central processing power to achieve a constantly balanced system.

The KSR1 permits interactive, batch, database and realtime applications to be run simultaneously, because the system is designed to support multiple user interaction models. All users are allowed transparent access to networks and devices, regardless of different network topologies and the varying levels of intelligence that may exist on the user interface devices (e.g., terminals and workstations). The KSR1 can simultaneously support the entire classical spectrum of user interactions:

- ***Distributed file*** — allows sharing of files between KSR1 systems, user interface devices and other computers
- ***Interactive interface*** — allows real-time access to information
- ***Batch processing*** — allows submission of jobs to be run without user interaction
- ***Client/server model*** — allows processing of a transaction on a single machine or split across multiple machines

Different users on the KSR1 interact with the system in ways that are already familiar and comfortable, including graphic and window-based interfaces.

Physical I/O connections to the KSR1 are of two basic types, depending on the performance level requirements:

- *Direct adapter connections.* The processor uses special adapters to connect with high performance and frequently used I/O interfaces such as the multiple channel disk, Fiber Distributed Data Interface (FDDI), Ethernet and the High Performance Parallel Interface (HiPPI);
- *General Purpose I/O System (GPIOS) connections.* The GPIOS provides convenient access to other networks and devices. The I/O subsystem provides industry-standard hardware (initially VME IEEE 1014 through the special VCC adapter) and the UNIX System V software environment to allow straightforward customization of applications. This cost-effective approach allows access to a wide variety of third-party hardware and software interfaces.

In a KSR1 system, software applications' access to the hardware can be device-independent and, if necessary, network-independent. This approach serves to protect user development investment as new networks and devices become available. Figure 5 on the following page shows connections between KSR1 systems and external devices.

APRD cells in the KSR1 system support 30 MBytes/sec transfers to external sources and users of data. Each 32-cell Processor Module can accommodate up to 15 I/O adapters. A KSR1-32 configured with 32 APRD cells thus achieves an aggregate I/O rate of 450 MBytes/sec (30 MBytes/sec x 15 adapters) and very high throughput with parallel I/O. A KSR1-1088 has an aggregate I/O capacity of 15,300 MBytes/sec.

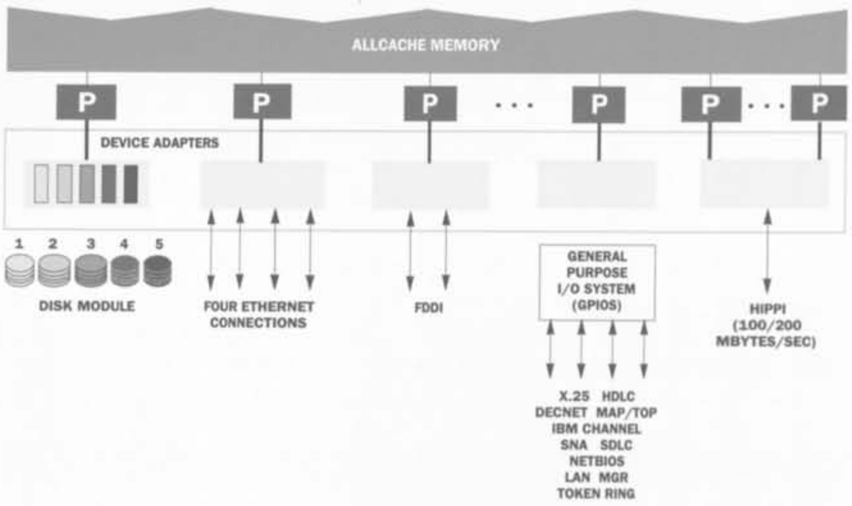
FIGURE 5 *Scalable I/O Bandwidth Combined with Standards*

UP TO 510 I/O CHANNELS PER KSR1-1088

ANY COMBINATION OF DEVICE ADAPTERS

EACH I/O CHANNEL 30 MBYTES/SEC BANDWIDTH

CONFIGURATION	I/O CHANNELS	BANDWIDTH (MBYTE/SEC)
KSR1-32	15	450
KSR1-128	60	1,800
KSR1-512	240	7,200
KSR1-1088	510	15,300



Adapters connect the APRD cell to specific I/O adapters:

- *Multiple Channel Disk (MCD)* — adapter supports disk arrays (RAID) that use five differential SCSI channels for connection to the mass storage subsystem;
- *Multiple Channel Ethernet (MCE)* — adapter with four Ethernet controllers for terminal and workstation servers;
- *Multiple Channel FDDI (MCF)* — adapter with two FDDI X3T9.5 controllers for connection to processors, terminal servers and workstations;
- *VME Channel Controller (VCC)* — adapter with an interface to the VME backplane. This gives users an open interface for insertion of the many peripheral and I/O boards which comply with IEEE 1014. The VCC adapter supports networking and an easy pathway for the customization of specific user requirements;
- *Single Channel HiPPI (SCH)* — adapter for HiPPI X3T9.3 connections supports both 100 and 200 MBytes/sec transfer rates.

Application access to either direct adapters or GPIOS is entirely transparent. For example, access to NFS files is independent of the physical connection type chosen. Transparency of access, which is network-independent, allows for future changes in network media or even protocol stacks.

The network connections supported on KSR1 high performance systems include:

- SNA 3270, 3770/RJE and LU6.2 over serial lines and X.25
- X.25
- X.29, X.28 and X.3
- FDDI
- Ethernet
- TCP/IP and NFS over Ethernet FDDI and X.25
- Serial lines

KSR1 AS A STORAGE SERVER

The KSR1 is an ideal system for the management of mass storage for groups, departments or an entire enterprise. Mass storage tasks include on-line updates via transaction processing (OLTP), file and archival storage and the maintenance and querying of complex databases. The KSR1 architecture achieves scalable performance as a storage, transaction and database server through a combination of two methods: reduction of disk I/O requirements by use of caching and scalable disk I/O.

An important principle in increasing I/O performance is to complete the operation using primary (main) memory. Main memory is 10,000 times faster (microseconds vs. 10s of milliseconds) than the time to access secondary (disk or tape) storage. The ALLCACHE memory system and 64 bit addressing of the KSR1 processor are fundamental to enabling operations which require physical disk I/O. On most systems physical disk I/O can be satisfied within ALLCACHE through a technique called single level store or mapped files.[1]

The second principle in increasing I/O performance is scalability in random access rates (typically required for OLTP and paging) and high bandwidths (typically required for complex database queries and file transfers). Although steadily improving access latency and transfer rates of mass storage devices have not kept pace with processing speeds, the size, power requirements and costs of mass storage devices are improving dramatically, allowing the deployment of redundant arrays of inexpensive disks (RAID).

Three distinct forces drive the trend toward the concept of multiple smaller disks under intelligent control as the optimum means of mass storage:

1. *Economic*: The use of PC and workstation technologies provide improved price/performance.

2. *Reliability*: The approach provides tolerance for individual disk failure. Expedients as simple as parity, ECC[2] and mirroring suggest seamless data integrity in an environment (e.g., OLTP) in which seconds of downtime may be unacceptable.

3. *Performance*: The use of many small disks allows parallel operations. The basic approach allows different mapping schemes and arrays to take full advantage of classical trade-offs: increase the number of actuators to increase the number of random seeks per second (e.g., OLTP); or spread the file across multiple disks to increase the I/O bandwidth (e.g., large-scale simulations in technical applications or complex database queries).

The KSR1 system combining a large shared memory, scalable I/O access rates and bandwidth, modular packaging of disk arrays, battery backup power system and a UNIX based operating system, incorporating RAID software, is capable of managing a broad spectrum of storage requirements.

1. Duby, R.C., J.B. Dennis, "Virtual Memory, Processes, and Sharing in Multics," Communications of the ACM, 11, 5, May 1968, pp. 306-312.
2. Error Correction Code

KSR1 FOR VISUALIZATION

Visualization is a new and fast growing area of computing. Its application is far reaching and spans areas as diverse as oil exploration, consumer purchasing patterns, and weather forecasting. Evolving from the simple graphs and charts of the past, true visualization gives the user tools to map any aspect of the data such as color, shape, density, motion and even sound in an exploratory environment.

As visualization needs grow beyond the capacity of the workstation, the same crippling architectural bottlenecks of mainframes and workstation clusters limit expansion of visualization capacity as they limit computation, communications and storage. Visualization is a demanding server application since it combines the need for large databases of 3D structures or scanned images, very high bandwidth communication to send many multi-megabyte pictures to a remote display device and very high computation rates to compute the motion of the millions of objects that make up a complex scene.

ALLCACHE provides a significant performance improvement for high end visualization, graphics rendering and image processing. With conventional multi-computers, whether a distributed cluster of workstations or message passing MPPs, it is necessary to rewrite the graphics or image processing programs, often from scratch, as a set of independent communicating tasks.

With ALLCACHE, existing visualization applications can be ported intact with only minor modifications. In graphics rendering and image processing in particular, a small portion of the code does most of the computational work, only this part needs to be tuned for parallel execution. Another important benefit derived from ALLCACHE is the improvement in performance that comes with memory allocation and memory movement being handled by the hardware rather than by message passing software.

Graphics Rendering

The KSR1 supports a native port of the OpenGL Graphics Library, an industry standard, three-dimensional graphics Application Programming Interface (API). This library includes facilities for drawing geometric objects, text, pixel operations, curves and surfaces, object hierarchies and objects for picking and selecting.

Direct display of rendered graphics from the KSR1 to a color monitor is accomplished via a HiPPI framestore at full resolution and at animation rates up to 170 MBytes per second. Lower resolution views or animations at non-realtime frame rates may be sent to a workstation over Ethernet, using KSR1 graphics compression architecture. (See the section below on "Networked Graphics" for more information.)

Scientific and Commercial Visualization

The KSR1 will support the Iris Explorer Visual Application Environment. Iris Explorer is a visualization product whose architecture has been designed specifically for maximum efficiency in a client-server environment. The user interface at the front end of Explorer, which resides on many popular graphics workstations, will communicate with the KSR1 parent over the network. Unlike earlier visualization applications, control of the modules' execution is distributed, rather than centralized at the level of the workstation. This design will leverage the full power of the KSR1 highly parallel processor in building very large visualization models.

Explorer is an application-building software system that provides tools to introduce and integrate visualization capabilities with an existing application. The package includes execution modules for numerical analysis, feature analysis, image processing, geometric representation and rendering. A visual programming model is used to interconnect the various modules. The system provides tools which create data filters but require no programming to add user-supplied modules.

Explorer is designed to support the following types of users: application developers who wish to add visualization capabilities to their applications; programmers whose function is to support groups of researchers; computational scientists whose primary task is to develop large-scale algorithms for high-order simulations; scientists who wish to simulate complex phenomena and explore the resulting parameter space visually; and business strategists who wish to analyze more completely complex sets of data.

Networked Graphics

The extremely high computational performance of the KSR1, combined with its massive I/O capacity and throughput, make possible a new computing paradigm for visualization. The KSR1 provides the means by which the power of the world's fastest computers can be brought down to the user's networked PC or X-terminal. The KSR1 is designed to stimulate progress toward the next evolutionary step in the development of the basic graphics model, from the current stage where polygons are drawn at a high performance graphics workstation, to a model that more closely resembles teleconferencing.

While the KSR1 can inherently provide solutions such as HiPPI framestores for high-end animation requirements, Kendall Square Research has continued to focus attention on the problems involved in providing graphics over conventional LANs to PCs and workstations. To this end, the KSR1 graphics architecture is organized to render animation sequences interactively to a software framestore in the KSR1, then to use advanced image compression to forward the images across the current-era network. While implementing JPEG (and, in the future, MPEG) for compatibility with emerging compression standards, Kendall Square has also adopted newer video compression schemes based on wavelet transforms and vector quantification. This provides both high quality displayed images for sharp edged computer graphics as well as faster decompression by software in workstations, PCs and X-terminals.

The graphics capabilities of the KSR1 conform to the de facto industry standards of the present day, and work continues to exploit the unique advantages of the KSR1 architecture while maintaining conformity. For example, Kendall Square's implementation of OpenGL will provide for parallel execution of entire frames in an animation, as well as fast-rendering of a single frame with parallel processing. The OpenGL implementation on the KSR1 will be optimized to leverage the advantage of high performance floating point capabilities for optimum image quality. For instance, rasterizing is accomplished by stochastic point sampling, rather than the conventional method of integer rasterization.

The KSR1 will meet the demanding needs of the visualization server for both technical and commercial applications.

PART THREE

SUPPORT FOR TECHNICAL APPLICATIONS

IDEAL FOR NUMERICALLY INTENSIVE PROCESSING

The KSR1 provides a completely integrated programming and development environment for applications, including all the software engineering features inherent in the UNIX operating system, along with an extensive set of programming languages and compilers, including Fortran, C, C++, Cobol, and assembly language. Numerous enhancements by Kendall Square, such as the addition of full screen debugging facilities, profiling tools, and utilities such as GNUMAKE, assures a productive environment for software development. For applications that require database management capabilities, the KSR1 supports the industry-leading DBMS software, ORACLE7. (See Part Four for details on database software.)

The software environment addresses the special requirements of multiprocessor parallel programming and makes the KSR1 an ideal platform for numerically intensive processing. It provides tools for examining the state of all the processors involved in an application simultaneously, and for analyzing the complex dynamical interaction of multiple processes. The Fortran compilation system incorporates advanced application parallelization technology. For database applications, the implementation of query decomposition in the relational database enables seamless parallel speedup on transaction-intensive applications.

BENEFITS OF THE SHARED MEMORY PROGRAMMING MODEL

The benefits of shared memory multiprocessors[1] are high performance and a conventional programming model (including virtual memory). When compared to multicomputers, the higher performance of shared memory has two benefits, efficiency and flexibility. Processor communication is more efficient because with shared memory, **hardware** automatically manages memory allocation, coherency and data movement transparently. In contrast, communication between processors within a multicomputer requires **software** to explicitly manage memory allocation, coherency and data movement.

The implicit nature of sharing an address space on a shared memory multiprocessor results in a flexible environment which dynamically moves only those addresses required by program execution between processors. In contrast, on a multicomputer, the exact details of all data movement must be specified statically at compile time. Attempts to build a more flexible environment on a multicomputer by emulating a shared address space in software[2] have resulted in lower performance on shared memory applications, since the shared memory primitives are in software. Thus, the most computationally efficient algorithms can be chosen to execute on a shared memory architecture such as the KSR1, while only explicit algorithms with rigid, static behavior execute efficiently on a multicomputer.

The shared memory programming model represents huge investments in existing programs and programmer training. Whether it is the development of new applications or the migration of existing codes from other shared memory systems, the KSR1 is the first large scale parallel system that

1. Bell, C. Gordon. "Multis: A New Class of Multiprocessor Computers;" Science, Vol. 228, pps. 462-467, 26 April 1985.
2. Li, Kai and Hudak, Paul. "Memory Coherence in Shared Virtual Memory Systems;" Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing, pps. 229-239, August, 1986.

preserves these investments while providing scalable performance for a broad range of applications. Existing codes can be migrated with minimal code changes. Starting with a port to a single processor, the application can be incrementally optimized and parallelized with data allocation and data sharing, transparently handled by the ALLCACHE memory system. A single, uniform address space of 1 Terabyte (1TB) is accessible to all processors. The system hardware handles data sharing among the processors, without resorting to the complexity of message-passing protocols. (See Part Four for further details.)

Thus users are not forced to rewrite their applications in data-parallel syntax or insert message passing calls to achieve parallel speedups. The parallel speedup is often achieved automatically (see section on Compilers and Languages below), or requires minimal modifications of the code. Not only can applications be ported easily to the KSR1, but because the KSR1 is programmed as a shared memory system, the resulting code remains portable from the KSR1 to other systems. Portability from the KSR1 greatly reduces the cost of maintaining applications across a number of platforms.[1]

BENEFITS OF INCREMENTAL OPTIMIZATION

During the migration process, a programmer is concerned with the following issues:

- Does the application port with minimal source code modifications?
- Does the performance of the application scale with the number of pthreads that participate in its execution?

On a KSR1, applications (whether previously parallelized or not) will run with no modification and depending on the previous level of parallelization some degree of parallelization will automatically be achieved. This is enabled by supporting the language extensions of several other vendors and the shared memory programming environment. The second question, *scalability,* has two aspects; scalability of memory size and scalability of performance. The user gets the benefit of scalability of memory size immediately because the ALLCACHE memory system transparently migrates addresses to the local cache processor being referenced. Data which has not been recently accessed is migrated to local caches that have excess capacity.

From the point of view of the application, local caches are treated as one large shared memory with its inherent aggregate storage capacity. This ability for a user to execute large applications on a single cell of the KSR1 sharply distinguishes the system from large scale message passing systems.

Since ALLCACHE handles data allocation and data sharing chores transparently, the KSR1 is ideally suited for incremental improvements of performance by restructuring only the key portions of the code.

Alternative architectures do not lend themselves to incremental optimization, as they must be completely rewritten. Data allocation and data sharing must be explicitly managed by the programmer. Only after the entire application is parallelized can the user run large problems that require the aggregate memory size of several processors.

1. S. Picano, E. Brooks, and J. Hoag, "Programming Costs of Explicit Memory Localization on a Large Scale Shared Memory Multiprocessor", Albuquerque, NM: *Proceedings of Supercomputing '91*, November, 1991.

IMPLEMENTING NEW ALGORITHMS

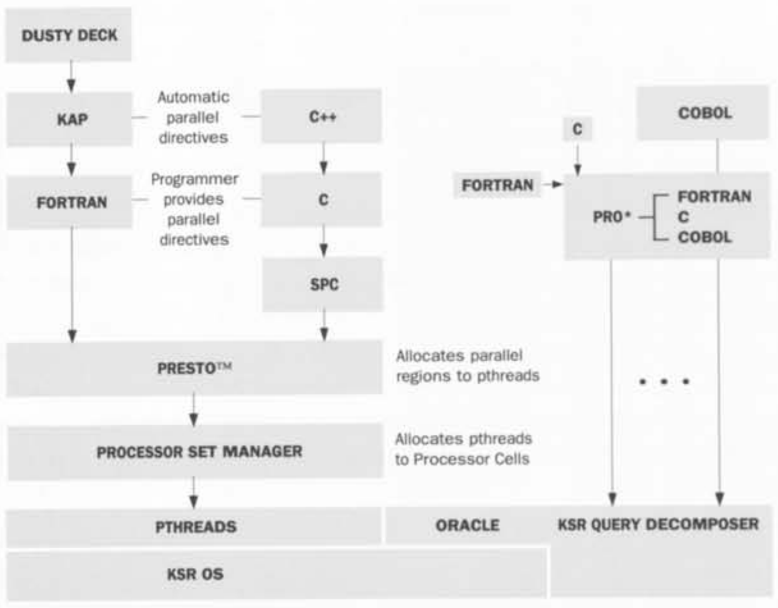
New algorithm development is easier with the shared memory programming environment. A programmer need not be constrained by having to first master data parallel programming techniques or message passing protocols. New algorithms can be "dropped-in" without rewriting the entire application, or being concerned over global data structures. This is illustrated in the examples at the end of this chapter.

PARALLEL PROGRAMMING ENVIRONMENT

The KSR OS facilitates technical and commercial applications including OLTP, relational database management and decision support.

The architecture enables single or multiple applications which mix technical and commercial characteristics to execute efficiently on a single KSR1 system as shown in Figure 6.

FIGURE 6 *KSR1 System Software*


DUSTY DECK
KAP
Automatic parallel directives
C++
COBOL
C
FORTRAN
Programmer provides parallel directives
C
FORTRAN
PRO*
FORTRAN
C
COBOL
SPC
PRESTO™
Allocates parallel regions to pthreads
PROCESSOR SET MANAGER
Allocates pthreads to Processor Cells
PTHREADS
ORACLE
KSR QUERY DECOMPOSER
KSR OS

The various languages are shown interfacing to lower level software layers. For example, the Fortran compiler, using automatic and programmer generated directives, creates parallel units of work, or ptasks. The number of ptasks created may be set at the time of compilation, or dynamically determined at runtime, based on user directives. The KSR1 runtime environment, PRESTO™, maps ptasks to pthreads. *Pthreads*[1] are POSIX-compliant lightweight processes that are scheduled by the KSR OS on available processors. Pthreads may be accessed by the programmer through the pthreads library calls, or through compiler directives which are replaced with calls to the pthreads library by the compiler. Pthreads provide a low-level interface to the operating system — thereby adding little overhead.

1. IEEE Technical Committee on Operating Systems, "Threads Extension for Portable Operating Systems", draft P1003.4a/D4, 1990.

The execution of a parallel Fortran program consists of the execution of one or more teams of pthreads. A "team" is a group of pthreads with a team identification number. Its behavior is as follows:

- A single pthread, called the program master, begins execution at the start of the program.
- When a pthread encounters a begin-parallel-directive, that pthread summons a team. The pthread that summons the team assumes the role of team leader. The other pthreads are the team members.
- Each parallel directive can take a team identification number as a parameter. If this parameter is specified, the designated team will execute the parallel segment. If no team is specified, the KSR Fortran runtime system, PRESTO, will designate a team to execute the parallel segment, creating a new team if needed.
- Each pthread in a team executes a portion of the work in the parallel segment. Together, the members execute all of it.
- When all the pthreads in the team reach the end-parallel-directive, the team is usually disbanded, its members are returned to the idle pool, and the team leader continues execution from the statement following the end of the parallel directive. Creation and disbanding of a team does not imply creation and destruction of pthreads: only the grouping of pthreads and the assignment of team identification numbers occur upon creation and disbanding of team operations. Teams that execute within an affinity region are not disbanded, until the end of the affinity region itself. Teams created explicitly by the user are not disbanded until the user explicitly disbands them.

Parallelization of technical applications is achieved by inserting parallelization directives in the source code. A preprocessor, KSR KAP, is available for source code analysis and automatic generation of compiler directives. While the compiler and KSR KAP are capable of automatic parallelization of applications, higher performance can sometimes be obtained with programmer assistance via compiler directives. These directives are described in more detail below. Message passing libraries are also provided as a convenience for some users, primarily for compatibility reasons, but not of necessity to the porting strategies themselves. Users will obtain higher performance by programming the KSR1 as a shared-memory system.

COMPILERS AND LANGUAGES

KSR FORTRAN

KSR Fortran lets programmers develop and port code from other systems quickly and easily. It adheres to the ANSI X3.9-1978 (Fortran 77) standard, and supports several popular extensions of the language present in standard Fortran 77 compilers such as the VAX/VMS, IBM VS, and Cray compilers. In order to maintain portability of the code from the KSR1, Kendall Square has not introduced any language extensions of its own to aid parallelization, rather it delivers the capability of constructs like "PARALLEL REGION" with simple compiler directives.

The KSR Fortran compiler provides optimizations seen in mature compilers[1], including:

- strength reduction
- loop invariant code motion
- common sub-expression elimination

1. M. Wolfe, *Optimizing Supercompilers for Supercomputers,* Cambridge, MA, the MIT Press, 1989.

- constant folding
- register allocation by graph coloring
- instruction scheduling/branch delay filling
- peephole optimization to minimize register-to-register moves, loads, and stores
- argument-passing in registers
- multi-level loop unrolling

There is ample hardware support to enable aggressive optimization, including a large set of registers, extensive pipelining, multiple instruction launch and chaining of operations.

FORTRAN EXTENSIONS

Some of the extensions beyond Fortran 77 that are included in KSR Fortran include:

Additional Data Types

INTEGER*1, INTEGER*2, INTEGER*4, INTEGER*8
LOGICAL*1, LOGICAL*2, LOGICAL * 4, LOGICAL *8
REAL*8 COMPLEX*16, DOUBLE COMPLEX

Bit Operations With Byte Addressability

Bit Field Manipulations
Bit Subfields
Bit Processing
Bit Constants

Dynamic Allocation Of Variables

Interlanguage Procedure Calls

The C Language

The KSR1 system supports C and offers an efficient C language compiler. The KSR C compiler conforms to the ANSI standard defined in ANSI X3.159-1989 (ANSI C). Parallelization of C codes is accomplished with manual insertion of pthread calls. Simple Parallel C (SPC) comprises higher-level subroutines that provide the same functionality as the Fortran parallelization directives described above.

C++ Compiler

C++ is an object-oriented language derived from C. On the KSR1 system, programs written in C++ may be run by compiling them with the C-Front pre-compiler.

C++ is a superset of the C language that provides flexible and efficient facilities for defining new "types." The key concept in C++ is *class* which is a user-defined type. Classes provide data hiding, guaranteed initialization of data, implicit type conversion for user-defined types, dynamic typing, user-controlled memory management and mechanisms for overloading operators.

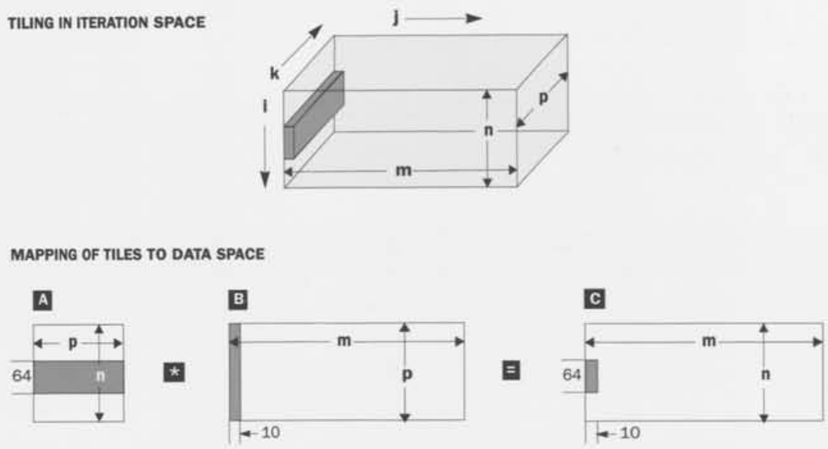
PARALLELIZATION CONSTRUCTS

The KSR1 Fortran compiler provides a powerful set of parallel processing capabilities. These include the ability to share variables and common blocks, and procedures for specification of parallel execution of code fragments.

Major parallel constructs include parallel regions, parallel sections and tile families which comprise the KSR1 high-level interface to pthreads. Each of these high-level interface directives is specified with *begin/end-parallel*-directive pairs surrounding the code fragments that may be executed in parallel.

- **Parallel region** – Execute multiple instances of a code fragment in parallel.
- **Parallel sections** – Execute multiple code fragments in parallel. Parallel sections in the text of a Fortran program are also denoted by a *begin/end-parallel-directive* pair.
- **Tiles** – Loop parallelization in KSR Fortran is achieved by *tiling*, in which the iteration space defined by a Fortran *do* loop nest is decomposed into *tiles*, or groups of loop iterations. The group of tiles that make up a loop nest is called a *tile family.* The tile directive specifies the loop indices over which tiling is to occur. These indices define an iterative space. For example, in Figure 7 the indices i, j and k define the iteration space. A point in this iteration space corresponds to unique values of the loop indices i, j and k.

FIGURE 7 *Iteration Space and Data Space*



```
    subroutine matrixmultiply (a,b,c,n,m,p)
    integer n,m,p
    real a(n,p), b(p,m), c(n,m)
c*ksr* tile (i,j)
    do i = 1,n
       do j = 1,m
          do k = 1,p
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
          end do
       end do
    end do
c*ksr* end tile
    end
```

Rather than have a pthread for each iteration, a tile directive creates groups of iterations that will be executed together. This increases the granularity of the parallel computation and minimizes scheduling overhead.

The tile directive causes the iteration space to be partitioned into rectilinear sub-spaces called *tiles*, each of which contains enough loop iterations to create a reasonable amount of work for one processor. Each tile is executed by a pthread, and each pthread may execute on a different processor. Numerous processors can execute the same loop nest, with each processor working on a separate tile simultaneously. There may be more tiles than pthreads, so that a given pthread may execute more than one tile.

The tile directive can specify parameters giving the user a broad range of control options. In the example presented in Figure 8, the runtime system determines the tile size and how they are allocated to pthreads, but the directives could have included additional tile parameters to control these choices.

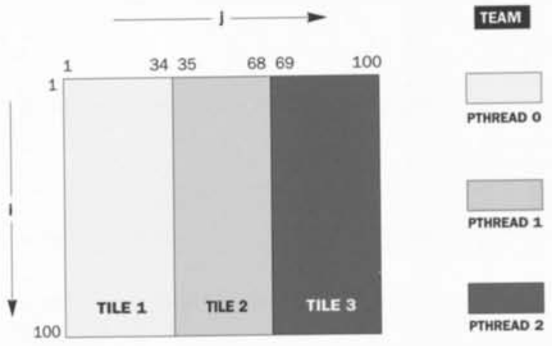
Major Tiling Strategies

The four distinct tiling strategies available to the programmer are outlined below. These fall into two major categories: static, as exemplified by *slice, mod, wave*, and dynamic as exemplified by *grab*. Depending upon the tiling strategy employed, each pthread will execute one or more tiles.

slice

The compiler simply slices the iteration space so that iterations are divided equally among the pthreads, where each pthread is to execute one tile. This strategy is the simplest and has the lowest overhead and is the default strategy adopted by the compilation system

FIGURE 8 *Slice Strategy*



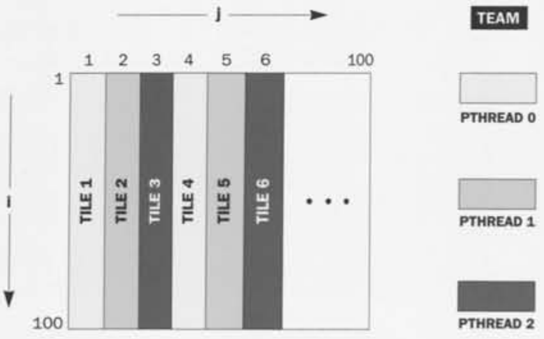
```
c*ksr* tile(i,j)
   do j = 1,100
      do i = 1,100
         a(i,j) = b(i,j)
      end do
   end do
c*ksr* end tile
```

NUMBER OF PTHREADS IN TEAM:		3
TILE SIZE:	i DIMENSION =	100
	j DIMENSION =	34 or 32
TOTAL TILES IN TILE FAMILY:		3
TILES PER PTHREAD:		1

mod

Assigns tiles to pthreads using a modulo mapping: tiles for which the tile number modulo the numbers of pthreads is the same are executed sequentially by the same pthread. There may be more tiles than pthreads, and thus each pthread may execute more than one tile. The compilation system chooses this strategy when data affinity must be maintained across a loop nest (as specified by the directive, *affinity region*) and the bounds of the iteration space vary dynamically.

FIGURE 9 *Modulo Strategy*



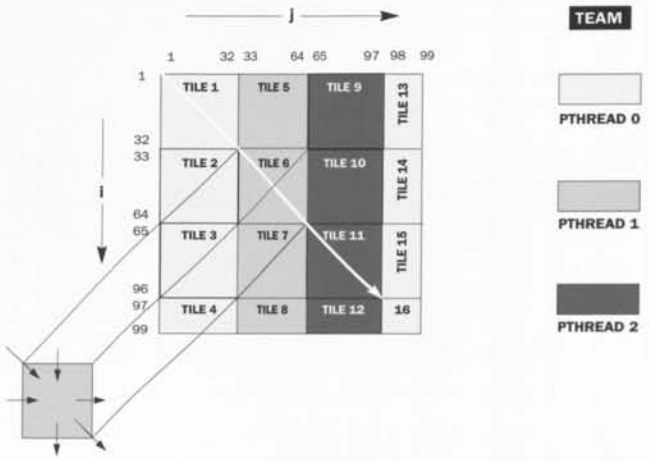
```
c*ksr* affinity region (i:1,100,j:1,100)
c*ksr* tile(i,j)
   do j = 1,100
        do i = 1,100
             a(i,j) = b(i,j)
        end do
   end do
c*ksr* end tile
c*ksr* tile(i,j)
   do j = 1,50
        do i = 1,20
             a(i,j) = b(i,j) + 1
        end do
   end do
c*ksr* end tile
c*ksr* end affinity region
```

TILE FAMILY		**FIRST**	**SECOND**
NUMBER OF PTHREADS IN TEAM:		3	3
TILE SIZE:	i DIMENSION =	100	20
	j DIMENSION =	1	50
TOTAL TILES IN TILE FAMILY:		100	50
TILES PER PTHREAD:		33 or 34	17 or 16

wave

The compilation system chooses this strategy when data dependencies impose ordering requirements. Typically there are more tiles than pthreads, and they will be executed in a wavefront manner, with synchronization that ensures program correctness. Assignment of tiles to their execution pthreads is done as in the modulo strategy.

FIGURE 10 *Wavefront Strategy*



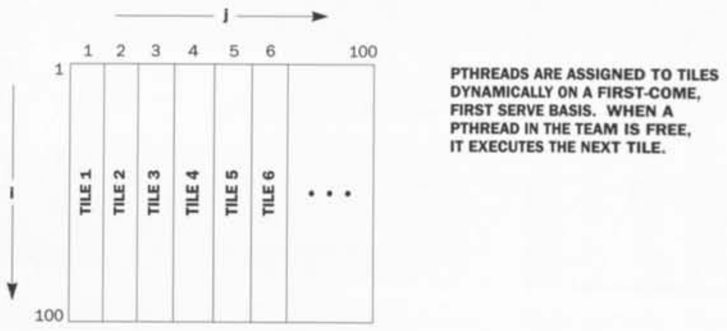
```
c*ksr* tile(i,j,order=(i,j))
    do j = 1,99
       do i = 1,99
          a(i,j) = a(i+1,j+1)
       end do
    end do
c*ksr* end tile
```

NUMBER OF PTHREADS IN TEAM:		3
TILE SIZE:	i DIMENSION =	32 or 3
	j DIMENSION =	32 or 3
TOTAL TILES IN TILE FAMILY:		16
TILES PER PTHREAD:		8 for pthread 1, 4 each for pthreads 2 and 3

grab

The grab strategy can adjust to an unbalanced load during execution of a tile family. With this strategy pthreads are assigned to tiles on a first come, first serve basis.

FIGURE 11 *Grab Strategy*



j
i
1 2 3 4 5 6 100
1
100
TILE 1
TILE 2
TILE 3
TILE 4
TILE 5
TILE 6
• • •
PTHREADS ARE ASSIGNED TO TILES DYNAMICALLY ON A FIRST-COME, FIRST SERVE BASIS. WHEN A PTHREAD IN THE TEAM IS FREE, IT EXECUTES THE NEXT TILE.

```
c*ksr* tile(i,j,strategy = grab)
   do j = 1,100
      do i = 1,100
         a(i,j) = b(i,j)
      end do
   end do
c*ksr* end tile
```

NUMBER OF PTHREADS IN TEAM:		3
TILE SIZE:	i DIMENSION =	100
	j DIMENSION =	1
TOTAL TILES IN TILE FAMILY:		100
TILES PER PTHREAD:		0–100

The directives that invoke particular tiling strategies are simple to insert, yet are powerful in their impact. On message passing systems, the equivalent functionality requires dozens of lines of user code.

If the user chooses not to specify a tile strategy, the compilation system and the run-time system do so, as follows:

- the *wave* strategy is used for tile families with ordering requirements
- the *mod* strategy is used when data affinity must be maintained
- the *slice* strategy is the default

KSR OS

The operating system of the KSR1, known as the KSR OS, is a UNIX operating system based on OSF/1 and is fully compatible with AT&T System V.2 and System V.3 and with Berkeley 4.3BSD and 4.4BSD. The Application Environment Specification (AES) is functionally complete, integrating the specifications of the major standards organizations, including ANSI C, FIPS 15-1, POSIX 1003.1 and XPG3.

The Unix shell, or command interpreter, has a built-in high-level language that allows the user to combine or pipeline programs without the need for compilation or recoding. Several standard shells are offered - including the C shell, tsch, and the Bourne shell.

KSR OS includes the standard set of UNIX text editors (vi, ed, and ex), as well as the GNU-MAKE screen editor. GNUMAKE provides an advanced set of editing capabilities that can be customized and extended by the user, with built-in help facilities that make it easy to use.

KSR OS also provides facilities for software development, including Revision Control System (RCS) and GNUMAKE. RCS supports the management of large development projects, in which many programmers are collaborating and where multiple versions of programs must be maintained. RCS manages the process of revising text files such as source code and documentation. It maintains a complete revision history, automates storage and retrieval of multiple versions of a program, provides release and configuration control, and controls programmer access to the source code.

GNUMAKE, a utility for maintaining consistency between program source and object modules, keeps track of the interdependencies between program elements so that when changes are made, modified source modules are automatically recompiled. A Kendall Square enhancement of GNUMAKE, known as *parallelmake*, automatically spreads compilations over multiple processors, taking a fraction of the time required on a single processor.

The udb debugger, which runs under the X Window System, or in a terminal mode, is a source level debugger designed for multiprocessors. A multiple-window display allows the programmer to track independent pthreads of execution, to set break points with mouse ("point and click") input, and to use pop-up menus for tracing program variables, call structures, or stack frames. The user can call on the integrated, context-sensitive, on-line help facility. Moreover, the command interface of udb is a superset of standard UNIX debuggers, such as dbx, so that minimal retraining is required for its use.

Without changing the standard user interface or file system, KSR OS gives users access to the powerful ALLCACHE shared memory system, the extensive address space, and scalable I/O and the large number of processors that may be configured in a KSR1.

The KSR OS environment also includes support for window-oriented user interfaces, built around industry standards, such as X-Windows and Motif.

Numerous extensions have been added to provide support for high-performance applications and for administrative functions.

A partial list of the KSR OS extensions to UNIX that facilitate operations in a highly parallel system environment is given below:

- KSR OS multi-threads the operating system to allow multiple processors to execute the same operating system code simultaneously.
- The I/O system supports parallel access for all read/write file system operations.
- Disk striping allows the construction of extremely large logical files over large numbers of physical disks. Data parity helps to maintain data integrity on logical disks. Files of up to 1 Terabyte are supported, and users are allowed to customize their file system for different types of file access.
- To reduce contention and data movement, the scheduler in KSR OS supports multiple-run queues, with the result that the operating system can sustain a simultaneous mix of jobs from high-performance OLTP to high-performance numerical computations.
- Process management facilities automatically group system resources, as appropriate. Cell and ALLCACHE Engine affinities are used to associate processes with a particular cell or ALLCACHE Engine:0, to maintain a favorable cache footprint.

A batch facility provides capabilities usually lacking in UNIX. Users can submit job requests specifying a minimum and maximum number of processor cells. The batch facility also supports other common batch-scheduling functions, including resource limits, allocation of I/O devices, and job status notification.[1]

KSR OS includes additional capabilities to ensure data integrity and high system availability:

- Disk parity, to allow the on-line replacement of components
- A check-point and restart facility for job recovery after unexpected interruptions
- Environment monitoring to help detect unsafe operating conditions
- On-line diagnostic facilities to help identify, report and deconfigure failing hardware components
- On-line maintenance
- Error logging
- Committed data recovery in conjunction with database management software
- Automatic system restart upon condition of a processor interconnect failure, with reduction of performance limited to the failed component so that operations may continue to the extent possible. The failed element need not be replaced immediately.

Single Level Store/Mapped Files

An important technique to increase I/O performance is to complete the operation using main memory and virtual memory hardware. Scalability and performance are significantly increased since the time to access primary (main) memory is 10,000 times faster (microseconds vs. 10s of milliseconds) than the time to access secondary (disk) storage. Other techniques such as software managed file caches are still 100-1000 times slower than main memory. The ALLCACHE memory system and 64 bit addressing of the KSR1 processor are fundamental to enabling operations which

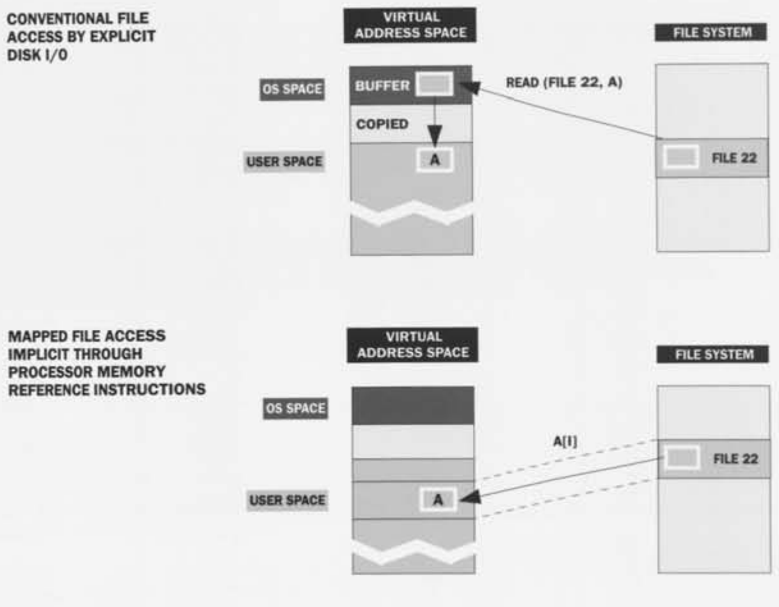
1. The batch facility is derived from the Network Queueing System developed by NASA/Ames.

require physical disk I/O on most systems to be satisfied within ALLCACHE through a technique called single level store or mapped files.[1]

On conventional mainframes, virtual address space is used to reference data and programs that have been copied explicitly from disk into a temporary region of virtual memory. The user's program moved the data and programs explicitly into the virtual address space with I/O commands that used a separate addressing scheme for the disk. Single level storage extends the concept of virtual memory to all storage. Files are never copied into virtual storage, as no explicit disk I/O is done by the program. A single address space (rather than one for memory and one for disks) is used to address all storage, both data and files. Thus, processor memory reference instructions are used to access files directly as data. Performance is accelerated, because the standard memory hierarchy is used to cache most recently accessed files.

Single level store requires virtual memory addressability of greater than 32 bits and scalable memory architecture to cache recently accessed portions of files. The KSR1 architecture uniquely meets these requirements in a scalable manner. The KSR1 processor and ALLCACHE meet the first requirement through native 64 bit addressing and meet the second requirement through hardware based hierarchical caches. ALLCACHE provides a scalable single level store by efficiently representing the most recently accessed portions of files (a sparse representation of the virtual address space) within a size and performance scalable physical memory.

FIGURE 12 *Mapped Files vs. Conventional File Access*



1. Duby, R.C., J.B. Dennis, "Virtual Memory, Processes, and Sharing in Multics," Communications of the ACM, 11, 5, May 1968, pp. 306-312.

Mass Storage

The KSR OS overlaps computing and I/O operations by means of asynchronous I/O. For example:

- A job can start several I/O operations through asynchronous I/O.
- A process can poll I/O operations or wait for an operating system signal on completion, whichever is more expeditious.
- The file system uses a read-ahead/write-behind feature.

System Administration

Processors can be divided into processor sets, and a given process can be restricted to one or more particular processor sets. For example, one processor set might be designated for interactive use, a second set for batch queue purposes, and a third set for a particular application. Processor sets can be altered manually or automatically by system administration commands without interrupting operations.

Scheduler

The KSR OS scheduler is hierarchical in nature and corresponds to the way the system itself is physically constructed. The scheduler is symmetrical and runs on each processor, rather than via a single master. It balances the load between processors and between processes with a single processor. For multi-threaded processes, processors can be added to or removed from an on-going process without disruption of the process' execution.

RUN-TIME ENVIRONMENT

The KSR1 runtime environment is called PRESTO. As seen in Figure 6 on page 21, the PRESTO environment is available to programs written in Fortran, C and C++. Its primary function is to provide a high-level interface between lightweight pthreads and the user application. The underlying POSIX-compliant pthread library contains primitives for the synchronization and management of pthreads, locks and barriers. PRESTO constructs are executed by teams of pthreads with a master responsible for executing the serial code before and after the parallel construct is encountered. PRESTO dynamically resizes the tile shape based on loop length and the number of operations per loop trip, and the number of pthreads assigned for the job.

It should be noted that PRESTO ensures load balance within a job, namely optimal assignment of tiles to pthreads. Load balance among processes on processors is handled by the OS scheduler.

PORTING CODES TO THE KSR1

Two examples of scientific/engineering applications are described:

Example 1: Automatic Parallelization

The first example is based on a code from "Programming Parallel Processors" by Robert G. Babb, Addison-Wesley (1988). The following code computes an approximation to Pi by using the rectangle rule on an integral representation of Pi. As Babb shows in the text book, even this simple example exposes the "flavor" of parallel programming environments. The critical component of the code is the three-statement DO 100 loop.

```
C
C - - Pi - Program loops over slices in interval, summing
C - - area of each slice
C
      real tt1(2), tt2(2)
      integer intrvls, cut
      real sumall, width, f, x
C
      f(x) = 4. / (1. + x * x)
C
      read(*,*) intrvls
      t2 = etime(tt1)
C
C - - Compute width of cuts
C
      width = 1. / intrvls
      sumall = 0.0
C
C - - Loop over interval, summing areas
C
      do 100 cut = 1, intrvls
         sumall = sumall + width * f((cut - .5) * width)
 100 continue
C
C - - Finish overall timing and write results
C
      t1 = etime(tt2)
      write(6, *) 'Time in main =', t1 - t2,', sum =', sumall
      write(6, *) 'Error =', sumall - 3.14159265358979323846
      stop
      end
```

On the KSR1, the critical loop is automatically parallelized by KSR KAP. No source code modifications are required. The command line for creating an executable is:

```
f77 -kap -para -O2 -o pi pi.f -lpresto.
```

The switches -para and -lpresto link parallel libraries, and the switch -O2 tells the compiler to use the highest optimization.

The compilation gives, together with an executable "pi," an intermediate Fortran file, pi.cmp:

```
C     KSR KAP              12.00 k091959 920206po1r2    23-Mar-1992 17:52:10
C
C - - Pi - Program loops over slices in interval, summing
C - - area of each slice
C
      REAL TT1(2), TT2(2)
      integer intrvls, cut
      real sumall, width, f, x
      SAVE INTRVLS
C
      F(X) = 4. / (X * X + 1.)
C
      READ (*, *) INTRVLS
      t2 = etime(tt1)
C
```

```
C - - Compute width of cuts
C
      WIDTH = 1. / INTRVLS
      SUMALL = 0.
C
C - - Loop over interval, summing areas
C
C*KSR* TILE (CUT,REDUCTION=(SUMALL))
      DO 2 CUT=1,INTRVLS
         SUMALL = SUMALL + WIDTH * F((CUT -.5) * WIDTH)
    2 CONTINUE
C*KSR* END TILE
C
C - - Finish overall timing and write results
C
      t1 = etime(tt2)
      WRITE (6, *) 'Time in main =', T1 - T2, ', sum =', SUMALL
      WRITE (6, *) 'Error =', SUMALL - 3.14159265358979323846
      stop
      end
```

It is this intermediate file (pi.cmp) that is actually read by the Fortran compiler. The preprocessor also yields a pi.out listing that describes in detail its optimization decisions.

The main function of the preprocessor is to introduce tiling directives, recognizable by the string C*KSR*.

With the directive,

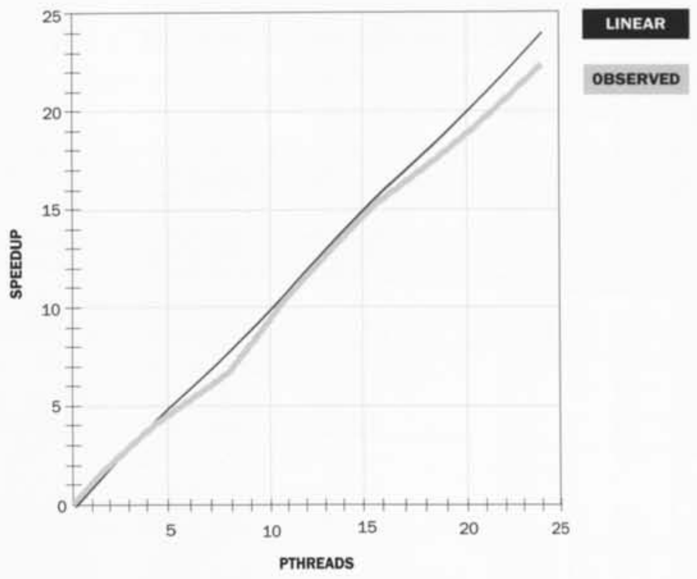
```
C*KSR* TILE (CUT,REDUCTION=(SUMALL)),
```

KSR KAP indicates to the compiler that CUT is the tiling index and that the SUMALL is a reduction variable. Each processor accumulates a private partial sum and adds it to the global sum, thus minimizing data contention and movement.

In codes that are parallelized manually, the desired number of processors is usually introduced as a variable. The 'Pi' source, however, contains no parallel construct; the desired level of parallelism is communicated to the program via a Unix environment variable (PL_NUM_THREADS).

The observed speedup is close to linear (see Figure 13).

FIGURE 13 *Speed up of the "Pi" Code With Automatic Parallelization*



Example 2: Incremental Optimization of the AMBER System of Codes

The AMBER system of codes was developed at the University of California, San Francisco, by Peter Kollman and Associates. AMBER is written predominantly in FORTRAN 77. It performs molecular dynamics and energy calculations on biological systems. Implementation of the AMBER module, MINMD, used for molecular dynamics and energy minimization is described below. The implementation included an initial port to a single processor, performance optimization on the single cell and subsequent parallelization and performance optimization for a KSR1-32.

The methodology used for porting the 110,000 lines of code is described in the following sections. Section I provides some background on energy minimization and the conventions used in porting to the KSR1. A description of porting and optimization for a single processor is reported in Section II and the parallelization effort is discussed in Section III.

I. Energy Minimization

Energy minimization is a technique in which the energy of a molecular system is minimized as a function of the molecular or atomic coordinates. The potential energy function and its gradients (the forces) are iteratively computed for sample configurations, until the change in energy is minimized. The potential energy function used in AMBER includes contributions from bonded atomic pairs, angular dependent atomic triplets, dihedral angular dependent quartets of atoms and non-bonded or long-range atomic pairs.

On the KSR1 the individual subroutines from module MINMD were compiled using a Makefile. The Makefile issued the following commands:

```
cat  file_name.f | /lib/cpp -P -DKSR1 > file_name_.F
      f77 -c -e -r8 file_name_.F
```

where "cpp" is the C-language preprocessor, that also can be used prior to FORTRAN compilation. The -DKSR1 option to "cpp" ensures the portability of the code between the KSR1 and any other Unix based computer, as all code modifications were enclosed within "cpp" directives. The following example illustrates the use of these directives:

```
#ifdef KSR1
      modified code
#else
      original code
#endif
```

The "-e" option to the compiler allows for Fortran source lines to extend beyond the 72nd column and the "-r8" option declares all real variables to have 64 bit precision. MINMD contains routines to time individual sections of the code. These timing data were used to improve the single cell performance and to isolate areas of the code that would benefit from parallelization.

II. Single Cell Performance

Improvements in the single cell performance were realized by reducing the storage of unnecessary temporary arrays and redundant loops, and combining many divide and square-root operations used in the computation of the forces. The superscalar nature of the KSR1 reduces the need for the temporary storage arrays that enhance performance on vector machines. In addition, the KSR1 hardware computes 1.0/sqrt(r_squared) in a single hardware operation. For example, in the original code, the interatomic distances were first stored as arrays, XWIJ(I), YWIJ(I), ZWIJ(I) and RWIJ(I), where I is the loop index (atom number) and the arrays were the X, Y and Z components of the distance and the resultant radial distances. The arrays were then used in another loop to compute:

```
RWIJ(I) = XWIJ(I)**2 + YWIJ(I)**2 + ZWIJ(I)**2
```

and subsequently used to compute 1.0/SQRT(RWIJ(I)). These steps were combined into a scalar operation (i.e. non-array) and the multiple loops were coalesced into a single step as:

```
        RWIJ = 1.0/SQRT(XWIJ**2+YWIJ**2+ZWIJ**2)
```

eliminating the need for the four storage arrays and reducing the chance of cache misses.

The energy minimization of triostin intercalculated in d(CGTACG)_2 molecule with 554 atoms was computed on a single processor. The output from AMBER produced a summary of the execution time for each of the subroutines and for sections of the computation. The force routines (NONBOND, EPHI and ANGL) account for 98% of the total time. Therefore these are the routines that were the focus of the parallelization effort.

III. Parallelization

Two different types of parallelism are utilized, known as PARALLEL REGIONS and PARALLEL SECTIONS. In PARALLEL REGIONS a single instantiation of a code is computed multiple times in parallel, while in PARALLEL SECTIONS, multiple groups of code are computed simultaneously.

The number of available processors is subdivided into three sets. The first set contains one pthread that is used to compute the bonded and angular dependent interactions. The second set contains between 1 and 3 pthreads which are used to compute the dihedral interactions. The third set contains all of the remaining pthreads and is used to compute the nonbonded interactions. The exact number of pthreads per set is dependent on the total number of processors that are being used. These parallel constructs were nested as follows:

```
C*KSR* Parallel Sections
C*KSR* Section! Section 1
       < compute bonded contributions >
       < compute angular contributions >
C*KSR* Section! Section 2
C*KSR*        Parallel Region (numthreads = iphi_procs)
              < compute dihedral contributions >
C*KSR*        End Parallel Region
C*KSR* Section! Section 3
C*KSR*        Parallel Region (numthreads = inb_procs)
              < compute non-bonded contributions >
C*KSR*        End Parallel Region
C*KSR* End Parallel Sections
```

In the preceding pseudocode, the parallel sections 1, 2 and 3 are computed simultaneously. Within sections 2 and 3, iphi_procs and inb_procs processors were used to compute the dihedral forces and the non-bonded forces respectively. The individual contributions to the total force from the nonbond, angle and dihedral interactions, are combined after the parallel section.

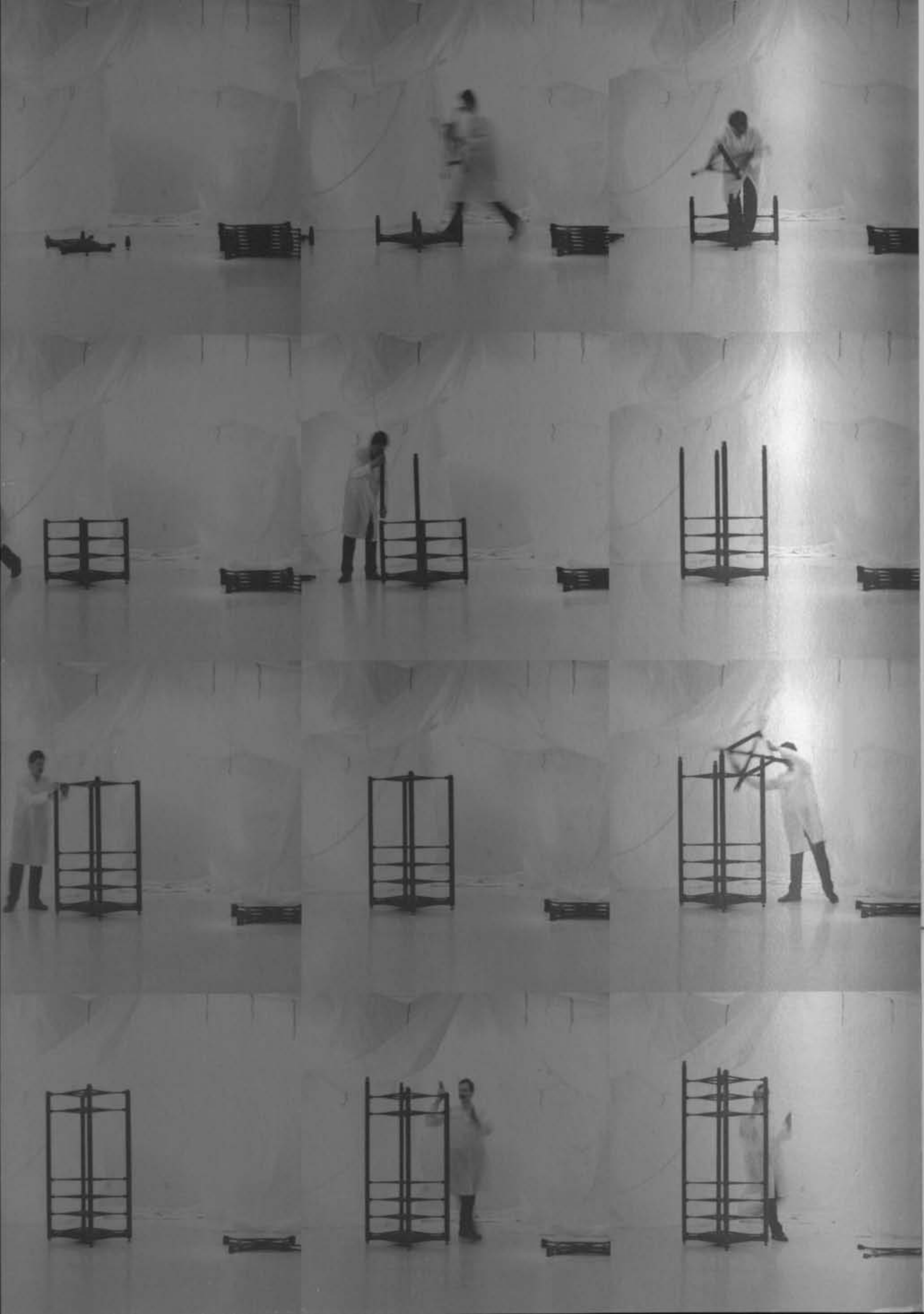
Fewer than 500 lines of source code had to be modified to enable a fully functional high-performance implementation of AMBER on the KSR1.

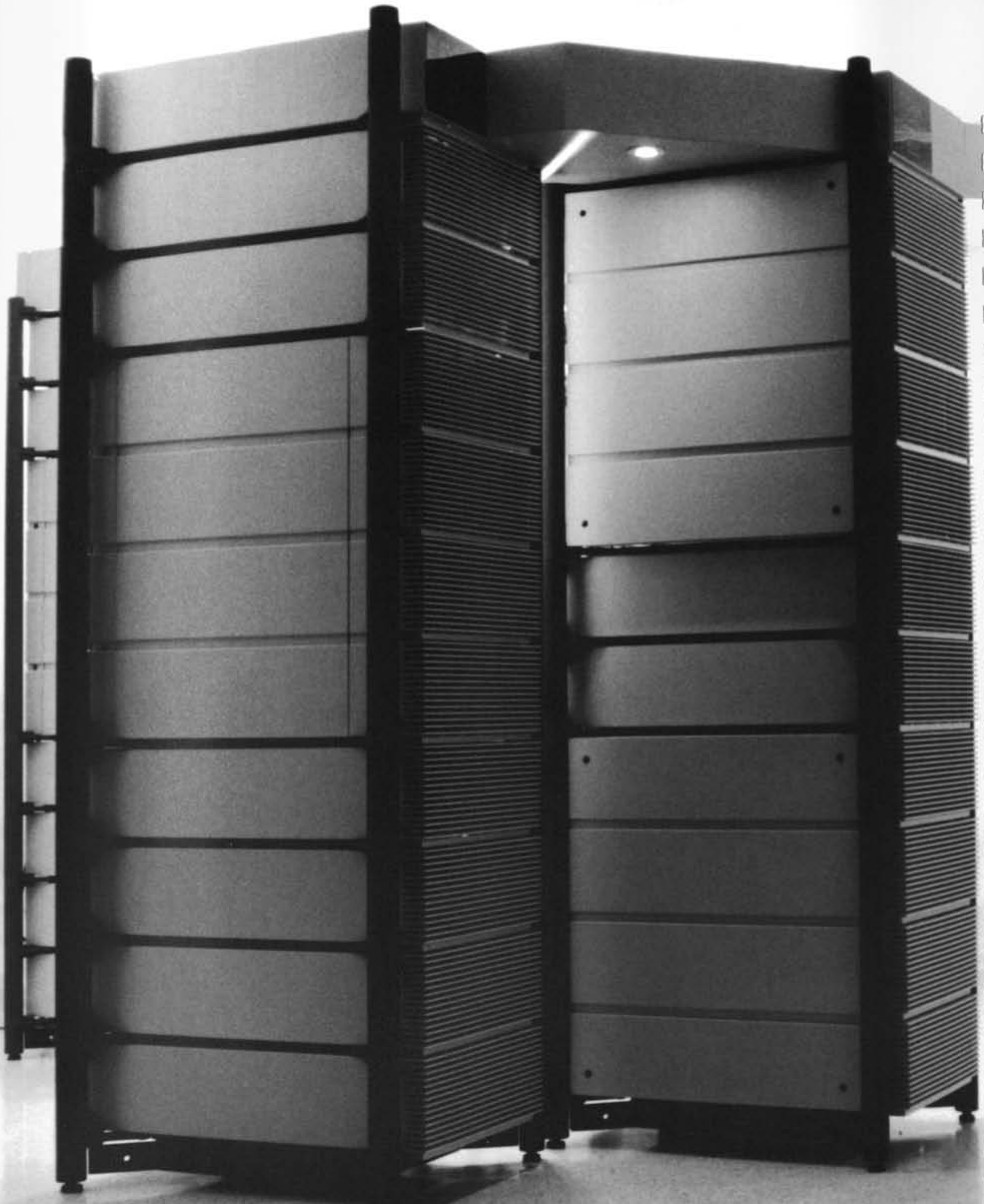
"THE MOST BEAUTIFUL THING
MYSTERIOUS. IT IS THE SOURCE

WE CAN EXPERIENCE IS THE

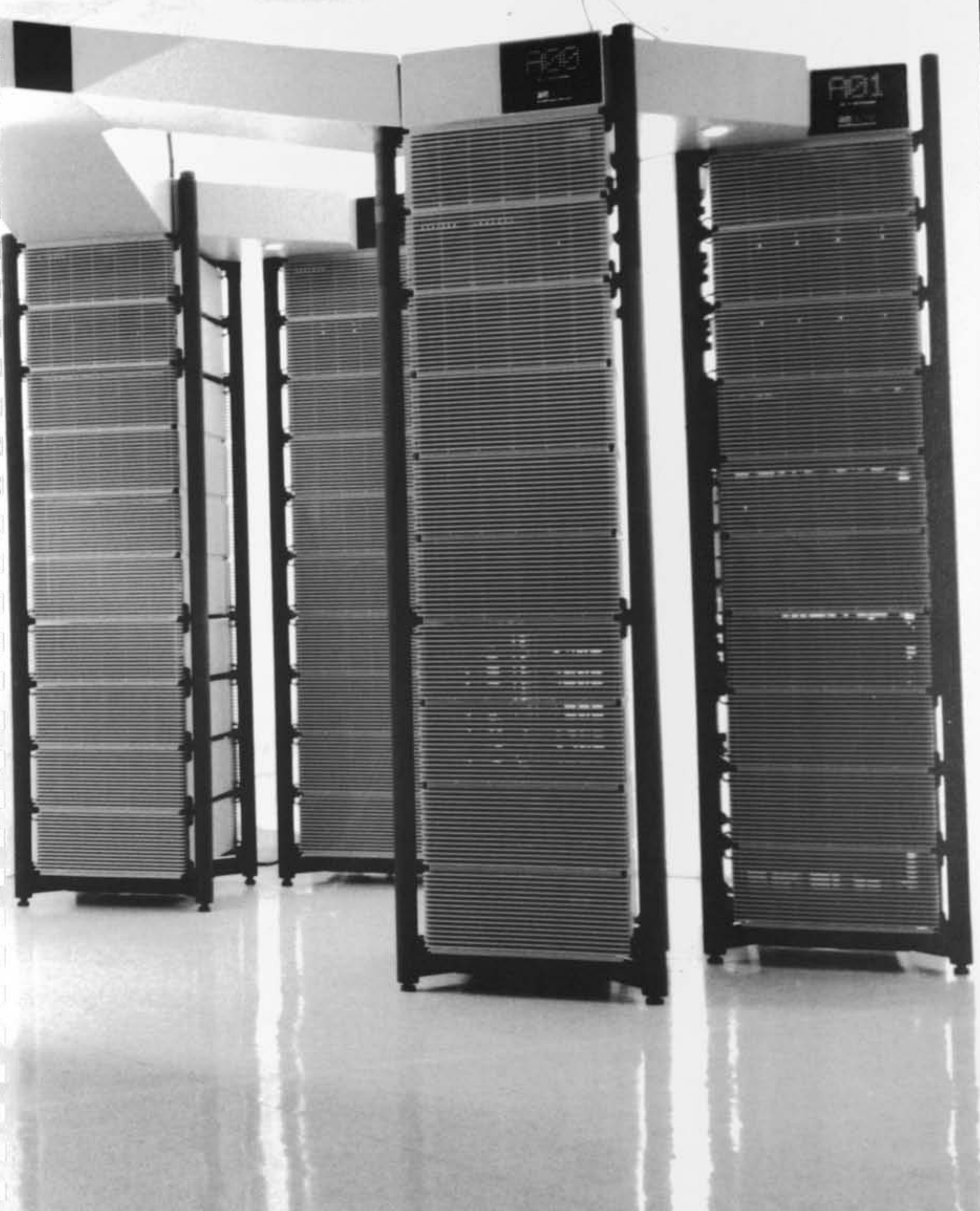
OF ALL TRUE ART AND SCIENCE."

ALBERT EINSTEIN

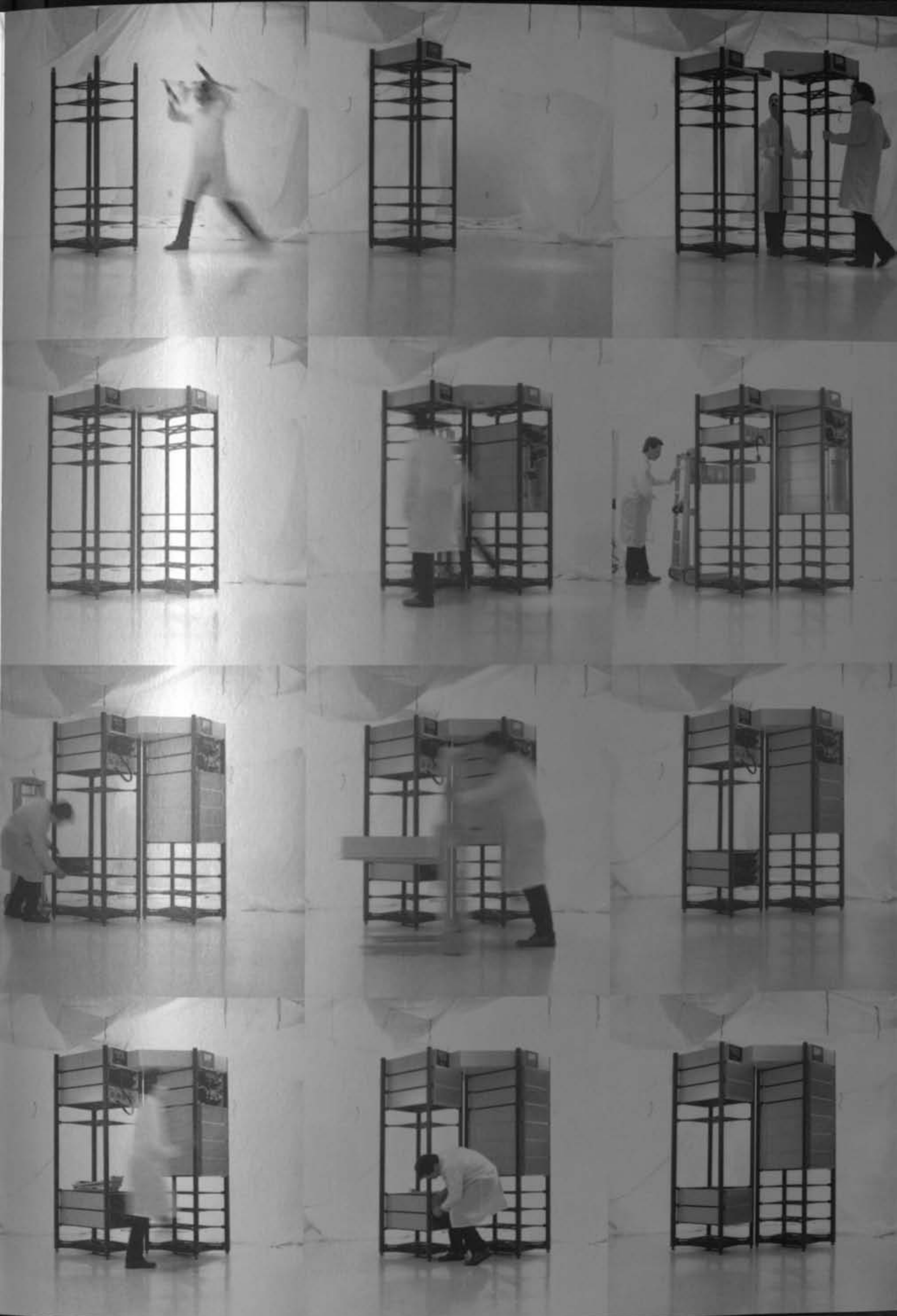




24 Photographs in Sequence: The KSR1 is designed as a fully modular system down to the component level. This allows customers to configure KSR1 systems with virtually any combination of disk drives, power modules or processors. Components can be added or changed as the need for memory, computational power, or redundancy dictate. Modularity is an important attribute of scalability.



CENTER SPREAD: THIS KSR1-256 IS A MULTI-TOWER COMPUTER CONFIGURED WITH 256 PROCESSORS PROVIDING, 10,240 PEAK MFLOPS, 8,192 MBYTES OF MEMORY, A MAXIMUM DISK CAPACITY OF 3,600 GBYTES AND 3,600 MBYTES/SEC. OF I/O CAPACITY. THE TOWER GROUPS ARE CONNECTED BY A CABLE TRAY WHICH CONTAINS THE INTERSYSTEM NETWORKING AND ELECTRICAL WIRING. NO SPECIAL COOLING OR FLOORING ARE REQUIRED.



PART FOUR

SUPPORT FOR COMMERCIAL APPLICATIONS

IDEAL FOR DATABASE PROCESSING

Very large commercial applications test the abilities of a system to handle many different challenges simultaneously — data management, I/O, computation, network access and visualization. The KSR1 system is the first highly parallel computer architected from the beginning to meet the needs of commercial users.

At the heart of the machine, the KSR1 ALLCACHE memory system brings dramatic and substantial benefits to database processing. It supports:

- Loading large databases into memory, with data accessible to all processors simultaneously
- Automatic migration of copies of frequently referenced indices and lookup tables
- Deployment of conventional, industry standard database management software
- High performance inherent in the shared memory programming model

When ALLCACHE is combined with the process-scheduling capabilities of KSR OS, the high-performance and scalable I/O subsystem, and the automatic KSR Query Decomposer, the result is a unique environment where, transparently to the developer and end-user:

- On-line transactions can be run in parallel at unprecedented throughput rates
- Decision support queries can be decomposed, parallelized and very rapidly executed

CRITICAL ISSUES FOR COMMERCIAL USERS

The KSR1 is the ideal high performance computer for users in industries such as banking and financial services, insurance, information services, telecommunications and cable, consumer and packaged goods and transportation. Users with large scale applications have focused on the issues below:

WORKLOAD

Large and continually-increasing data volumes

Very high transaction volumes per unit of time

Rapid processing of complex queries

High performance for all categories of database processing

Virtually unlimited scalability

ADMINISTRATION

High availability for production usage

Ability to mix and control processing categories

Complex processing cycles (extract, cleanup, merge)

Familiar application development approaches

Preservation of investment in legacy data and applications

Effective coexistence with the existing DP environment

Across the spectrum of database workloads, the underlying parallelism and memory management architecture of the KSR1 manifest themselves in previously unattainable performance and practically unlimited scalability, while maintaining conventional ease-of-use for developers and end users. In the database software offering, these benefits are combined with a strong emphasis on standards, openness and co-existence with customers' legacy systems.

Table 1 below illustrates the differences between decision support and on-line transaction processing (OLTP) workloads. As a rule, large numbers of short transactions that modify the database characterize OLTP. In contrast, lesser numbers of longer-running queries that must read and analyze data characterize decision support applications. The KSR1 serves as a platform for both modes of operation, either singly or in combination.

TABLE 2

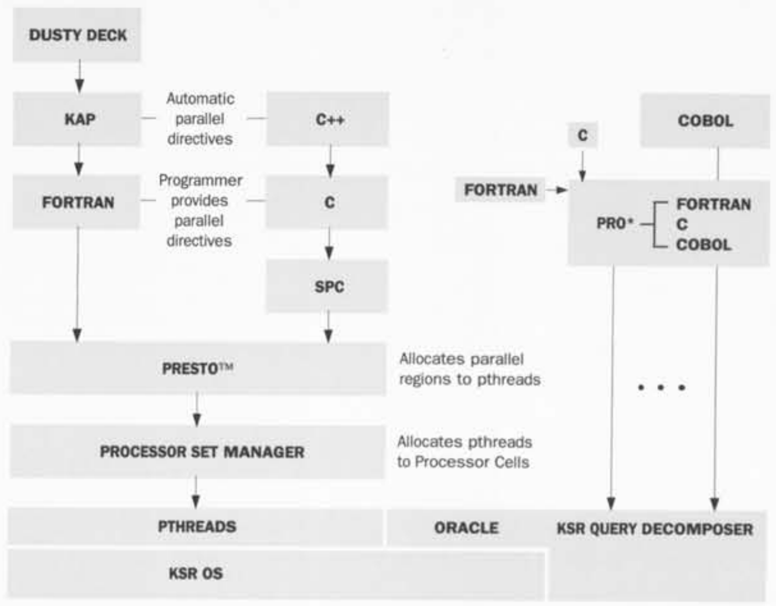
CATEGORY	DECISION SUPPORT	OLTP
Type of access	Read-only	Reads and writes
Complexity	Medium to high	Low to medium
Number of joins	Medium to high	Low
Data intensity	High	Medium
Query selectivity	Low	High
I/O access patterns	Mainly sequential	Random
Transaction volumes	10-200K queries/day	10-1000 TPS or more

Application level high availability and database integrity is provided through component redundancy, committed data recovery and fast system restart. The KSR1 maintains highly available features throughout the system. (See Part Five for greater detail.) A key objective underlying the KSR1 is to ensure that it will rapidly restart, with data integrity guaranteed at all times. Redundant support and on-line replacement are available throughout the system in Processor Modules, the disk subsystem, the power system and I/O subsystem.

BROAD COMMERCIAL SOFTWARE OFFERING

The KSR1 system software facilitates both commercial applications and technical applications. The software architecture is shown in Figure 14. This architecture enables single or multiple applications with mixed commercial and technical characteristics to execute efficiently on a single KSR1 system.

FIGURE 14 *KSR1 System Software*


DUSTY DECK
KAP
Automatic parallel directives
C++
Programmer provides parallel directives
FORTRAN
C
SPC
PRESTO™
Allocates parallel regions to pthreads
PROCESSOR SET MANAGER
Allocates pthreads to Processor Cells
PTHREADS
KSR OS
C
COBOL
FORTRAN
PRO* FORTRAN C COBOL
• • •
ORACLE
KSR QUERY DECOMPOSER

The KSR1 system offers application developers a widely-deployed, state-of-the-art relational database management system (ORACLE7); a proven, UNIX-based transaction monitor for high-volume OLTP (TUXEDO/T); and a range of familiar database application development software that includes COBOL, fourth generation languages (4GLs), CASE and GUI tools, Fortran and C.[1]

In addition to supporting ORACLE, TUXEDO and COBOL, Kendall Square has added significant value to the software while maintaining standard interfaces and languages. The innovative KSR Query Decomposer works with ORACLE to parallelize decision support queries automatically. The KSR OS process scheduler enables a huge degree of parallelism for transaction processing, aided by parallelism in TUXEDO internal data structures. COBOL benefits greatly from the ability of the KSR1 to run large numbers of application instances, and to parallelize the processing of the database and network requests that they generate.

Each of the major areas for commercial applications is described in more detail below.

THE ORACLE DBMS

ORACLE is a comprehensive DBMS, with a strong client/server orientation. ORACLE7 features high performance data access, sophisticated query processing, complete data integrity enforcement, and heterogeneous and distributed database processing. It is well-known in the commercial world for full application portability, industrial strength application development tools and wide database connectivity.

Contributing to its high performance are ORACLE7's query optimization based on database statistics; row-level locking to minimize data contention; flexible matching of client processes to

1. Support for Fortran, C and C++ languages are described in Part Three, Support for Technical Applications.

server processes; fast commit, group commit and deferred write technologies; data partitioning and multi-table clustering; and multi-instance capabilities (Parallel Server). Oracle supports very large databases as well as binary large objects (e.g., image data). Database integrity is enforced through multi-user concurrency control, transaction management and recovery, declarative referential integrity, triggers and stored procedures.

ORACLE7's full tool set includes 3rd and 4th-generation languages, graphical and character-based form and report generators, CASE tools, application generators and end-user tools. On the KSR1, precompilers and runtime libraries for COBOL, Fortran, C and C++ programs allow them to call the DBMS directly.

Commercial applications previously developed with the ORACLE tool set on other hardware platforms are directly portable to the KSR1, and the reverse is true as well. Currently, more than 1,000 third-party applications and tools have been developed on ORACLE, covering such fields as accounting, business planning and control, CASE, financial management, human resources, insurance, materials management, project management, sales/marketing and telemarketing. Oracle Financials and Oracle Core Manufacturing applications are available directly from the Oracle Corporation.

Gateways to other databases are enabled by interfaces to a wide array of networks, such as IBM SNA, TCP/IP, LAN Manager, NFS, IBM channel connect and HiPPI. On top of this networking capability runs ORACLE software for on-line connection to DB2, SQL/DS, RMS, Teradata and TurboIMAGE, as well as copies of ORACLE running on other platforms.

Access to heterogeneous data sources is also provided by loading flat-file extracts from sources such as VSAM files, IDMS, IMS, DATACOM, Model 204, ADABAS, Sybase SQL Server and INFORMIX. ORACLE has a fast and flexible bulk loader well suited to applications demanding large or frequent extracts from other environments.

DECISION SUPPORT – KSR QUERY DECOMPOSER

The KSR Query Decomposer works transparently in conjunction with the underlying ORACLE7 relational database management system to greatly speed the execution of decision support queries. Such queries are typically data-intensive and expensive to run, requiring that major portions of one or more relational tables be scanned by the DBMS. Generated by applications or by on-line users, they sweep sequentially across the database to detect trends, make comparisons across years or quarters, summarize results for product lines or regions, and so on. They typically require combining information from multiple tables, and often include sorting, grouping, averaging, counting, and otherwise aggregating large amounts of data. The data returned may be further analyzed, for example in a spreadsheet or statistical package, or the user or program may generate follow-up queries in an iterative fashion. The sample queries below illustrate the difference between OLTP and decision support.

OLTP— Short-running queries

```
SELECT ACCT_BALANCE
FROM ACCT
WHERE ACCT_NO = 23586
```

Decision Support — Long-running queries

```
SELECT ACCT.ACCT_NO,
       AVG(XACTION.AMT)
FROM ACCT, XACTION
WHERE ACCT.REGION = "Northeast"
AND ACCT.BALANCE<1000
AND ACCT.ACCT_NO = XACTION. ACCT_NO
GROUP BY ACCT
```

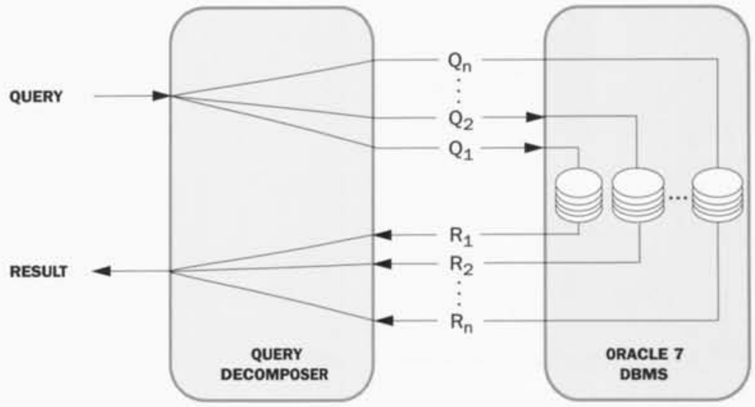
By their nature, decision support queries place a heavy load on computer system resources. They are less predictable than simpler queries, and may well interfere with the smooth processing of mission-critical OLTP loads if done concurrently. Every database administrator (DBA) is familiar with the "killer query" which – however innocently intended – brings a system to its knees.

HOW THE KSR QUERY DECOMPOSER WORKS

In order to maximize parallel reads from disks for decision support queries, all large database tables are partitioned physically over multiple disks by the DBA, using straightforward ORACLE data definition language (DDL) commands. There may be tens or hundreds of partitions for a given table, which can be populated randomly or via a hash function that scatters tuples into small clusters. These partitioning techniques were originally developed to spread out OLTP "hotspots," or frequently accessed portions of tables, but they are well suited to supporting the KSR Query Decomposer.

When an application, tool, or user first submits a query to ORACLE, it is intercepted by the Query Decomposer at a common processing point (see figure below). Queries that will not benefit from decomposition are simply passed through to ORACLE unchanged; those that do are transformed. Based primarily on the data access strategy selected by the ORACLE query optimizer, the Query Decomposer generates a number of subqueries to match the underlying physical data partitions of one of the partitioned tables, called the "driving" table. Decisions made automatically include: the number of subqueries, the choice of the driving table, the minor query transformations to handle aggregate functions and the method of combining subquery results. Each subquery looks very much like the original query, with minor transformations that include an additional condition to restrict it to just one partition of the driving table.

Flow of Processing with KSR Query Decomposer



The subqueries are submitted in parallel to ORACLE over multiple, coordinated connections established by the Query Decomposer, so that they see a consistent view of the database. All of the subqueries are executed in parallel, each accessing only its own part of the driving table, and doing further joins as required. Since partitions are approximately equal in size, initial partition scans are unaffected by data skew. Taking advantage of the ALLCACHE memory system, common lookup tables and index pages are only brought in once from disk, and are shared among subqueries when needed.

Subquery results are combined inside of the Query Decomposer, and returned to the application, tool, or user which submitted the original query. If the original query called for grouped or sorted data, so will the subqueries, and the Query Decomposer will correctly combine their results. This effectively gives the user a parallel sorting capability for decision support queries. Aggregate functions such as maximum, count and average are also correctly computed when subquery results are combined.

Two sample queries are shown below, one straightforward and the other requiring additional query transformation on the part of the Query Decomposer.

Sample Query #1

Assume department (dept) and employee (emp) tables are partitioned into multiple files across disks. Assume there is an index on dept.deptno, with none on emp.title. The initial query is:

```
SELECT empname, empno, deptloc
FROM emp, dept
WHERE emp.deptno = dept.deptno
AND emp.title = 'MTS'
```

The ORACLE7 query optimizer reports that it will use a nested loop join with emp as the outer table and dept as the inner one. In other words, its data access strategy is:

```
Loop through all emp tuples;
for each emp tuple meeting the condition on emp.title
      use index on dept.deptno to access
      corresponding dept tuple;
```

The ith subquery generated by the Query Decomposer is below, with "i" and "i+1" replaced by actual values:

```
SELECT empname, empno, deptloc
FROM emp, dept
WHERE emp.deptno - dept.deptno
AND emp.title = 'MTS'
AND emp.rowid >= '0.0.i' AND emp.rowid <'0.0.i+1'
```

Each subquery executes against a different partition of emp, based on ORACLE's interpretation of the two conditions on emp.rowid. Subquery results are combined by the Query Decomposer using a Union All operation.

Sample Query #2

The assumptions are as in Sample Query #1. The initial query is:

```
SELECT deptno, AVG(salary)
FROM emp
GROUP BY deptno
```

The ith subquery generated is:

```
SELECT deptno, SUM(salary), COUNT(salary)
FROM emp
WHERE emp.rowid >= '0.0.i' AND emp.rowid <'0.0.i+1'
GROUP BY deptno
```

As before, each subquery executes against a different partition of emp. The Query Decomposer creates a two-column temporary table (with columns sum_col and count_col) which is filled once for each deptno with SUM(salary), COUNT(salary) tuples. There is at most one such tuple for each partition.

Subquery results are combined, calculating average salaries with the following query which is executed once against the temporary table for each deptno:

```
SELECT (SUM(sum_col)/SUM(count_col))
FROM temporary table
```

APPLICATION DEVELOPMENT WITH THE QUERY DECOMPOSER

The decomposition process is transparent to the developer and end-user, except for the substantial performance improvement it brings. Queries do not need to be modified, and existing ORACLE applications do not need to be re-written to use this powerful facility. The Query Decomposer works transparently on queries once submitted.

Conceptual and logical database designs are unchanged. When use of the Query Decomposer is planned, the physical database design is slightly different in two areas:

- Large tables should be partitioned across disks by the DBA, with one table partition per disk drive.
- Hash clusters, supported in ORACLE7, are an effective design approach for use with the Query Decomposer. If tables are relatively static in size, hash clustering may be used with small hash buckets (we call this combination "scatter clustering"), and with a secondary index built whose initial fields are the same as those used in calculating the hash key. This combination maximizes I/O parallelism for subqueries, while minimizing total I/Os.

ON-LINE TRANSACTION PROCESSING

The KSR1 provides an ideal environment for OLTP, featuring:

- Very large physical memory and enormous virtual address space (1 Terabyte per process)
- Large numbers of MIPs, disks and broad I/O bandwidth, with nearly unlimited scalability
- Disk arrays, disk striping and data partitioning to maximize bandwidth
- Disk mirroring for enhanced reliability and faster read performance
- Broad connectivity in heterogeneous environments
- Dynamic processor scheduling by the KSR OS
- Fault-tolerant database processing (via ORACLE7)
- Application level high availability and database integrity through component redundancy, database committed data recovery and fast system restart

In the KSR1, the multiple transactions of an OLTP load are each assigned to processes which, in turn, are adaptively assigned by KSR OS to the processors available; therefore, throughput scales with the number of processors.

The shared memory architecture of the KSR1 leverages locality of reference on such occasions by migrating copies of subpages to the processors that have referenced them. For typical OLTP loads, the top-level index pages and smaller look-up tables are generally present in local cache memory when needed.

The KSR Distributed Lock Manager coordinates multiple instances of ORACLE (Parallel Server), to support scalability of OLTP load processing within the DBMS itself.

TUXEDO TRANSACTION MONITOR

For very high-volume OLTP applications, Kendall Square has ported and enhanced the TUXEDO 4.2 Enterprise Transaction Processing components from UNIX System Laboratories. These include the TUXEDO/T transaction monitor and the TUXEDO/D access method: they provide a modular, industrial-strength environment for developing client/server, OLTP applications.

TUXEDO/T provides the services of traditional mainframe-class OLTP systems while supporting modularity and heterogeneity of software and hardware. Based on the distributed client/server model, TUXEDO/T provides communication services, including device support for PCs or workstations not part of the /T system, distributed two-phase commit, distribution of services within local- or wide-area networks, name services and online administration and maintenance services.

TUXEDO/T can be called from COBOL, C, C++ and ORACLE application clients, and can use ORACLE, /D and other servers. For client and server programs, /T employs the standard ATMI interface adopted by X/Open.

Neither clients nor servers need reside on the KSR1 itself. For example, TUXEDO System/WS supports the use of ATMI's client-side functionality on the KSR1 from PCs (under MS-DOS) and workstations (under UNIX). TUXEDO System/Host allows OLTP applications on the KSR1 to use ATMI to access and update data in a proprietary MVC/CICS environment, using LU6.2, SNA Link, VTAM and NCP. CICS application programs need only minimal modification to be used as /T devices.

In /T, facilities for the operation, administration and maintenance of production OLTP applications begin with application definition and configuration management and go on to include dynamic reconfiguration of servers, automatic recovery, application security, data-dependent routing and load-balancing among servers.

In addition to the performance gains of an underlying DBMS optimized for the KSR1 system, /T itself takes advantage of the parallelism of the system. Multiple service requests from the same client can execute in parallel, with service processes allocated to different processors. Requests with the highest priorities are serviced before those of lower priorities. Kendall Square has enhanced the central /T Bulletin Board as well for parallel performance.

TUXEDO/D is a transaction-oriented access method that provides secure, concurrent access to files, optionally establishing records and fields within files. To ensure database integrity, it provides two phase locking, atomic commit, logging and recovery. Although used primarily as a file access method on the KSR1, /D supports interactive SQL via a command interpreter interface and embedded SQL for the C programming language.

File types used by /D include hashed files, heaps, FIFO (sequential) files and indexed files. Two types of indices are available: B-tree files and inverted indices.

COBOL

The KSR COBOL compiler is based on MicroFocus COBOL/2 1.2, a broadly-used and complete language for business application development, compatible with ANSI standards and IBM COBOL. It includes development tools, conversion tools, utilities, and a run-time library including C-ISAM support for COBOL INDEXED files.

The MicroFocus COBOL/2 product set consists of several core tools for program compilation and execution, and an assortment of development tools and utilities which are used to ease the process of COBOL development and to aid in porting applications. For developers currently using MicroFocus COBOL on other platforms, the COBOL environment on the KSR1 is familiar. Developers using other versions of COBOL will find MicroFocus COBOL on the KSR1 to be a complete development environment.

The MicroFocus COBOL/2 compiler (version 1.2) is a certified, high-level ANSI COBOL85 compiler, which also complies with the X/Open standard. It has also been certified at high-level ANSI 74 and can be used at this level through a compiler switch. The compiler is source-code-compatible with several other dialects of COBOL, including IBM OS/VS, IBM VS COBOL II, Ryan-McFarland RM/COBOL and Data General Interactive COBOL V1.2.

The output of the compiler is intermediate code whose format is compatible with COBOL/2 on any platform. This intermediate code can be executed by the COBOL/2 Run Time System and is used by ANIMATOR, the COBOL/2 source level debugger. The compiler intermediate-code output also serves as the input to the Native Code Generator (NCG).

The COBOL/2 NCG is essentially the second pass of the compiler and produces highly-optimized object code for the KSR1 processor. Output of the NCG is an assembler code representation of the COBOL program, which is then code-scheduled and assembled to an executable object.

The Dialog System productivity tool aids developers to produce screen layouts and keyboard dialogs. In addition, two different generators (Forms-2 and SCREENS) assist interactive creation and testing of screen-based applications. Both automatically generate screen-handling code.

PART FIVE

ARCHITECTURE AND THEORY OF OPERATION

OVERVIEW

The KSR1 is a highly parallel computer system scalable from 8 to 1,088 processors (and more in future generations), which is programmed as a shared memory multi-processor. Kendall Square Research's patented ALLCACHE memory system is the key that provides the user with the high performance, programming simplicity and familiarity associated with conventional shared memory computers.

The major phyla of highly parallel MIMD computers are distinguished by their basic computational models: shared memory or message passing systems. As the computing model for a general purpose computing resource, shared memory offers important advantages:

- Shared memory provides a higher performing and more flexible means of communicating between threads executing on different processors. Shared memory is a more general communications model than message passing because shared memory can be used as a means of exchanging messages (by reading and writing a shared message buffer) or directly as a shared respository. In general, exchanging messages in shared memory will be faster than the mechanisms in message passing systems because no system software is involved. In addition, the shared memory can be used directly. For example, one thread can read the values of array elements written by another thread. Experience has shown that direct use of shared memory is often the most straightforward way to adapt previously serial codes to run on parallel systems.

- Shared memory allows the use of conventional methods of memory management. General purpose machines used as shared computing resources must provide run-time mechanisms for mapping program-specific addresses into system addresses. For 25 years these mechanisms have been refined in the context of shared memory and they work effectively. Today there are no effective means for providing global memory management in the context of message passing (e.g., explicitly distributed memory) systems. Hence message passing systems have been employed as single user systems or as partitioned systems — but not general purpose shared computing resources. The current solution to this problem is the creation of a shared memory programming model implemented via software on a message passing hardware base. Such mechanisms are at least two orders of magnitude slower than hardware based solutions like ALLCACHE.

THE KSR1 MEMORY SYSTEM

The KSR1 shared memory programming model is made possible by a patented new architectural technique called ALLCACHE. It does for distributed memory what virtual memory did for hierarchical memory — it replaces the complexity and rigidity of the physical mechanism with a uniform address space, now shared by a set of processors. As in a conventional system, hardware and software maps this space into physical devices. The KSR1 ALLCACHE hardware manages this address space across physically distributed memory, and achieves this programming simplicity without sacrificing the major benefit of distributed memory — scalability. (See Table 1, Part Two, page 10.)

The memory models of earlier MPP computer architectures raised problems for programmers reminiscent of the difficulties of storage management in the 1960s.

Three decades ago, storage management via overlay structures was an integral part of the job of writing a program. Of necessity, programmers attacked the task with a static analysis of the memory requirements of a single program.

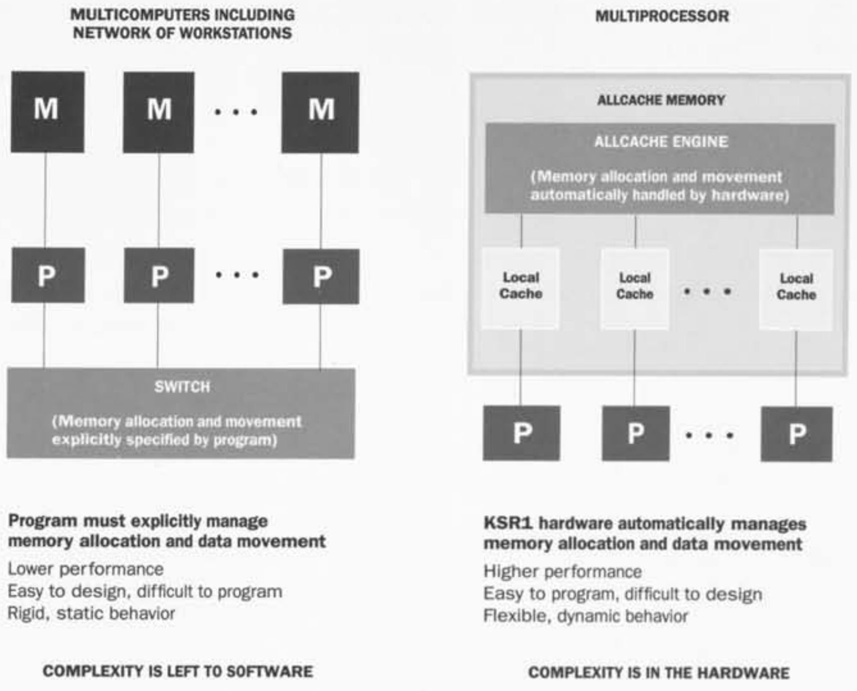
Advances in programming practice and system architectures, however, gradually rendered static storage management impractical. The goals of machine independence and re-use of modular program elements, and the use of very complex algorithms characterized by data structures of widely varying size and shape were inconsistent with static, programmer-controlled storage management. In addition, the introduction of system environments in which computers were organized for simultaneous use by several programs made it impossible for the author of a single program to predict accurately the time-varying storage requirements of the entire system.

These factors ultimately led to the adoption of virtual memory as a near-universal feature of storage management in modern computer architectures. Virtual memory makes storage management dynamic and largely automatic. It permits programmers to write applications with a storage abstraction that is simple and powerful — a single uniform address space. System hardware and software map this space into physical devices. Other parallel processing architectures reprise these early storage management issues with a new twist. All of the MPP systems that have been introduced have distributed memories. That is, the physical memory comprises a set of memory units, each connected to a unique processor. The processor-memory pairs are interconnected by a network. Distributed memories have been universal among massively parallel machines because they provide the only known means of implementing completely scalable access to memory — access whose bandwidth increases in direct proportion to the number of processors.

In these MPP systems, the task of managing the movement of codes and data among these distributed memory units belongs to the programmer. The job is similar in style to the task of managing the migration of data back and forth between primary and secondary storage prior to the introduction of virtual memory but it is much more complex. As before, programmers need to be concerned about exactly what will fit where and what to remove to make room for something new. Now, however, there are thousands of memory units to deal with instead of just two or three. Parallel systems of this type are "multi-computers" — sets of network connected independent computers.[1] In contrast the KSR1 and all of today's mainframe computers are "multi-processors" – single computers with multiple processors sharing memory symmetrically.

1. Bell, C. Gordon, "Ultracomputers A Teraflop Before Its Time," Communication of the ACM, August 1992/Vol. 35 No. 8.

FIGURE 15 *Multicomputer versus Multiprocessor*


MULTICOMPUTERS INCLUDING NETWORK OF WORKSTATIONS
M
M
M
P
P
P
SWITCH
(Memory allocation and movement explicitly specified by program)
Program must explicitly manage memory allocation and data movement
Lower performance
Easy to design, difficult to program
Rigid, static behavior
COMPLEXITY IS LEFT TO SOFTWARE
MULTIPROCESSOR
ALLCACHE MEMORY
ALLCACHE ENGINE
(Memory allocation and movement automatically handled by hardware)
Local Cache
Local Cache
Local Cache
P
P
P
KSR1 hardware automatically manages memory allocation and data movement
Higher performance
Easy to program, difficult to design
Flexible, dynamic behavior
COMPLEXITY IS IN THE HARDWARE

THE ALLCACHE SOLUTION

ALLCACHE memory system provides programmers with a uniform 2^{64} byte[1] address space for instructions and data. This space is called System Virtual Address space (SVA). The contents of SVA locations are physically stored in a distributed fashion.

ALLCACHE implements a sequentially consistent shared address space programming model because such consistency is the strongest requirement for shared-memory coherence, and this form of implementation guarantees that a program will behave in the most intuitive manner to the programmer: e.g., the result of program execution on a multiprocessor is equivalent to the execution of the program on a single processor with multi-tasking. In this context, the formal definition of "sequential consistency" is:[2]

> "A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

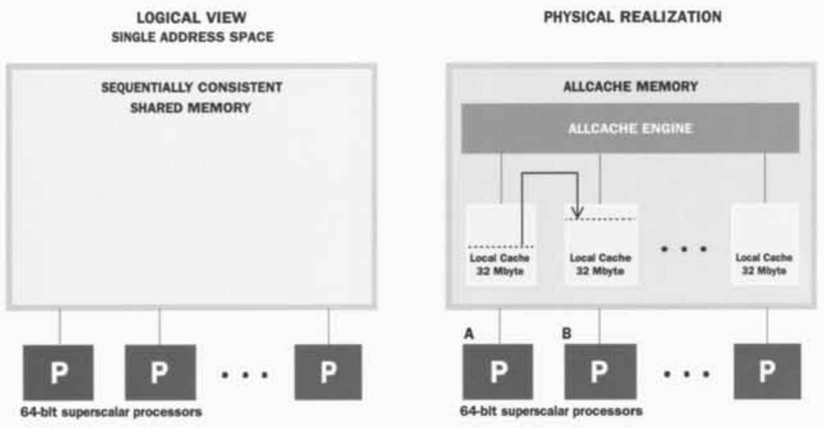
Note that any ordering scheme other than a sequentially consistent programming model inherently requires both the explicit specification of the sharing and a legal time order of access.

1. The KSR1 implements a 2^{40} byte (1 terabyte) address space utilizing 64 bit pointers. Future generations will implement the full 2^{64} byte address space of the ALLCACHE memory architecture.
2. Lamport, Leslie: "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, C-28, No. 9 (September 1979), pps. 690-691.

ALLCACHE physically comprises a set of memory arrays called local caches, each capable of storing 32 MByte. There is one local cache for each processor in the system. Hardware mechanisms (the ALLCACHE Engine described below) cause SVA addresses and their contents to materialize in the local cache of a processor when the address is referenced by that processor. The address and data remain at that local cache until the space is required for something else.

As its name suggests, the ALLCACHE behavior is like that of familiar caches: data moves to the point of reference on demand. However, unlike the typical cache architecture (called "SOME-CACHE" memory), the source for the data which materializes in a local cache is not main memory but rather another local cache. In fact, all of the memory in the machine consists of large, communicating, local caches — the main memory of the machine is identical to the collection of local caches. See Figure 16.

FIGURE 16 *ALLCACHE Memory System*



ALLCACHE: DATA MOVES TO THE POINT OF REFERENCE ON DEMAND. THERE IS NO FIXED PHYSICAL LOCATION FOR AN "ADDRESS" WITHIN ALLCACHE MEMORY.

The address and data that materialize in local cache B in response to a reference by processor B may continue to reside simultaneously in other local caches. Consistency is maintained by distinguishing the type of reference made by processor B:

1. If the data will be modified by B, the local cache will receive the one and only instance of an address and its data.
2. If the data will be read but not modified by B, the local cache will receive a copy of the address and its data.

When processor B first references the address X, ALLCACHE examines that processor's local cache to see if the requested location is already stored there. If processor B's local cache contains address X, the processor request is satisfied without any request to the ALLCACHE Engine. If not, the ALLCACHE Engine hardware locates another local cache (e.g., local cache A) where the address and data exist.

If the processor request being serviced is a read request (for example, to load the value into a register) then the ALLCACHE Engine will copy the address and data from local cache A into local cache B. The amount of data copied will be 128 bytes, called a subpage[1]. At the end of this operation the subpage will reside at both A and B. If the processor request is a write request (for example, to store the contents of a register into this location) then the ALLCACHE Engine will remove the copy of the subpage from local cache A as well as from any other local caches where it may exist before copying it into local cache B. Thus the ALLCACHE Engine is responsible for finding and copying subpages stored in local caches and for maintaining consistency by eliminating old copies when new contents are stored.

In order to maintain consistency, the ALLCACHE Engine records state information about the subpages it has stored. These states are specific to the physical instance of a subpage within a particular local cache. Four states are required to describe the basic operation of the ALLCACHE Engine:[2]

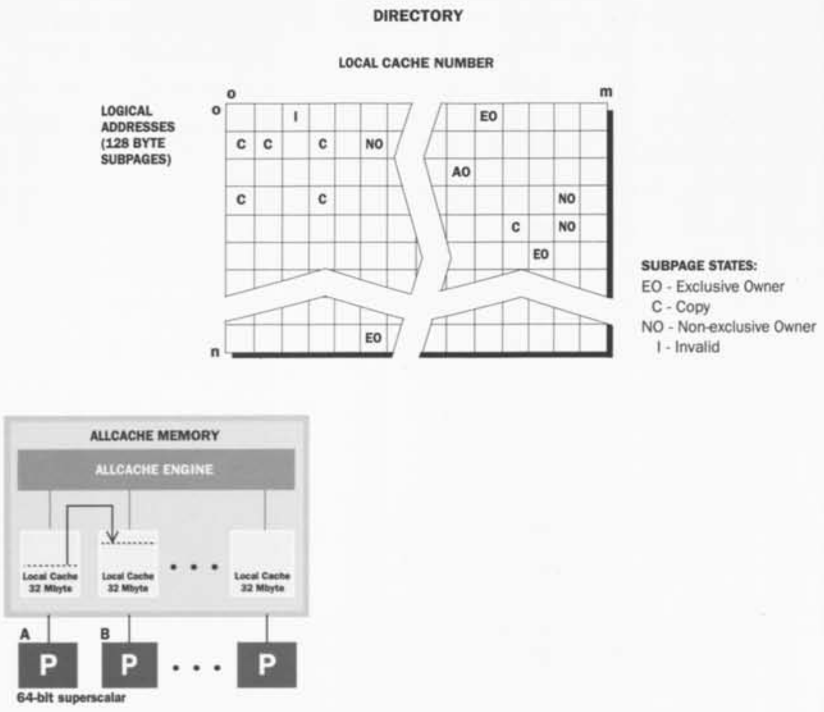
- Exclusive (owner): This is the only valid copy of the subpage in the set of local caches.
- Copy: Two or more valid copies of the subpage exist among the set of local caches.
- Non-exclusive (owner): When multiple copies exist, one copy is always flagged as the non-exclusive owner.
- Invalid: Memory is currently allocated for this subpage at this local cache but the contents are not valid and will not be used.

None of these states are explicitly visible to the programmer. They are used as internal bookkeeping by the ALLCACHE Engine.

The ALLCACHE Engine manages a directory that determines which one or more local caches contain an instance of each subpage. This directory is physically stored in a distributed and compressed form but its logical function is illustrated in Figure 17 on the following page. The directory is logically a matrix consisting of a row for each subpage and a column for each local cache. Each entry in the matrix is either empty, to indicate that the corresponding subpage is not present in the local cache, or it contains a "state" designator. A non-empty state designator means that a spot for a copy of this subpage is currently allocated in the corresponding local cache, and the state value indicates what operations the memory system is allowed to perform on the particular copy. This matrix is a very sparse representation of the mapping, because nearly all elements will be empty. The ALLCACHE Engine implementation actually stores this matrix by column and compresses out all of the empty elements. More details of the ALLCACHE Engine implementation will be presented later in this section.

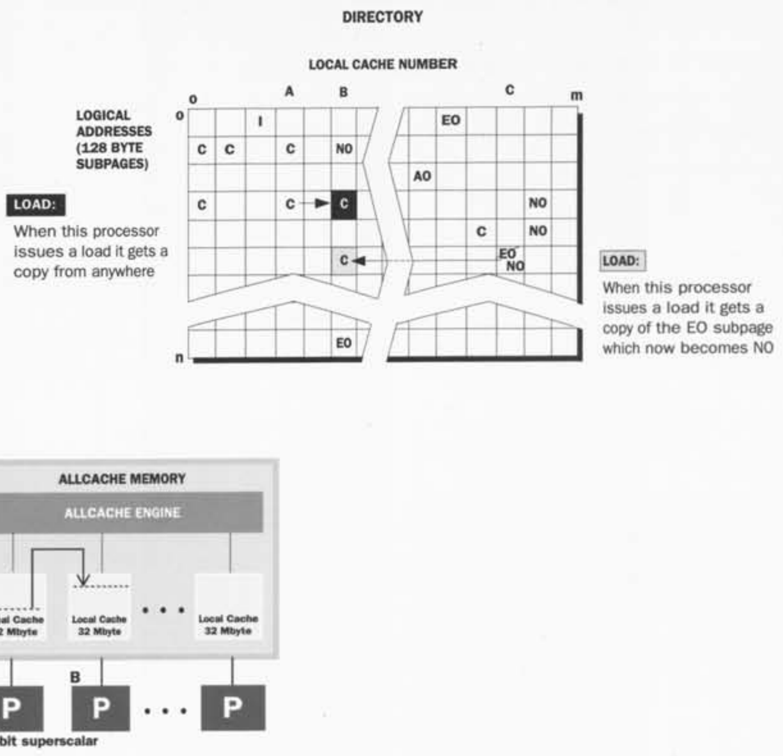
1. ALLCACHE stores data in units of pages and subpages. Pages contain 16 KB (2^{14} bytes), divided into 128 subpages of 128 (2^7) bytes. The unit of allocation in local caches is a page, and each page of SVA space is either entirely represented in the system or not represented at all. The unit of transfer and sharing between local caches is a subpage. Each local cache has room for 2,048 (2^{11}) pages or a total of 32 MByte (2^{25} bytes). When a processor references an address not found in its local cache, ALLCACHE memory first makes room for it there by allocating a page. The contents of the newly allocated page are filled as needed, one subpage at a time.
2. Additional states, including, Atomic state (see Synchronization Primitives below) are actually used in the implementation of ALLCACHE.

FIGURE 17 *Logical Organization of ALLCACHE Engine Directory*



The ALLCACHE Engine manipulates the directory in response to load and store instructions from processors. For a load instruction, if a copy exists in the requesting processor's local cache, the load request can be satisfied without any interaction at all with the ALLCACHE Engine. If a copy does not exist in the local cache, the local cache sends a request packet to the ALLCACHE Engine. The ALLCACHE Engine then delivers a response packet containing a copy from any other local cache which has a valid copy, as illustrated in Figure 18 on the following page. For example, Processor B issues a request for a copy to the ALLCACHE Engine. The ALLCACHE Engine routes the request to a local cache in which a copy exists (for instance, the local cache associated with Processor A). Local cache A responds to the ALLCACHE Engine, which in turn passes the copy back to the local cache of the requesting processor and automatically updates the ALLCACHE directory and the local cache directory to indicate that a copy of the subpage now exists in the local cache associated with Processor B. Had the subpage been in Exclusive (owner) state within some local cache (C, for example) at the time of the reference, the ALLCACHE Engine would create the requested copy and change the owner's state to Non-exclusive (owner).

FIGURE 18 *ALLCACHE Engine Operations - LOAD Instructions*

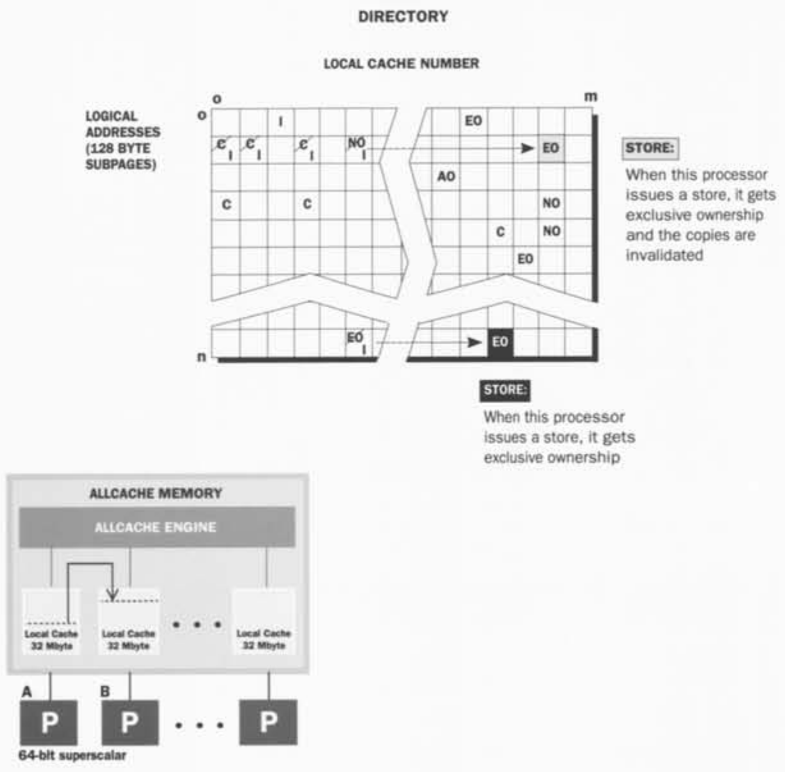


DIRECTORY
LOCAL CACHE NUMBER
o
A
B
C
m
LOGICAL ADDRESSES (128 BYTE SUBPAGES)
o
n
I
EO
C
C
C
NO
AO
C
C
C
NO
C
NO
C
EO
NO
EO
LOAD:
When this processor issues a load it gets a copy from anywhere
LOAD:
When this processor issues a load it gets a copy of the EO subpage which now becomes NO
ALLCACHE MEMORY
ALLCACHE ENGINE
Local Cache 32 Mbyte
Local Cache 32 Mbyte
Local Cache 32 Mbyte
A
B
P
P
P
64-bit superscalar

Note that the program that issues the load or store has no knowledge of the respective physical locations of the local caches. The ALLCACHE Engine transparently manages the routing, based on the subpage address and the directory. Memory allocation is handled by the respective local caches.

For a store instruction, if a subpage exists in the local cache in Exclusive (owner) state at the time of the request, the store can be satisfied locally without any interaction with the ALLCACHE Engine. If the requestor's local cache state is empty or invalid, the ALLCACHE Engine will move the subpage from the Exclusive (owner) to the requestor in Exclusive (owner) state (Figure 19). In cases where multiple copies are involved, the effect is for the ALLCACHE Engine to move the ownership to the requestor's local cache in exclusive owner state and to invalidate all other copies. The ALLCACHE Engine moves ownership to the requestor and invalidates all other copies (to make the new ownership exclusive) in a single operation.

FIGURE 19 *ALLCACHE Engine Operations - Multiple Copies*


DIRECTORY
LOCAL CACHE NUMBER
o
m
LOGICAL ADDRESSES (128 BYTE SUBPAGES)
o
n
I
EO
C
NO
EO
AO
NO
NO
EO
STORE:
When this processor issues a store, it gets exclusive ownership and the copies are invalidated
STORE:
When this processor issues a store, it gets exclusive ownership
ALLCACHE MEMORY
ALLCACHE ENGINE
Local Cache 32 Mbyte
Local Cache 32 Mbyte
Local Cache 32 Mbyte
A
B
P
P
P
64-bit superscalar

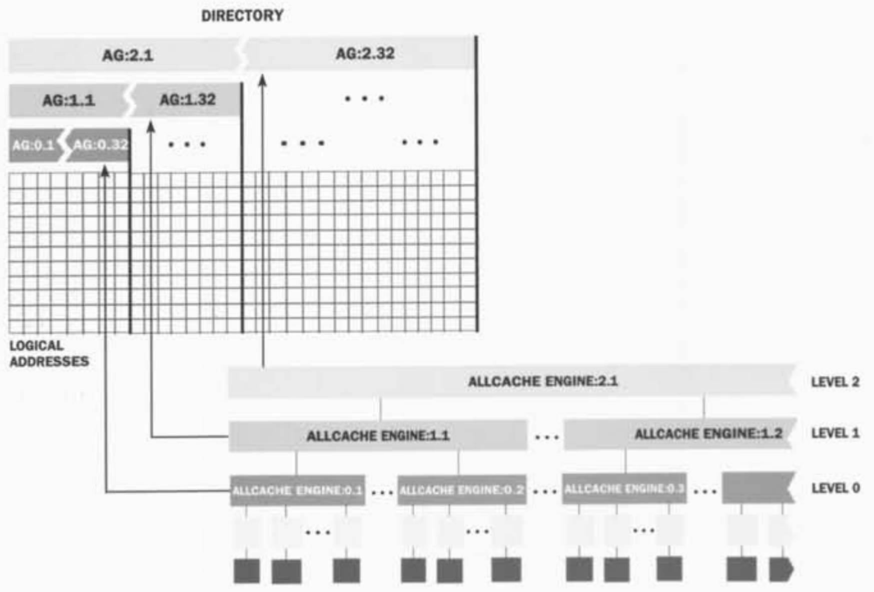
HIERARCHICAL ORGANIZATION OF THE ALLCACHE ENGINE

The KSR1 architecture exploits locality of reference by organizing a number of ALLCACHE Engines in a hierarchy (Figure 20). At the lowest and most heavily populated level of this hierarchy are ALLCACHE Group:0s (AG:0s), each of which is the combination of ALLCACHE Engine:0s and the complete set of local caches associated with them.

At the next level of the hierarchy, the family of all AG:0s, combined with their associated ALLCACHE Engine:1s, are the ALLCACHE Group:1s (AG:1s) and so on, to a potentially unlimited number of levels.

An ALLCACHE Engine:0 includes the directory which maps from addresses into the set of local caches within its group. An ALLCACHE Engine:1 includes the directory which maps from addresses into its constituent set of ALLCACHE Group:0s. Higher level ALLCACHE Groups are hierarchically constructed in the same manner..

FIGURE 20 *Hierarchical Organization of the ALLCACHE Engine*



The initial KSR1 system implements two levels of ALLCACHE Engine hierarchy. The ALLCACHE Engine is constructed with a fat-tree[1] topology[2], so that the bandwidth increases at each level of ALLCACHE Engine (the next section will further discuss the theoretical aspects of a fat-tree as they apply to the ALLCACHE Engine). For the KSR1, ALLCACHE Engine:0 has a bandwidth of 1 GB/sec and ALLCACHE Engine:1 has a bandwidth of 1, 2 or 4 GB/sec. For example a KSR1-1088 consists of 34 ALLCACHE Group:0s, each consisting of 32 processors and their associated local caches. As we shall see, due to locality of reference, the effective ALLCACHE Engine bandwidth is asymptotic to the aggregate ALLCACHE Engine:0 bandwidth of 34 GB/sec.

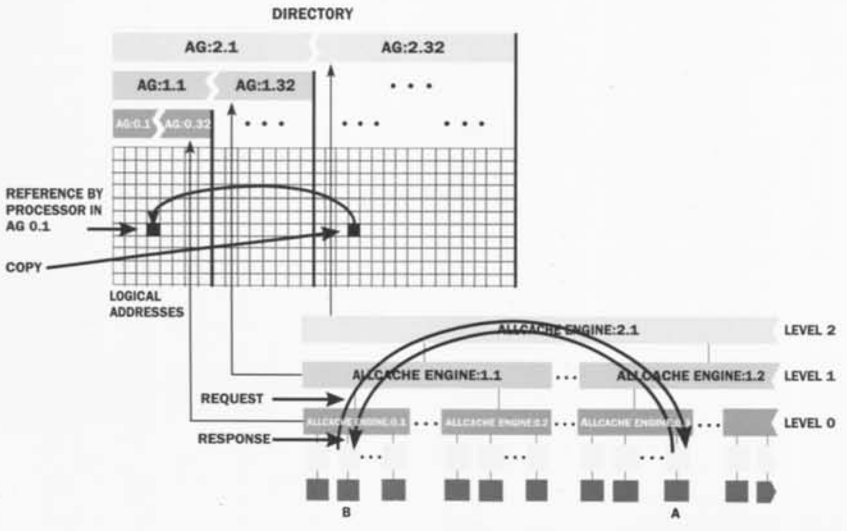
1. Leiserson, Charles E. "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing;" IEEE Transactions on Computers, Vol. C-34, No. 10, pps. 892-901, October, 1985.
2. See Part Five, section on scalability for a discussion of fat-tree topology.

The hierarchical ALLCACHE Engine handles simultaneous independent requests and simultaneous requests to the same address in parallel.

Figure 21 illustrates the path of a request and response through the hierarchy of ALLCACHE Engines. A request initiated at a processor will move up through the levels of the hierarchy until it reaches an ALLCACHE Group which contains a directory entry in the appropriate state for the desired subpage address. The request then moves down through the levels of the hierarchy to the location of the subpage. The response reverses this path to return to the requestor.

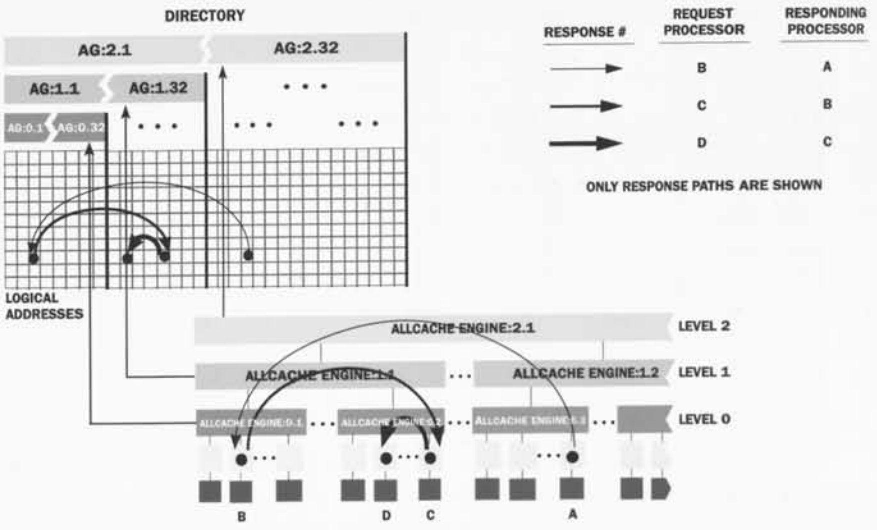
For example, consider a request initiated at processor cell B, for a subpage which hierarchically first appears in the directory at ALLCACHE Engine:2.1. The request is first moved into ALLCACHE Engine:0.1 where the address is not found. It is then moved on to ALLCACHE Engine:1.1 where the address is not found either. Finally the request is routed to ALLCACHE Engine:2.1, where the address is found to be in ALLCACHE Group:1.2. It is then routed to ALLCACHE Engine:1.2 which finds that the address is in ALLCACHE Engine:0.3. The request packet is then routed to ALLCACHE Engine:0.3 which routes it to the local cache at processor A. The maximum length of the request path is proportional to the log of the number of processors.

FIGURE 21 *Hierarchical Organization of the ALLCACHE Engine — Search Path Through the Hierarchy*



A crucial characteristic of the hierarchical structure is that it allows the KSR1 to exploit hierarchical locality of reference. The hierarchical structure of the ALLCACHE Engine exploits this characteristic by moving referenced subpages to a local cache and by satisfying data references from nearby copies of a subpage whenever possible. In the example in Figure 22 on the following page, the first reference by processor B to the subpage in processor A needs to travel through ALLCACHE Engine:2.1 to find the designated subpage. The second reference to the same subpage by processor C finds the data closer as does a subsequent reference to the same subpage by processor D.

FIGURE 22 *Hierarchical Organization of the ALLCACHE Engine — Exploits Hierarchical Locality of Reference*



Locality — The Key to Scalability

While the fat-tree topology of the KSR1 ensures maximization of bandwidth, the inherent ability of the ALLCACHE memory system to exploit locality of reference achieves the second major goal of scalability — reduction of the bandwidth requirement itself.

Locality is the key to the achievement of a scalable interconnect bandwidth in which the bandwidth requirement itself scales more slowly than the delivered bandwidth. Three reasons may be cited:

- Because communication speeds are fundamentally limited by the speed of light, communications should be kept as close as possible to the processor.
- Communication time is also affected by the number of switches through which messages or data must pass. Thus path lengths should be minimized.
- Communication should stay within as small a subsystem as possible to avoid congestion.[1]

The ALLCACHE Engine exploits locality – both the usual serial locality of reference and its image in parallel programs, parallel locality.

Locality of reference refers to a property of a program in which near future memory references are likely to reference memory locations nearby the addresses of recent past references. The most important memory architecture innovations of the last thirty years, virtual memory and cache memories, are designed to exploit this program behavior. The phenomenon is so pronounced in most programs that even small caches with a few tens of kilobytes of memory will exhibit hit rates of over 98%.

1. Leiserson, Charles E. "VLSI Theory and Parallel Supercomputing;" Pasadena, CA: Proceedings of the 1989 Decennial Caltech Conference, March, 1989.

Parallel locality refers to a related property of parallel programs. The best predictor of future memory references by a thread of a parallel program is that thread's own recent memory reference pattern – in other words, the usual serial locality of reference applies to the serial pieces of a parallel program. But the next best predictor of future memory references is the recent memory reference pattern of related threads. This phenomenon of common reference patterns for related threads is called parallel locality.

Both serial locality of reference and parallel locality are exploited by the ALLCACHE memory system. A KSR1 has a large cache, 32 MByte, designed to exploit serial locality of reference. The hierarchical structure of the ALLCACHE Engine, combined with the scheduling algorithms of KSR OS, provide the means to exploit parallel locality. The KSR scheduler will allocate a set of related threads to execute in the same ALLCACHE Engine:0 whenever possible. Thus, each thread of a parallel program gains a benefit from the parallel memory referencing activity of related threads: an address not found in a thread's local cache is likely to be found in the same branch of the ALLCACHE hierarchy no matter how many other branches there may be. Communication will then stay within as small a subsystem as possible, avoiding congestion. Although a fat-tree can deliver scalable bandwidth, ALLCACHE does not require that the bisection bandwidth scale in linear fashion to keep step with the number of processors.

Both types of locality are usually present in programs to a substantial extent without any effort on the part of the programmer. Programmers can increase locality by careful design of data structures and processing flow, much as they do in writing certain programs for virtual memory machines. KSR compilers and the KSR OS use a number of techniques to automatically increase locality.

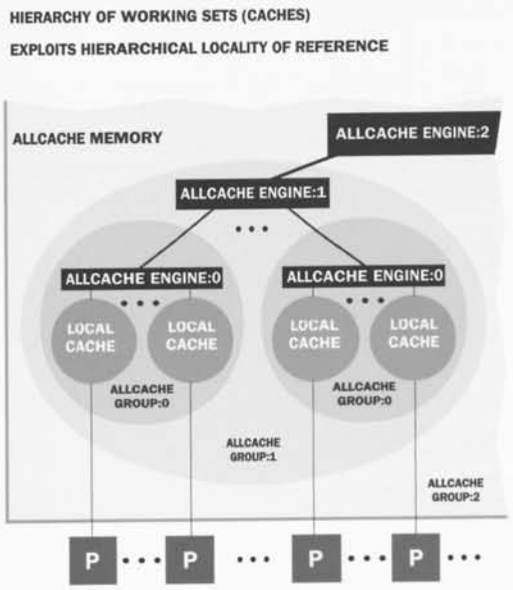
Another way to look at this phenomenon is as a hierarchy of "working sets." For each processor, the addresses most likely to require reference lie in the closest and smallest working set, which is realized in the local cache of that processor. The next most likely addresses to be referenced lie in the ALLCACHE Group:0 (AG:0) working set, which is realized as the aggregate of the local caches of the AG:0.

Taken together, the local caches of all processors in a given ALLCACHE Group, e.g., "AG:N," form an AG:N cache, which holds the working set for that "AG:N." Processors with an AG:N share addresses without any communication outside their own ALLCACHE Group. Thus the hierarchical nature of the ALLCACHE Engines and ALLCACHE Groups allows the distributed local caches to form, collectively, a hierarchy of caches corresponding to the hierarchy of AGs. The KSR1 implements two levels of ALLCACHE Groups:

Level of Hierarchy	Working Set Size
Local Cache	32 MByte
AG:0 Cache	1 GByte
AG:1 Cache	34 GByte

Figure 23 provides an illustration of the hierarchical organization of the ALLCACHE Engine.

FIGURE 23 *Hierarchical Organization of ALLCACHE Engine*



OPTIMIZING LOCALITY OF REFERENCE

Locality of reference is present in programs to a substantial extent, usually without any conscious effort on the part of the programmer. Programmers can increase locality of reference by optimizing memory-reference patterns with this property in mind. KSR compilers use a number of techniques to increase locality of reference automatically.

The KSR1 also incorporates a number of features designed to assist programmers in their efforts to optimize locality of reference on a customized basis. Each is described below:

Event Monitor Unit (EMU)

Each KSR1 processor contains an event monitor unit (EMU) designed to log various types of local memory events and intervals. The job of the EMU is to count events and elapsed time related to memory system activities that are not otherwise directly visible to the processor.

The types of events that are logged include local cache hits/misses and how far in the hierarchy a request had to travel to be satisfied. The EMU also accumulates the number of processor cycles involved in such events. These counters can be read at the appropriate points in the application code, to help characterize loop nests or other sections of code. Since the events to be logged are counted by hardware, the measurement overhead is extremely low. Because the events are monitored on an individual-processor basis, an extremely clear picture can be created to facilitate the customized parallelization of applications and the optimizing of locality of reference.

Prefetch

Prefetch is an instruction that allows memory activity to go on in parallel with computation, by planning for data needs in advance rather than stalling the processor to wait for needed data. The prefetch instruction requests the memory system to move a subpage into the local cache of the requesting processor, thus allowing the memory system to fetch data before it is needed. The processor that issues the prefetch instruction does not need to wait for this operation to be completed; it continues executing until it needs to load or store the prefetched address. If the prefetch is issued far enough in advance, the desired address will have already arrived in the local cache, and minimum latency will be incurred in accessing it. KSR1 compilers automatically insert prefetches in certain types of code sequences. Programmers may also request prefetches explicitly by means of an intrinsic function.

Poststore

Any program that executes a store can use the poststore instruction to ask the memory system to broadcast the new value to other local caches that may need it. Local caches in which the corresponding page is allocated already (and in which the subpage state is, necessarily, invalid) take a read-only copy of the subpage. Poststore instructions allow a processor to broadcast data needed by one or more other processors at the earliest possible time the data is available, and before the other processors have to request the data.

Like prefetch, poststore is controlled by the processor that writes the data. Programmers can explicitly request poststores with an intrinsic function.

The Benefits of Locality

The KSR1 ALLCACHE Groups form a hierarchy of cache working sets, with increasing magnitude at each succeeding level. The ability to hold a larger working set at each level effectively reduces the bandwidth requirement of communications to the next higher level, by satisfying most references without the need to communicate with the next-higher AG.

From the point of view of the requesting processor, the possible locations of the desired subpage will fall into just four categories:

- The processor subcache (see below),
- The local cache,
- A cache on the same ALLCACHE Engine:0,
- Or a cache on a different ALLCACHE Engine:0.

Within each of these categories, the cache-refill times are always identical.

For some operations (those requiring read-only data) only the location of the closest cache containing the data is relevant. Because any copy of the subpage will satisfy the request, ALLCACHE memory automatically chooses the closest copy. For other operations (such as a store instruction) which require a subpage in exclusive state, the location of the farthest local cache containing a copy of the subpage is relevant. To put one copy of the subpage into exclusive state, all other read-only copies must be marked invalid, so the farthest of these determines the total cache-refill time of the operation.

Processor Subcache

One additional element incorporated in the design of the ALLCACHE memory system remains to be considered: each processor contains a 0.5 MByte memory array, called a subcache, which always contains a subset of the addresses and data stored in that processor's local cache. Half of each subcache is used for data and half for instructions. This is the only level of the memory hierarchy that distinguishes between instructions and data in storage. Subcaches are managed entirely by hardware. Even though a processor's subcache is small compared with the rest of memory, the great majority of memory references can be satisfied from the subcache. This is another consequence of locality of reference. Table 3 provides the approximate cache-refill times, measured in clock cycles, involved in completing memory system operations, as a function of the type of operation and the location of the required data.

TABLE 3 *Cache Fill Times*

Operation Required	Which Cache is Relevant?	Working Set	Working Set Size	Cache-Refill Time (cycles)	Cache-Refill Time With Prefetch
Load (copy)	Closest	Local subcache	.5 Mbyte	2	2
Load (copy)	Closest	Local cache	32 Mbyte	20	20
Load (copy)	Closest	Same AG:0	1 Gbyte	150	20
Load (copy)	Closest	Different AG:0	34 Gbyte	570	20
Store (exclusive)	Farthest	Local subcache	.5 Mbyte	0	0
Store (exclusive)	Farthest	Local cache	32 Mbyte	20	20
Store (exclusive)	Farthest	Same AG:0	1 Gbyte	150	20
Store (exclusive)	Farthest	Different AG:0	34 Gbyte	570	20
Any	NA	Not in any cache — page fault		≈ 400,000[a]	

a. Conventional virtual memory page fault time, which corresponds to disk access time.

SYNCHRONIZATION PRIMITIVES

There are times when two or more processors need to synchronize their access to memory locations. The ALLCACHE memory system supports this requirement through instructions which lock and unlock subpages. These instructions can be used to implement any multi-processor synchronization function including data locks, barriers, critical regions and condition variables. (All of these forms of synchronization and others are available via KSR compilers, libraries and OS calls.)

The development of the KSR1 synchronization primitives was influenced by two primary considerations:

1. *Scalability.* To support a large number of cooperating processors, the KSR1 is designed to provide extremely efficient access to concurrent data structures.

   The KSR1 supports this efficient access by means of fine grained (subpage) data locking. This technique may be viewed in contrast with critical section synchronization, in which a single lock prevents more than one processor from executing the "critical section" of the code between the locks. An alternative version of the critical section approach involves the use of a single lock to control concurrent access to all elements of a data structure. With either version of the critical section synchronization technique, concurrent accesses to different elements of the data structure are serialized. A large portion of contention in past parallel systems has been artificial, a result of the "false contention" that is a by-product of these critical section synchronization techniques.

   By comparison, with synchronization by means of fine grained data locking, concurrent access to different elements of the data structure can occur in parallel. Fine grained data locking enables a programming model that minimizes contention. Fine grained data locking lowers the probability of contention, because concurrency for each element of the data structure is managed independently. Contention only results when multiple processors must gain atomic access to the same element(s) of the data structure.

2. *A single set of high performance primitives made directly available to user applications in an unprivileged manner.* The KSR1 synchronization primitives execute with low overhead when contention is not present, as well as when it is. The no-contention case is especially significant, because most accesses to individual elements using fine grained data locking are contention free. Thus the application only pays for the overhead of synchronization for concurrent access when contention actually exists.

   Synchronization and data movement are achieved in parallel, in a single operation.

   Because the synchronization primitives are part of the memory system state, locked data structures can migrate transparently between processors or processes. Locking state is recorded as part of the backing store state.

   Complex atomic updates, such as fetch-and-add or linked-list queue manipulation, can be built by using the basic primitives.

The underlying memory-system primitives that make possible the higher level abstractions are the following:

Setting locks. A "lock" in ALLCACHE is achieved by setting a subpage to the Atomic state. (Like Exclusive state, Atomic state indicates the subpage is the only valid instance of this address in any local cache. But Atomic state also provides a flag that allows multiple processors to synchronize their access to a subpage.) A program may do this by issuing a *get* or *get-and-wait* instruction specifying the address of the desired subpage. Both instructions will cause the ALLCACHE Engine to find the subpage and — if the page is not in Atomic state — return it to the requesting processor in Atomic state. In the process the ALLCACHE Engine will ensure that all other copies of the subpage are set Invalid.

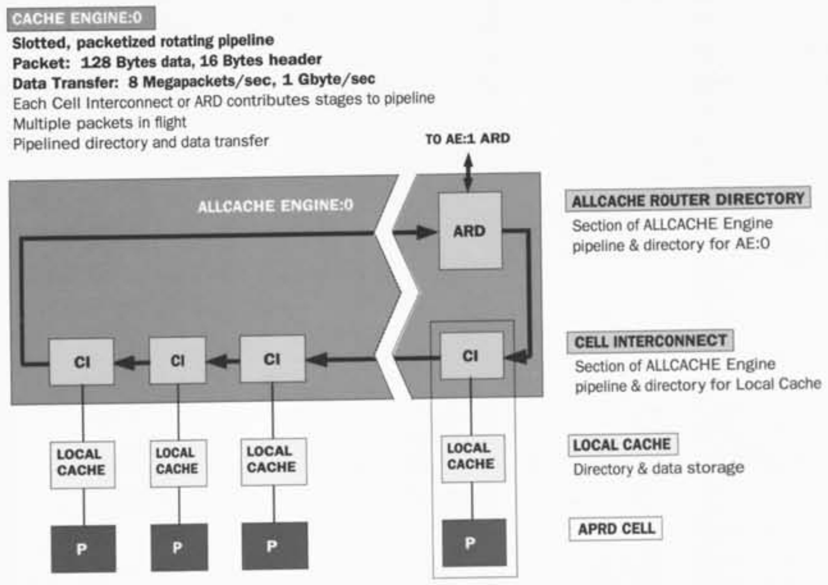
If the subpage is already in Atomic state, it will not be returned to the requestor immediately. Instead, the request will return to the requestor with an indicator that the subpage was found in the Atomic state. At this point, if the instruction involved was *get*, the ALLCACHE memory system returns an *already* Atomic condition to the processor and the issuing program will decide what to do. The software might try again immediately or go on to something else and try again later. If the instruction involved was *get-and-wait*, the processor will stall until the requested subpage arrives. (As a rule, *get-and-wait* is used for locks which are expected to be held very briefly.)

Releasing locks. A program removes Atomic state from a subpage by issuing the *release* instruction. When this instruction is issued, the subpage state is changed to Exclusive. If a *get-and-wait* has attempted to access this subpage while it was in the Atomic state, the effect of the release instruction will be to send the newly released subpage into the ALLCACHE Engine, through which it will be routed to any waiting processor(s).

INSIDE THE ALLCACHE ENGINE

The assignment of the ALLCACHE Engine is to resolve references to addresses that are not found in the requestor's local cache (or are found there but not with the proper subpage state). The ALLCACHE Engine must find the indicated address in the set of local caches and return it to the requestor's local cache in an appropriate state. The ALLCACHE Engine also must leave the subpage states of other local caches consistent with the state returned to the requestor. For example, if the subpage is returned to the requestor in Exclusive (owner) state, that subpage must be in Invalid state in all other local caches where it is allocated.

FIGURE 24 *ALLCACHE Engine is a Slotted, Pipelined Packet Router (ALLCACHE Engine:0 Shown)*


CACHE ENGINE:0
Slotted, packetized rotating pipeline
Packet: 128 Bytes data, 16 Bytes header
Data Transfer: 8 Megapackets/sec, 1 Gbyte/sec
Each Cell Interconnect or ARD contributes stages to pipeline
Multiple packets in flight
Pipelined directory and data transfer
TO AE:1 ARD
ALLCACHE ENGINE:0
ARD
CI
CI
CI
CI
LOCAL CACHE
LOCAL CACHE
LOCAL CACHE
LOCAL CACHE
P
P
P
P
ALLCACHE ROUTER DIRECTORY
Section of ALLCACHE Engine pipeline & directory for AE:0
CELL INTERCONNECT
Section of ALLCACHE Engine pipeline & directory for Local Cache
LOCAL CACHE
Directory & data storage
APRD CELL

The ALLCACHE Engine:0 (see Figure 24) is a series of point-to-point connections between the APRD (see below for more detailed information) cell interconnect and the ARD, used to interconnect hierarchies of the ALLCACHE Engines (see below). The ALLCACHE Engine pipeline provides a parallel pipelined directory and transport mechanism. Each APRD cell interconnect consists of a local cache directory and stages of the ALLCACHE Engine pipeline. The cell interconnect imposes a logical structure of rotating slots. Each slot contains a packet consisting of a 16 byte header and a 128 byte subpage of data. As the number of APRDs or ARDs increases, the number of slots, packet carrying capacity and directory lookup parallelism also increases. The point to point nature makes it component and technology scalable from an electrical point of view.

An ALLCACHE Engine:0 handles a request in the following manner. If the local cache cannot satisfy a local processor request, a request packet is inserted into the pipeline. As the request packet passes each Cell Interconnect, that Cell Interconnect checks to see if the subpage is present in its local cache. If the Cell Interconnect has a copy of the subpage in the appropriate state, it extracts the request and then inserts a response onto the pipeline. The response then travels to the original requesting APRD, where the packet is extracted. As the packet is traveling around the pipeline, the directories of each local cache are updated appropriately.

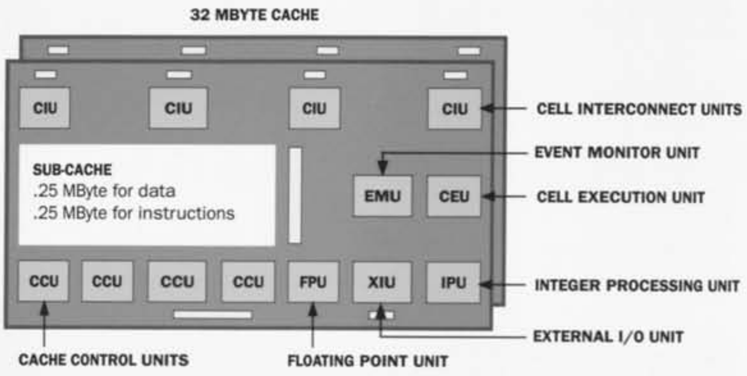
The ARD cell is responsible for connections to the next higher level of the ALLCACHE Engine hierarchy. An ARD cell contains a portion of the ALLCACHE Engine pipeline and a directory with an entry for every subpage allocated on every local cache throughout the entirety of the ALLCACHE Engine:0. When a request packet reaches an ARD, it is moved to the next cell on ALLCACHE Engine:0 if the directory in the ARD indicates that the requested subpage is within

ALLCACHE Engine:0. Otherwise, the packet is routed up to the next higher level in the ALLCACHE Engine hierarchy.

The ALLCACHE Engine:1 consists entirely of ARDs and operates in the same manner as an ALLCACHE Engine:0.

THE KSR1 PROCESSOR

FIGURE 25 *KSR1 APRD Cell*



2 INSTRUCTIONS PER CYCLE

HIGH PERFORMANCE; 64 BIT
40 peak MIPs,
40 peak MFLOPs,
28 MFLOPs (FFT),
32 MFLOPs (Matrix Multiply)

PIPELINED HARDWIRED INSTRUCTION PROCESSING

BUILT IN EVENT MONITOR UNIT

IEEE STANDARD FLOATING POINT FORMAT

1 GBYTE/SEC PROCESSOR AND MEMORY INTERCONNECT

30 MYBTEs/SEC STANDARD I/O CHANNEL

20cm BY 32cm (8 INCH BY 13 INCH) BOARD

The KSR1 processor is implemented in four 1.2 micron CMOS chips.

One of these chips, called the Cell Execution Unit or CEU, is the basic control unit of the processor. On each clock cycle it fetches two instructions from memory. Certain instructions (loads, stores, branches, address arithmetic) will be executed directly by the CEU; others will be executed by a co-processor for execution. The CEU is responsible for all instructions dealing with memory. These instructions operate on 40 bit addresses. The KSR1 architecture actually envisions a 64 bit address (pointers are stored as 64 bit quantities) but, due to implementation constraints, the current address size is 40 bits — and that is clearly sufficient for 1088 processor systems being built at this time. The CEU has 32 address registers, each 40 bits wide.

The CEU operates with three co-execution units.

The FPU (floating point unit) executes arithmetic operations on IEEE floating point format values. The FPU has 64 registers each 64 bits wide, supporting linked triad instructions in which two floating point operations are initiated from a single instruction, giving a peak floating point rate of 40 MFLOPS. Sustained floating point performance depends on the application. For example: 6.6 MFLOPS (Livermore Loops harmonic mean); 15 MFLOPS (100 x 100 LINPACK); 28 MFLOPS (FFT); and 32 MFLOPS (Matrix Multiply).

IPU (integer and logical operations unit) — This chip performs arithmetic and logical operations on 64 bit integers stored in 32 registers (each 64 bits wide).

XIU (external I/O unit) — This chip provides a 30 MBytes/sec pathway to peripheral devices. Since there is an XIU on every APRD cell, large systems can be configured with very high aggregate bandwidth to disk drives, networks, display devices and other peripherals.

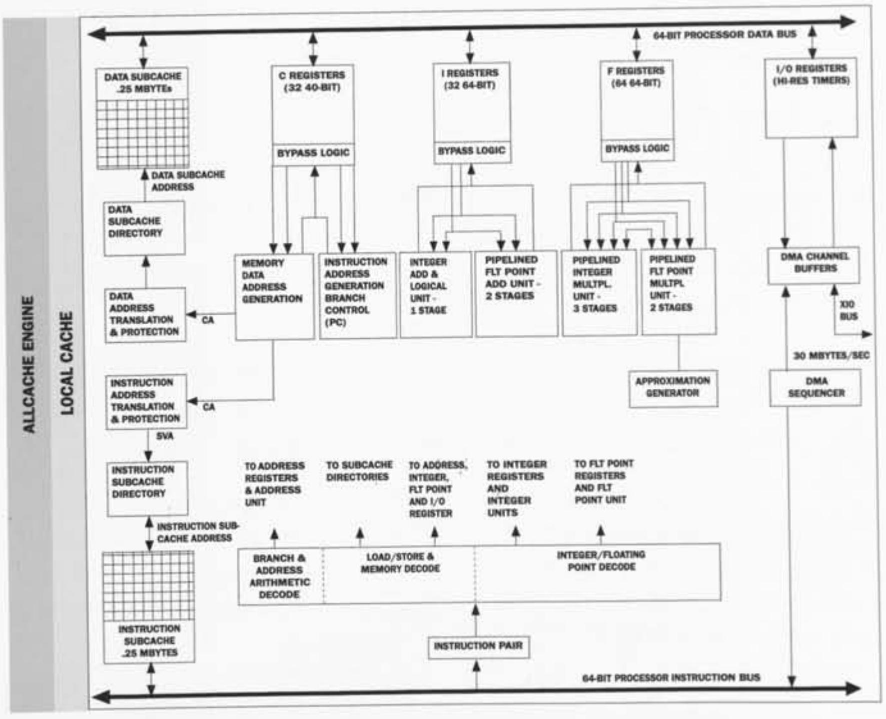
Note the very large number of registers provided by the processor — a total of 128 in the CEU, FPU and IPU plus 64 special I/O control and data registers in the XIU. The large register set makes it possible for the compilers to unroll loops and, in general, to keep operands in registers for as long as they are needed, thereby reducing the requirements for load and store operations.

All functional units of a KSR1 processor are pipelined so that an operation can be started in each functional unit on each clock cycle, even if the result of the previous operation is not yet available. Of course, the result of an operation cannot be used until that operation is complete. Therefore, KSR compilers ensure that the correct number of clocks elapse between the initiation of one instruction and the initiation of another instruction, which depend on the result of the first instruction.

Figure 26 is a functional diagram of the KSR1 processor. The processor is connected to memory by two buses, one for instructions and one for data. Both are 64 bits wide. The 64 bit instruction bus is used to fetch a pair of 32 bit instructions on each clock cycle.

- Instruction-1 of the pair is an address calculation, branch, memory instruction or I/O operation (to be executed by the CEU or XIU).
- Instruction-2 of the pair is a floating point or integer arithmetic operation or a logical operation (to be executed by the FPU or IPU).

FIGURE 26 *KSR1 Instruction Execution Model*


ALLCACHE ENGINE
LOCAL CACHE
64-BIT PROCESSOR DATA BUS
DATA SUBCACHE .25 MBYTEs
C REGISTERS (32 40-BIT)
I REGISTERS (32 64-BIT)
F REGISTERS (64 64-BIT)
I/O REGISTERS (HI-RES TIMERS)
BYPASS LOGIC
BYPASS LOGIC
BYPASS LOGIC
DATA SUBCACHE ADDRESS
DATA SUBCACHE DIRECTORY
MEMORY DATA ADDRESS GENERATION
INSTRUCTION ADDRESS GENERATION BRANCH CONTROL (PC)
INTEGER ADD & LOGICAL UNIT - 1 STAGE
PIPELINED FLT POINT ADD UNIT - 2 STAGES
PIPELINED INTEGER MULTPL. UNIT - 3 STAGES
PIPELINED FLT POINT MULTPL UNIT - 2 STAGES
DMA CHANNEL BUFFERS
DATA ADDRESS TRANSLATION & PROTECTION
CA
XIO BUS
30 MBYTES/SEC
INSTRUCTION ADDRESS TRANSLATION & PROTECTION
CA
APPROXIMATION GENERATOR
DMA SEQUENCER
SVA
INSTRUCTION SUBCACHE DIRECTORY
TO ADDRESS REGISTERS & ADDRESS UNIT
TO SUBCACHE DIRECTORIES
TO ADDRESS, INTEGER, FLT POINT AND I/O REGISTER
TO INTEGER REGISTERS AND INTEGER UNITS
TO FLT POINT REGISTERS AND FLT POINT UNIT
INSTRUCTION SUB-CACHE ADDRESS
BRANCH & ADDRESS ARITHMETIC DECODE
LOAD/STORE & MEMORY DECODE
INTEGER/FLOATING POINT DECODE
INSTRUCTION SUBCACHE .25 MBYTES
INSTRUCTION PAIR
64-BIT PROCESSOR INSTRUCTION BUS

A typical block of assembler code for a KSR1 processor appears in Figure 27. It contains two columns, one consisting primarily of loads and stores, the other mainly arithmetic operations. Memory references are being overlapped with computations.

FIGURE 27 *KSR1 Processor — Typical Block of Assembler Code*

```
  fadd8.tr     %f8, %f10, %f8        ; bcc.qn        @citst, .L5
  fadd8.tr     %f7, %f9, %f7         ; st8           %f8, -16(%c6)
  itsteq8      0, %i10               ; st8           %f7, -8(%c6)
  add8.ntr     2, %i31, %i31         ; bcs.qt        @citst, .L2
.L3:
  itsteq8      4, %i9                ; ld8           32(%c7), %f15
  sub8.ntr     %i9, 4, %i9           ; ld8           40(%c7), %f11
  add8.ntr     31, %i31, %i31        ; ld8           0(%c7), %f6
  fmul8.tr     %f0, %f15, %f14       ; ld8           8(%c7), %f5
  fmas8.tr     %f1, %f11, %f14       ; ld8           16(%c7), %f8
  fmul8.tr     %f0, %f11, %f13       ; ld8           24(%c7), %f7
  fmad8.tr     %f1, %f15, %f13       ; ld8.ex        40(%c6), %f18
  fmul8.tr     %f0, %f6, %f10        ; ld8.ex        24(%c6), %f16
  fmas8.tr     %f1, %f5, %f10        ; ld8.ex        16(%c6), %f17
  fmul8.tr     %f0, %f5, %f9         ; ld8.ex        8(%c6), %f5
  fmad8.tr     %f1, %f6, %f9         ; ld8.ex        0(%c6), %f6
  fadd8.tr     %f13, %f18, %f13      ; ld8.ex        32(%c6), %f19
  finop                              ; ld8.ex        48(%c6), %f21
  fadd8.tr     %f10, %f6, %f10       ; ld8.ex        56(%c6), %f20
  fadd8.tr     %f9, %f5, %f9         ; ld8           48(%c7), %f5
  fadd8.tr     %f14, %f19, %f14      ; ld8           56(%c7), %f6
  fmul8.tr     %f0, %f7, %f11        ; sadd8.ntr     0, %c6, 64, %c6
  fmad8.tr     %f1, %f8, %f11        ; st8           %f9, -56(%c6)
  fmul8.tr     %f0, %f8, %f12        ; sadd8.ntr     0, %c7, 64, %c7
```

KSR1 instructions fall into six classes:

- Memory reference instructions
- Execute instructions
- Control flow instructions
- Memory control instructions
- I/O instructions for the XIU
- Inserted instructions

Memory reference instructions (loads and stores) move data between memory and the registers of the processor. Consider, for example, the following KSR1 load instruction:

```
ld8 60 (%c12), %f16
```

The instruction says to load eight bytes (one full word) into floating point register %f16 (one of 64 floating point registers). The eight bytes to be loaded begin at an address designated by the expression 60 (%c12). The address is computed by taking the value in the 40 bit address register %c12 (one of 32 address registers) and adding the displacement 60.

This address is called a *context address (CA)*. CAs refer to locations within a process address space. CAs are translated into a system-wide address called a *system virtual address (SVA)* by mapping hardware in the CEU. CA to SVA translation provides the means by which the system can control sharing of the address space, supporting sharing simply where desired and preventing inadvertent sharing.

As one can see in the example in Figure 27, loads and stores may appear one after another in the text of a program. Since each load and store moves eight bytes between registers and memory and they occur at the rate of 20 million per second, the memory bandwidth of each processor is 160 MBytes/sec.

Execute instructions (arithmetic, logical and type conversion operations) typically operate on the data in two source registers and place results in a third register. For example:

```
add8.tr %i1, %i22, %i17
```

This instruction adds the eight byte integer in the integer register %i1 to the eight byte integer in %i22 and puts the result in %i17. The processor's three register operations eliminate the need to copy or reload registers which is common in other architectures.

Control flow instructions (branches and jumps) cause the processor to branch out of its sequential flow instruction execution. For example:

```
beq.qt %c1, %c9, .+32
```

This instruction tells the processor to branch out of its sequential flow if the contents of register %c1 equal the contents of %c9. If so, execution jumps to the instruction at the address computed by adding 32 to the current program counter.

The "qt" in the instruction determines its "quashing" behavior. A branch instruction takes two cycles to complete. The two instruction pairs immediately after the branch in the text of the program will have begun execution by the time the branch is completed. The quashing control tells the processor what to do with the results of those instructions. The instruction can specify one of three actions:

quash_never (qn) — always retain the results of these instructions

quash_true (qt) — retain the results if the result of the test is true

quash_never (qn) — retain the results if the result of the test is false

The flexible quashing behavior of the KSR1 provides the compilers with a means of scheduling code so that there is very little cost to branches and jumps in program flow.

Memory control instructions provide hardware primitives for interprocessor synchronization as well as instructions for controlling the behavior of the memory system.

I/O instructions control the actions of the XIU in transmitting and receiving data from I/O devices. (See Part Two for details on I/O capacity.)

Inserted instructions are sequences of operations forced into a program's flow by a co-processor. The XIU uses inserted load and store instructions to accomplish DMA transfers from/to an I/O device to/from memory. The memory system uses inserted instructions to maintain coherence between a processor's sub-cache and the rest of memory. Inserted instructions provide a very general technique for interfacing new co-processors with the rest of the KSR1 architecture.

AVAILABILITY AND RELIABILITY

The KSR1 family of computers is designed to achieve very high levels of availability through a combination of techniques whose net effect is to make service interruptions rare and recovery times short.

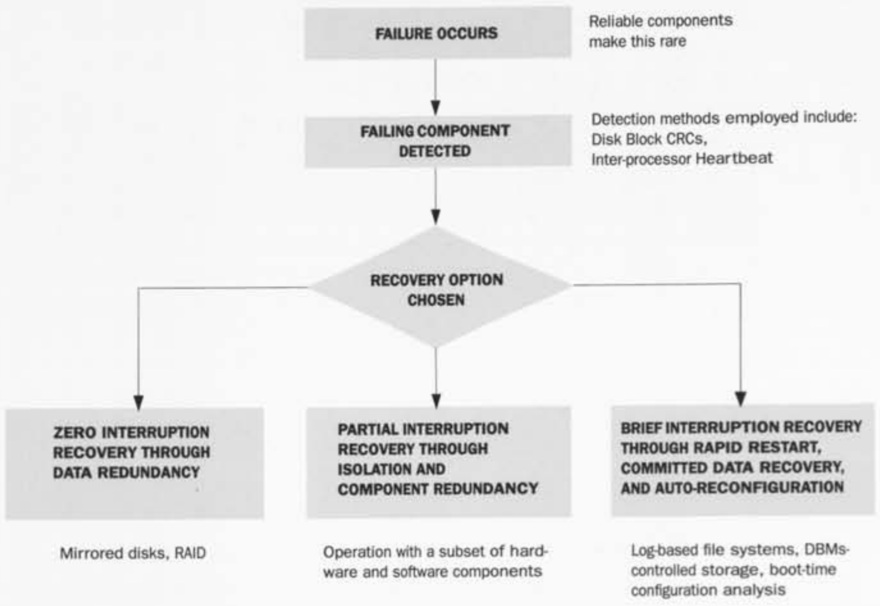
The high availability features of Kendall Square computers revolve around seven themes.

1. Reliable components — The use of custom CMOS lowers component count and allows low-power operation.
2. Ubiquitous fault detection — Component failures are detected through extensive checks for internal consistency. Detected failures are reported to system software to initiate recovery procedures.
3. Data redundancy — KSR1 systems employ redundant storage of data both in RAM (always present) and on disk (configurable) to protect against loss of information if memory components fail.
4. Component redundancy — KSR1 systems are built from large numbers of identical components. The system is designed to be able to operate with only a subset of its components in operation.
5. Fault isolation — KSR1 systems employ a number of techniques to ensure that a failing component, either hardware or software, can be isolated from the healthy remainder of the system. These techniques will often permit a system to "ride through" a failure without requiring a restart.
6. Fast restart — Some component failures require a system restart for recovery. In these circumstances, systems will rapidly restart with no loss of committed data.
7. On line maintenance — Portions of a KSR1 system can be deconfigured under system operator control for maintenance, while the remainder of the system stays on line. Since memory is viewed symmetrically by all processors, KSR OS can migrate processors from the portion of the KSR1 which is being deconfigured to the remainder of the system.

The overall availability strategy is illustrated in Figure 28 on the following page. When a failure occurs, the first job for the system is to detect it, so that remedial action can be taken. Each failure is attributed to some system component. Depending on the nature of the component that failed and the role that it is playing in the system (e.g., if it is a memory device, what data is stored there) one of three recovery options is chosen:

- Whenever possible, the system continues operation with no interruption to any executing process. This option relies upon data redundancy if a memory device has been lost.
- If completely interruption free recovery is not possible, then the system will aim for a recovery plan in which processes which had been employing the failed component must be restarted but other processes continue unaffected.
- If neither of these courses is possible, then the system is restarted very rapidly and database recovery and checkpoint/restart techniques are employed to ensure that no committed data has been lost.

FIGURE 28 *KSR1 High Availability Strategy*


FAILURE OCCURS
Reliable components make this rare
FAILING COMPONENT DETECTED
Detection methods employed include: Disk Block CRCs, Inter-processor Heartbeat
RECOVERY OPTION CHOSEN
ZERO INTERRUPTION RECOVERY THROUGH DATA REDUNDANCY
PARTIAL INTERRUPTION RECOVERY THROUGH ISOLATION AND COMPONENT REDUNDANCY
BRIEF INTERRUPTION RECOVERY THROUGH RAPID RESTART, COMMITTED DATA RECOVERY, AND AUTO-RECONFIGURATION
Mirrored disks, RAID
Operation with a subset of hardware and software components
Log-based file systems, DBMs-controlled storage, boot-time configuration analysis

Reliable Components

Several key component technologies are employed to increase the underlying reliability of the system. The use of custom-designed CMOS serves to reduce component count and achieves lower power dissipation per component. Modern tape automated bonding (TAB) packaging contributes to efficient cooling. TAB packaging allows heat from the die to be conducted directly to the inner plane of the printed circuit board.

All major components of the KSR1 are designed for very high reliability. Among the system's major components only two basic types predominate: the APRD cells (a processor with its local cache) and disk drawers. Each has been engineered with reliability in mind. For example, DRAMs used on the APRD are protected by error correcting code that makes unrecoverable memory failures extremely rare. For mass storage, the KSR1 employs disk drives of very high reliability, and field configurations of the KSR1 allow for two-way mirroring of all disks and other RAID strategies. With this approach, the only occasion on which disk failures become visible to an application in the field is the rare instance when a disk fails while its partner is also unavailable.

Error Detection

KSR1 hardware provides extensive checking for errors. All faults, including those which are corrected transparently by hardware, are reported to system software. Fault detection is supported on a consistent basis, in the following manner:

The Exoskeleton of the KSR1

The KSR1 system was designed both to augment functionality and to convey a visual impression of innovation. It is in keeping with the unique achievement of the patented ALL-CACHE memory system — embedding virtual memory in silicon

Unlike conventionally-appearing supercomputers and mainframes, the KSR1 appears to have no external skin, no vast slabs of painted metal, no opaqueness at all to prevent the eye from penetrating the internal workings of the system. The KSR1 is a series of handsome, vertically turreted panels connected to each other only by a canopy under which a person may easily walk. The modularity of the KSR1 is immediately seen in its external packaging.

Those familiar with the appearance of traditional mainframes and supercomputers in a laboratory or business environment are immediately struck by the lack of gunmetal sidings. Here for the first time is a major computer system that permits people to walk about within its working innards.

To service a KSR1 in the field, one simply "walks" into the system. The traditional rackmount has given way to a new functional modularity, and one whose careful air-cooled design yields the practical benefits of a small footprint and an energy demand that is minuscule in comparison to traditional mainframes or supercomputers. For instance, an IBM ES/9000-620 requires 796 square feet of floor space without peripherals and has a power supply demanding 65,000 watts. A comparable KSR1 with 16 processors uses but 68 square feet with peripherals and requires but 2,000 watts.

- All DRAMs are ECC protected. Single-bit errors are correctable, and software-assisted memory scrubbing is used to minimize the probability of multiple-bit errors. If the single-bit error rate for an APRD exceeds a threshold, the unit should be replaced.
- All other RAM is parity-protected to detect errors. This includes all visible and transparent locations.
- All data transfers over interconnects and buses are at least parity protected. The interconnection between ALLCACHE Engine:0 and ALLCACHE Engine:1 is further protected by a guaranteed transport mechanism.
- The memory-system protocol detects logical inconsistencies. For example, inconsistencies in memory-system state — such as dual-subpage ownership or inappropriate commands — are detected.
- Memory-system faults are confined by a flag within each descriptor, which records that the descriptor or the data which it describes might have been damaged.
- Request time-outs are used by memory-system requestors to detect missing responses.
- The disk subsystem optionally provides redundancy to handle disk drive and most device-controller failures. This is accomplished by a parity and checksum technique which tolerates a single failure across each array of five disk drives. In addition, an error-

correcting code allows small burst errors to be corrected by the device controller. If correctable errors for a disk drive exceed a threshold, the disk drive should be replaced.

- Some system-software faults are detected by individual watchdog timers and "heart beat" monitors on each APRD cell.

Using these mechanisms, the system can rapidly detect the occurrence of errors and isolate them to particular failing components.

Data Redundancy

KSR1 systems can often tolerate the failure of data storage devices and continue operation without interruption to any executing process through the use of data redundancy. Data redundancy provides the system with an alternative source for data that was stored on the now inaccessible device. The alternative source is used as both the means to satisfy the data needs of the executing process and as the means to recreate redundancy when the failed device is replaced (if necessary).

Data redundancy arises in three places in KSR1:

- ECC protection of DRAMs. This mechanism employs redundancy to reconstruct the contents of a single failed bit in a word.
- Mirrored disks. KSR1 users can choose to store the contents of logical volumes on two or three physical disk drives. System software manages the synchronization of these devices and automatically reconstructs the contents of a failed disk after replacement. Mirrored disks also perform better than single disks on read accesses by providing two or three disk arms which can access the same logical device simultaneously.
- RAID3. Another available method for implementing data redundancy on disk storage is configuring the disk drivers to perform RAID3 functionality. This method is designed for large block data transfers. It stripes logical blocks across four disk drives and stores a parity bit computed from these four drives on a fifth drive. This method is more cost effective than mirrored disks, but does not perform as well for short random references.

Component Redundancy

Highly parallel systems offer the potential for high system availability unmatched by earlier architectures. Since parallel systems are built from large numbers of identical components, they can be engineered to operate with some of those components out of service. The KSR1 exploits this potential. Since all processors view memory symmetrically and programs reference memory addresses, not processors, the KSR1 operates effectively with a subset of its processors, a subset of its disk modules, even a subset of its interconnect and power system.

The interconnect system of a KSR1 is a hierarchy of ALLCACHE Engines. Each ALLCACHE Engine is built as a set of independent sub-ALLCACHE Engines, so that the failure of a single component results only in the breaking of a single sub-ALLCACHE Engine communication line. When a failure does occur, a portion of the ALLCACHE Engine's bandwidth is lost temporarily (because of the loss of the sub-ALLCACHE Engine itself). But connectivity is not lost. The failure of a single sub-unit component does not prevent continued communications among the other ALLCACHE Engines, since none are classical descendants of the broken component. ALLCACHE Engine:0s have two sub-ALLCACHE Engines. ALLCACHE Engine:1s can be configured to have two, four, or eight sub-ALLCACHE Engines.

The KSR1 power system is modular in its design and is intended to provide continuous operations. Each Power Module provides 300 volts DC to the various electronic modules (such as the modules of ALLCACHE Engine:1, ALLCACHE Group:0, etc.) which regulate their own power as appropriate to their needs. Each Power Module is equipped with battery back-up which stores sufficient energy to deliver its rated capacity for five minutes when external power is interrupted. This is enough time to survive most power failures and provides the interval needed to switch to a generator or other alternative power source. The power system signals KSR OS toward the end of the battery coverage period, so that the system may be shut down gracefully if need be.

Fault Isolation

The KSR1 system confines the effects of a component failure to as small a population of users as possible. This effect is achieved through a combination of hardware and software techniques.

Hardware faults are detected, reported and recorded in special registers in each local cache. If a memory system fault is detected, this register records the addresses of any pages which may have been damaged. Fault containment is achieved by preventing subsequent access (and any movement of the data within the system) to the damaged pages by signaling that a fault has been previously detected.

Fast Restart

Some failures will require system restart for recovery and KSR1 systems are capable of rapid restarts, to ensure that no committed data is lost.

When a system is restarted, the boot process examines the state of all components for correct operation. This process includes examination of configuration data left behind during the previous run. These records often identify the failing device even when simple tests conducted at restart do not.

At restart, the DBMS processes its re-do log to ensure that all committed transactions are reflected in the database and that uncommitted transactions leave no traces in the database. The DBMS can be configured to make this recovery process arbitrarily short. The KSR OS can be configured with a log based file system which can be rapidly restarted with no loss of information.

PART SIX

A BRIEF HISTORY OF MULTIPROCESSOR RESEARCH

Throughout the Technical Summary, we have presented the patented ALLCACHE memory system as the only known, practicable way to achieve scalability in a highly parallel processing system while retaining the familiar shared-memory programming model. Until the advent of the KSR1, architectures that provided the shared-memory programming model generally exhibited one of two major deficiencies – either they were not scalable or they did not support memory coherency. This is a review of the state of the art prior to the introduction of the KSR1.

VIRTUAL MEMORY

The most fundamental influence on the KSR1 architecture was the development of virtual memory and its ability to present a single address space model to the programmer and to automatically exploit locality.

The property of locality is a program's preference for a subset of its address space over a given period of time. Exploiting locality in the construction of computers (for example, short interconnections on or between chips), as well as exploiting locality in program behavior, has been a primary factor in enhancing computer performance. To understand the relevance of locality and single address space to parallel computing, one must go back in history 30 years when the magnitude and complexity of storage management on uni-processors caused programming difficulties similar to those experienced today on MPPs.

In 1961, the designers of the Atlas Computer at the University of Manchester in the UK, provided an elegant solution to the problems of storage management through the invention of virtual memory.[1] Their invention has profoundly influenced the course of computing.

Simply stated, virtual memory moves the responsibility of managing memory from the application to the computer hardware and systems software, by applying the notion that the "address" is a concept distinct from the physical location of its corresponding data. Programming is simplified, because applications are written with one simple and powerful abstraction – a single address space. Virtual memory provides excellent performance by dynamically exploiting "the property of locality, which is exhibited to varying degrees by all practical programs."[2]

Virtual memory is fundamental to the architecture and programming of all modern mainframes, mini-computers and workstations. Cache memory, a more recent invention, is based on the ideas of virtual memory and locality, and cache is now present on all computers from mainframes to PCs (both RISC and CISC processors). The concept of single level store[3], or mapping files directly into the single address space, is also a direct descendant of the concepts of virtual memory. Modern computer systems depend on locality to extract maximum performance, from single mainframes to networks of workstations paging across a LAN, to fileservers.

1. Kilburn, T., Edwards, D.B.G., Lanigan, M.J., and Sumner, F.H. "One-level Storage System;" IRE Transactions, EC-11, Vol.2, pps. 223-235, April, 1962.
2. Denning, Peter J. "On Modeling Program Behavior;" Arlington, VA: AFIPS Press: Proceedings, Spring Joint Computer Conference, Vol. 40, pps. 937-944, 1972.
3. Organic, E.I., "The Multics System: An Examination of Its Structure," Cambridge, MA: MIT Press, 1972.

KSR1 ALLCACHE extends the concept of virtual memory to highly parallel processing for the first time, thus providing all the benefits of virtual memory, including high performance, ease of programming and scalability.

NON-SCALABLE SHARED MEMORY ARCHITECTURES

The first research multiprocessor was C.mmp.[1] It consisted of 16 processors with an optional cache, connected through a crossbar to 16 shared memory modules. The cache was designed and prototyped, but never used because of cache coherence problems.

The problem of cache coherence on a multiprocessor was first solved in 1981 by Synapse.[2] The Synapse architecture consisted of as many as 28 processors and four memory modules on a shared bus. The basic innovation of Synapse was to introduce the concept of ownership, distributed directories and bus monitoring (later dubbed "snooping") as a way to solve the cache coherence problem. Coherence algorithms, based on the concept of ownership, reduced bus traffic significantly. But, as a rule, bus-based multiprocessors proved to be scalable only to a maximum of 20 to 30 processors on a single bus. Encore Computer Corp. and Sequent Computer have developed similar bus-based multiprocessors.

SCALABILITY

Architectures based on distributed processors and their associated memory units have been the most common solutions to the scalability problem. In this distributed processor/memory approach, the aggregate memory bandwidth theoretically increases in direct proportion to the number of processors.[3]

Distributed parallel computers can generally be classified into two distinct categories:

- Multicomputers – with a separate address space for each processor/memory pair.
- Multiprocessors – with a single address space.[4]

Because the preferred programming model is shared memory, the major issue soon arises whether to provide the primitives necessary to support shared memory in the hardware, the software, or in some combination of the two. How these primitives are implemented has significant influence on a number of system design issues, including:

- support of sequential consistency
- granularity of sharing
- efficiency of the coherency implementation (bandwidth, cache refill time, processor overhead, cost)
- scalability

A genuinely scalable design must, however, take into consideration both component and generational scalability. Component scalability can best be illustrated by the simple addition of pro-

1. Wulf, William A. and Bell, C. Gordon. "C.mmp-A multi-miniprocessor;" Proceedings, AFIPS 1972 Fall Joint Computer Conference, 41, pp. 765-777, 1972.
2. Frank, Steven J. "Tightly Coupled Multiprocessor System Speeds Memory Access Times;" Electronics, pps. 164-169, 1984.
3. Seitz, Charles L. "Concurrent Architectures;" Morgan Kaufmann: Frontiers VLSI and Parallel Computation, Suaya R. and Birtwhistle, G., eds., pps. 21-23, 1990.
4. Bell, C. Gordon. "Multis: A New Class of Multiprocessor Computers;" Science, Vol. 228, pps. 462-467, 26 April 1985.

cessing, memory, interconnect and I/O resources to a system; the system's capacity for work scales in an upward fashion with the increase in the number of components employed.

Generational scalability, by contrast, is measured by how these resources scale as the underlying technology itself (e.g., transistor devices packaging) improves over time. In this context, technological scaling means the increase in the capacity of the architecture for work as implementations move from one underlying technology generation to the next. For example, the frequency of buses does not significantly improve with the evolution of integrated-circuit technology. The maximum frequency of bus operations is electrically limited by the speed of light and the number of devices connected to the bus. By contrast, an implementation with short point-to-point connections, such as that employed in the KSR1, provides frequency of operations limited only by the underlying integrated-circuit technology, and it will improve directly as the technology itself improves.

One widely accepted working definition of scalability suggests two distinct considerations with respect to the scaling of the interconnect architecture:[1]

- Scaling of delivered bandwidth
- A bandwidth requirement that itself increases more slowly than the delivered bandwidth of the system

In this regard, much interest has recently developed in a family of routing networks called fat-trees, whose properties include universality and the potential to exploit locality of reference.[2] From a theoretical standpoint, fat-trees can be shown to be a nearly universal network scheme; a fat-tree routing network of any given size is always nearly the best possible network of that particular size.

The image conveyed by the notion of the fat-tree is metaphorically accurate. Like real trees, fat-trees get thicker the farther one gets from the leaves. In the analogy, the processors are the leaves and the tree's internal nodes are its switches. Communication bandwidth increases as one goes down the tree, from the leaves, to the branches, to the trunk, toward the roots.

For a given amount of communications hardware, a fat-tree topology can simulate every other possible network built with an equivalent amount of hardware, using only slightly more time (a polylogarithmic factor greater). More importantly, the bandwidth of a fat-tree can be varied independent of the number of processors. The tree topology of the network also transparently implements bandwidth load balancing.

Other networks described in the literature – such as hypercube, multistage interconnection networks (MINS), or meshes – do not demonstrate the combination of universality, locality, bandwidth variability or load balancing characteristics of the fat-tree.

The KSR1 hierarchy of ALLCACHE Engines described in these pages employs the fat-tree topology, with a span at each level of 32. The basic architecture can be extended to an arbitrary number of levels.

1. Scott, Steven L. "A Cache Coherence Mechanism for Scalable, Shared-Memory Multiprocessors;" Proceedings of International Symposium on Shared Memory Multiprocessing, pps. 49-59, April, 1991.
2. Leiserson, Charles E. "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing;" IEEE Transactions on Computers, Vol. C-34, No. 10, pps. 892-901, October, 1985.

LOCALITY OF REFERENCE

Locality of reference has been shown in recent literature[1] to be the key to achieving scalable interconnect bandwidth whose feature is that the actual bandwidth requirement increases less slowly than the bandwidth delivered. Three design guidelines emerge from this consideration:

- Because communication is limited in a fundamental way by the speed of light, the architecture should be designed in such a way that most communication is kept as close as possible to the referencing processor.
- Communication time is influenced by the number of switches through which messages and data must pass.
- Communications should be maintained within as small a subsystem as possible to reduce congestion in the overall system.

All three points highlight the significance of locality of reference, which suggests that the characteristic of locality – measured and defined in terms of "working sets"[2] – is a key to bandwidth requirements that do not increase as rapidly as the bandwidth available.

A working set may be defined as a collection of addresses which have been referenced recently (and, therefore, from a statistical standpoint, are apt to be referenced again within a short period of time). When combined with the patented ALLCACHE memory system, which has the fundamental ability to exploit locality of reference, the fat-tree topology of the KSR1 provides a system that meets the criterion of bandwidth requirements that increase more slowly than bandwidth delivered.

The hierarchy of ALLCACHE Groups, formed from local caches interconnected by a hierarchy of ALLCACHE Engines, establish a corresponding hierarchy of working sets whose magnitude increases collectively at each successively higher level of system organization. Because most references may be satisfied from within the level where the reference request was itself issued, the effective demand on communications bandwidth to the next higher level is, in actual practice, reduced.

Although a fat-tree topology can deliver scalable bandwidth, the KSR1 system does not require that the bisection bandwidth scale in linear fashion, because the combination of a fat-tree network of ALLCACHE Engines and the caching behavior of the memory system itself serve to reduce the bandwidth requirements in the first place.

The ordering enforced by the fat-tree topological structure also simplifies the maintenance of coherency and allows the implementation of sequential consistency with little performance loss. Sequential consistency conveys the most intuitive impression to the programmer, and sequential consistency itself is derived from the fact that the system behaves as if it were executing all operations on a single processor.[3] To the programmer, the appearance is that of the familiar single-processor/multi-tasking model.

1. Leiserson, Charles E. "VLSI Theory and Parallel Supercomputing;" Pasadena, CA: Proceedings of the 1989 Decennial Caltech Conference, March, 1989.
2. Denning, Peter J. "Working Sets Past and Present;" IEEE Transactions on Software Engineering, SE-6, pps. 64-84, January, 1980
3. Lamport, Leslie, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, Vol.C-28, No. 9, Sept. 1979

DISTRIBUTED ARCHITECTURES WITH SOFTWARE IMPLEMENTATIONS OF SHARED MEMORY

The major phyla of highly parallel computers today may be differentiated by their basic computational models: shared memory versus message passing systems. Multicomputers are typically programmed using a message passing model, rather than shared memory. Several multicomputer architectures have been proposed and built on the basis of the message passing model, including the Cosmic Cube[1], IPSC and the J Machine.[2] In addition, a number of research projects have designed and prototyped a shared virtual memory software layer on a multicomputer, including Ivy[3] and Mether[4]. Although the programming model was improved, these various efforts brought a number of significant problems to the surface. Four particular difficulties with these approaches have emerged:

- Software-based implementations of shared memory are two to three orders of magnitude lower in performance than hardware implementations.
- Searching and directory functions are much slower when managed at the software level.
- Sequential consistency is extremely difficult to achieve in software alone, and sequential consistency is a key to the user-friendly impressions conveyed by the system – it behaves as if it were a single-processor/multitasking machine.
- The grain size in all the software-based implementations of shared memory has been the complete page. The granularity should be smaller to avoid false sharing and provide fast cache-refill times for data movement.

SHARED MEMORY DISTRIBUTED ARCHITECTURES WITHOUT COHERENCY

Two of the key concepts of scalability are the distributed and hierarchical organization of the multiprocessor. The Cm* was the first computer of this type.[5] The basic building block of Cm* was a processor-memory pair called a computer module (Cm). The local memory associated with each processor formed the shared memory for the system. The Cedar project was similar in concept to Cm*. NYU Ultracomputer, RP3 and the BBN Butterfly were other non-hierarchical, distributed memory multiprocessors. None of these architectures fully exploited locality of reference.

In all these systems, because addresses had fixed physical locations, the programmer was compelled to copy data to local addresses to optimize performance. Coherency also had to be managed explicitly within the program.

All these systems adopted shared memory as a syntactic convention, mapping a portion of each processing cell's local memory into a global address space. However, non-local accesses invariably had a longer latency than local references, and these systems had no way to adjust automatically to the addressing pattern of a program. Such adjustments were left, instead, to the application programmer.

---

1. Seitz, Charles L. "The Cosmic Cube;" Communications of the ACM, 28-1, pps. 22-33, January, 1985.
2. Dally, William L. "The J-Machine: A Fine-Grain Concurrent Computer;" MIT VLSI Memo 89-532, May, 1989.
3. Li, Kai and Hudak, Paul. "Memory Coherence in Shared Virtual Memory Systems;" Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing, pps. 229-239, August, 1986.
4. Minnich, Ronald G. and Farber, David J. "The Mether System: Distributed Shared Memory for SunOS 4.0;" (private communication).
5. Swan, R., Fuller, S., and Siewiorek, D. "Cm*- A modular, multi-microprocessor;" Proceedings AFIPS 1977 Fall Joint Computer Conference, 46, pps. 637-644, 1977.

In essence, these machines were similar in programing style and performance to message passers: blocks of data had to be copied from global space to a processor's local space, manipulated there and then written back. Management of the contents of the local memory and the maintenance of coherence between data in local memory and data in global space were relegated back to the application programmer as well.[1] In this sense, this entire class of machines achieved the form of shared memory but never its substance.

SHARED MEMORY AS A NETWORK ABSTRACTION

Memnet and Capnet are distributed, virtual, shared memory architectures aimed at using shared memory as a network abstraction for high performance communications.[2]

The Memnet programming model involves coherent shared memory.[3] Memnet was implemented in the form of multiple processor nodes communicating over an insertion-modification token ring. Each node is connected to the ring through a dual-ported memory, which is managed as a cache by hardware. Each computing node references this cache as part of the real memory space from the local bus. The interfaces communicate over the ring to maintain coherency, using a broadcast-based ownership coherency protocol. Data is transferred and coherency maintained on "chunks" of 32 bytes.

The ring provided several advantages over bus-based systems:

1. Buses suffer from decreased bandwidth as the length of the bus is increased to raise the number of computing nodes. By comparison, ring bandwidth does not decrease as additional nodes are added.

2. Rings provide an absolute upper boundary on the time of network access.

3. The topology of the ring allows ordering to be maintained and a pseudo broadcast capability, which simplifies the management of coherency.

The KSR1 uses a ring to implement the ALLCACHE Engine hierarchy, for the same reasons as Memnet, but with the addition of the concept of distributed directories. While ring latency does increase as processors are added to the configuration and may be quantified as 0(N), this is not significant for small N. Because the KSR1 is based on a fat-tree hierarchy of ALLCACHE Engines, if the ring size of ALLCACHE Engine:n is r, the overall latency grows as $0(\log_r N)$.

Another advantage of a ring over a bus is the enhancement of "generational scaling." As described before, this is the increase in an architecture's capacity for work as the underlying technology evolves and improves from one generation to the next. Bus frequency will not be significantly augmented by improvements in integrated circuit technology, because the maximum frequency of bus operations is fundamentally limited by the laws of physics. By contrast, however, for a ring implementation with short point-to-point connections, such as that in the architecture of the KSR1,[4] the only limit on the frequency of operations resides in the underlying technology and

1. Picano, S., Brooks, E., and Hoag, J. "Programming Costs of Explicit Memory Localization on a Large Scale Shared Memory Multiprocessor;" Albuquerque, NM: Proceedings of Supercomputing '91, pps. 36-45 November 1991.
2. Delp, Gary, Farber, David, Minnich, Ronald et al. "Memory as a Network Abstraction;" IEEE Network, July, 1991.
3. Delp, Gary. "The Architecture and Implementation of Memnet: A High-Speed Shared Memory Computer Communication Network;" Ph.D. thesis, University of Delaware, 1988.
4. The ALLCACHE Engine implementation uses short point-to-point connections. When longer connections are used, such as in Memnet, multiple bits can be in flight between nodes in a pipelined manner. As increased numbers of transistors become available with the technological scaling of CMOS, the same technique may be used on shorter connections.

will improve as the technology itself evolves and improves. In addition, the ring's point-to-point connections map well into CMOS and optical technologies.[1]

Because it bypassed the layered communication protocol and message passing, Memnet demonstrated a factor of up to 1,000 speedup for distributed system interaction. But the major inhibition of Memnet was its flat ring topology, which limited scalability. From an implementation point of view, the cache directories did not scale [growing at the rate of $0(N^2)$] because the entire shared address space was mapped onto each node.

CapNet[2] is a proposal by Tam and Farber to improve the scalability of Memnet and to explore the possibilities of implementing the shared memory paradigm across a wide area. Three key issues are listed as critical to the attainment of these goals:

- Memory hierarchy,
- Translation of virtual shared address to physical address,
- Maintenance of cache coherency.

All of these issues are directly addressed by the KSR1 ALLCACHE memory architecture. In point of fact, the proposed CapNet solutions are similar to those already implemented in the KSR1. CapNet consists of a number of processors (or possibly clusters of processors) interconnected by a switching network. Memory requests are routed according to the system's virtual shared address. Each switch keeps a page table that indicates the outgoing path in the network to the owner of a page (chunk). Thus page location information is distributed across the network. The current page location is always maintained within the network because the page tables in the switches are modified as pages move in the network. The proponents of CapNet point out that this type of architecture results in the minimum number of messages and no broadcast. This minimizes latency for wide area implementations, where latency is governed by physical distances.

CapNet has identified the following major research and implementation issues:

- Providing proof of correctness in light of high system concurrency and dynamic movement of addresses,
- Avoiding degeneration (how to guarantee forward progress),
- Handling page table overflows,
- Maintaining memory coherency.

Solutions to each of these issues are embodied in the implementation of the KSR1.

The basic notion of CapNet, that memory requests are to be routed based on virtual shared address, is essentially a subset of the ALLCACHE memory architecture. The heart of ALLCACHE memory is the concept from virtual memory that "address" (we call it "System Virtual Address" or SVA) is distinct from "physical location." In the KSR1, there is no fixed home (physical location) for an "address" within ALLCACHE. A "System Virtual Address" (SVA) migrates to the point of reference on demand.

The ALLCACHE Engine consists of ARD (ALLCACHE Router and Directory) cards that route SVA requests dynamically to the point of reference. The ALLCACHE Engine is guaranteed to have sufficient capacity to hold all of the requests that the number of configured processors can have in progress at any one time. This remains true whether the pending requests are for the same

1. CMOS technology is fastest for short point-to-point connections.
2. Tam, Ming and Farber, David. "CAPNET – An Approach to Ultra High Speed Network;" Proceedings IEEE International Conference on Communications '90, 1990.

or different addresses, and even if the addresses are currently moving in the network. The coherency protocol for this level of operation has been formally verified.

The CapNet proposal suggests that a tree structure network guarantees forward progress. Experience with the KSR1 has shown that the ordering enforced by the tree-structured network also significantly simplifies the problem of maintaining coherence, especially with regard to multiple simultaneous requests in the network for a single address.

The CapNet notion that the current location of a virtual shared address is maintained by switches within the interconnect network is a subset of the ALLCACHE Engine concept. The ALLCACHE Engine implementation has solved many of the research issues identified in the CapNet proposal.

The pages described by an ALLCACHE Engine are bounded, because the directory at that level of the hierarchy includes all pages described by its descendants. This is in sharp contrast to CapNet, which is required to store logical pointers to other owners, leading to the possibility of overflow. In the KSR1, the ALLCACHE Engine directory is not required to store logical pointers to other owners of all other pages. An overflow event is impossible in the ALLCACHE architecture.

OTHER CACHE-ORIENTED, SHARED MEMORY, SCALABLE COMPUTERS

The Alewife[1] and Dash[2] research projects share a common goal and with the KSR1: development of a scalable, shared memory multiprocessor. Both projects depend primarily on caching to achieve scalability. Despite these apparent similarities, there are many differences among the efforts.

Where ALLCACHE memory system uses a fat-tree topology to interconnect nodes, both Alewife and Dash use a 2D mesh network. (The advantages of the fat-tree approach have been described earlier.)

In the case of Alewife, each node corresponds to a single processor, while, in Dash, each corresponds to a cluster of four processors.

The KSR1 and Alewife both support a sequentially consistent memory model, in contrast to Dash, which supports only a form of weak consistency called "release consistency." The approach taken by Dash introduces a new task for the programmer: specifying which memory accesses require sequential consistency.

Alewife uses a directory based cache coherence protocol, which is implemented as a combination of hardware and software techniques. Dash also uses a directory based scheme, but one that is entirely implemented in hardware. By comparison to the KSR1 ALLCACHE memory system, there are three considerations that may be enumerated as inhibiting the scalability of Alewife (issues 1-3 below), and two issues (2 and 3 below) which would seem to limit the scalability of Dash:

1. The Alewife hardware directly supports only a small number of copies of an address. For any greater number of copies, system software must intervene. Thus, for the common case of an application with a high degree of read-only sharing, Alewife performance falls off significantly. By contrast, the KSR1 ALLCACHE approach handles an unlimited number of copies in hardware.

1. Chaiken, David, Kubiatowicz, John and Agarwal, Anant. "LimitLESS Directories: A Scalable Cache Coherence Scheme;" Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, pps. 224-234, April 1991.

2. Lenoski, Daniel, Laudon, James, Gharachorloo, Kourosh, Wolf-Dietrich Weber, Gupta, Anoop, and Hennessy, John. "Overview and Status of the Stanford DASH Multiprocessor;" Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing, pps. 229-239, April, 1991.

2. Alewife and Dash are "somecache" architectures that have a fixed home for addresses. In addition to the caches (whose constituent addresses change dynamically), both Alewife and Dash employ ordinary memory modules (whose constituent addresses are fixed). These modules provide a "home" storage location for all addresses. The location of an address's home is determined statistically from the address, not dynamically according to program behavior. Local cache misses are resolved by referencing the home memory module. By way of contrast, in the KSR1 ALLCACHE memory system, there is no fixed home for an address, all instances of an address move on demand and programmers need not be cognizant of the "home" locations to optimize locality. The concept of a "home" location leads to the problem of "de-scaling" discussed in detail below.

3. Neither Alewife nor Dash guarantees forward progress, because a processor is not guaranteed fair access to an address with high contention. By comparison, the KSR1 fat-tree hierarchy and ring topology within its ALLCACHE Engines guarantee fair access to addresses with high contention.

THE DE-SCALING TEST

Determining the potential scalability of a plausible computer architecture is often fraught with difficulty unless extensive experimentation can be done. However, it is often easier to test for de-scaling behavior given only an abstract description of an architecture. A proposed architecture can be said to exhibit de-scaling behavior if a fixed size parallel program performs worse as the machine gets bigger. A truly scalable architecture must not manifest de-scaling behavior.

The de-scaling test may be conducted as follows:

- Consider a parallel program executing on a computer of fixed size. Set the degree of parallelization to a level appropriate to that size of machine and to the application.
- Now expand the computer, adding processing elements, interconnect capacity and other components. Run the same program as before at the same degree of parallelization.
- What has happened to execution times? These will not improve with the second test because the degree of parallelization has not been increased. However, performance may degrade because of the increased scale of the computer. This is de-scaling behavior, and any architecture that manifests it while the system is properly used has inherent limits to scalability.

All highly parallel systems will exhibit some degree of de-scaling behavior unless programs exhibit parallel locality. (But, as we shall see, some architectures manifest de-scaling propensities even when programs exhibit parallel locality.) In all highly parallel systems, the distance between an arbitrarily chosen pair of processing elements grows monotonically with the count of processing elements. Thus, if communicating threads of a program are assigned to arbitrary processing elements, communication time will rise with the system scale, and de-scaling will result. To avoid de-scaling, programs must exhibit parallel locality: rather than scheduling communicating threads on arbitrary processors, the work must be scheduled on "nearby processors." The distance between "nearby processors" does not grow with system size.

ALLCACHE memory is designed to exploit parallel locality. If related threads of a parallel program are scheduled for execution within the same branch of the ALLCACHE hierarchy, then local cache misses will be resolved with that branch, regardless of how many other branches there may be to the system. Thus ALLCACHE does not exhibit de-scaling behavior.

By comparison, de-scaling is manifest in "somecache" architectures such as Dash and Alewife. Although their basic objectives are the same, there remains one crucial difference – both Alewife and Dash use conventional memory to provide a "home" for addresses. Local cache misses are resolved by referencing the home memory module, and the location of home is not adjusted dynamically with the behavior of programs. The average distance from a processor experiencing a local cache miss and the home memory module where that reference will be resolved is equal to the average distance between an arbitrary pair of processing elements in the system – and this condition will grow as the size of the system does. As a result, "somecache" systems exhibit de-scaling behavior, even when applications demonstrate parallel locality. Shared memory distributed architectures, such as that embodied in Cm*, will also manifest de-scaling behavior.

ALLCACHE is the only highly parallel, shared memory architecture that avoids de-scaling.

"KENDALL SQUARE'S SUCCESS IS TIED TO THE CHALLENGES YOU FACE; TO SUPPLY THE WORLD WITH BETTER PRODUCTS AND SERVICES. THESE CHALLENGES MAY BE PART OF THE SCIENTIFIC GRAND CHALLENGES SET FORTH BY THE FEDERAL HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS PROGRAM OR, THEY MAY BE THE COMMERCIAL GRAND CHALLENGES FACING AEROSPACE, CHEMICAL, BANKING, TELECOMMUNICATION, PHARMACEUTICAL, AIRLINE, INSURANCE, PETROLEUM AND OTHER INDUSTRIES. IN BOTH AREAS KENDALL SQUARE IS DETERMINED TO HELP BY OFFERING THE FUNCTIONALITY AND PERFORMANCE YOU DEMAND AT A UNIT COST THAT MAKES OUR SYSTEMS THE INEVITABLE CHOICE. OUR VISION FOR THE FUTURE IS SIMPLE: A NEW KIND OF PARTNERSHIP WITH CUSTOMERS, DRIVEN BY THEIR REQUIREMENTS AND OUR ABILITY TO SUPPLY A CONTINUING STREAM OF TECHNICAL INNOVATION TO MEET THEIR NEEDS."

HENRY BURKHARDT III

170 Tracer Lane Waltham, Massachusetts 02154-1379 Tel 617 895 9400 Fax 617 890 0996 email: ksr-info@ksr.com