The KSR1: Bridging the Gap Between Shared Memory and MPPs

Steven Frank, Henry Burkhardt III and Dr. James Rothnie
steve@ksr.com

Kendall Square Research Corporation
170 Tracer Lane, Waltham, MA 02154

Abstract

Historically, shared memory and virtual memory have been the main line programming model from an applications and computer science perspective for two reasons: shared memory is a flexible and high performance means of communicating between processors, tasks or threads and; shared memory provides high programming efficiency through use of conventional memory management methods.

The KSR1™ bridges the gap between the historical shared memory model and MPPs by delivering the shared memory programming model and all of its benefits, in a scalable, highly parallel architecture. The KSR1 runs a broad range of mainstream applications, ranging from numerically intensive computation, to on-line transaction processing (OLTP) and database management and inquiry. The use of shared memory enables a standards based open environment. The KSR1's shared memory programming model is made possible by a new architectural technique called ALLCACHE™ memory.

Virtual Memory: The Precursor of the KSR1 Architecture

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 uniprocessors caused programming difficulties similar to those experienced today on MPPs.

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.

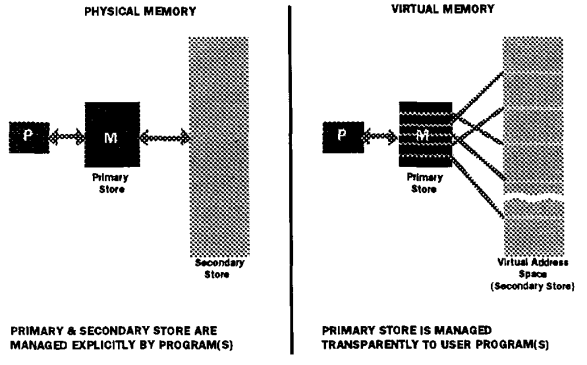
These factors led the designers of the Atlas Computer at the University of Manchester in the UK, to an elegant solution to the problems of storage management through the invention of virtual memory. Their invention has profoundly influenced the course of computing. [1]

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 (see Figure 1). 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 [3], 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 [4], 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.

1063-6390/93 $3.00 © 1993 IEEE

Figure 1 Before the advent of virtual memory primary (main memory) and secondary storage were managed explicitly by the program. With virtual memory a combination of hardware and system software manage the primary store transparently to user programs.



Virtual Memory and MPPs

These 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. [5]

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.

Prior research - multiprocessors

The first research multiprocessor was C.mmp. [6] 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 implemented in 1981 by Synapse for greater than four processors. [7] 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 Systems have developed similar bus-based multiprocessors.

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 [8], IPSC and the J Machine. [9] In addition, a number of research projects have designed and prototyped a shared virtual memory software layer on a multicomputer, including Ivy [10] and Mether [11]. 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 faster porting of programs as this is the model assumed by most programmers.
- 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.

"No-Cache" architectures

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. [12] 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.

In essence, these machines were similar in programming 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 [13] as well. Effectively this is the same style of data movement as used in a message passing MPP with all the same drawbacks.

Other "Some-Cache" architectures

The Alewife [14] and Dash [15] research projects share a common goal with the KSR1: development of a scalable, shared memory multiprocessor. Both research projects depend primarily on caching to achieve scalability, but have scalability limitations similar to "some-cache" and "no-cache" architectures.

Alewife and Dash are "some-cache" 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 statically from the address, not dynamically according to program behavior. Local cache misses are resolved by referencing the home memory module. Since the size of the caches is small, compared to the size of ordinary memory, these research projects do not exploit locality of reference and behavior similarly to "No-Cache" architectures.

The KSR1 memory system

ALLCACHE memory system [16] 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: [17]

> "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.

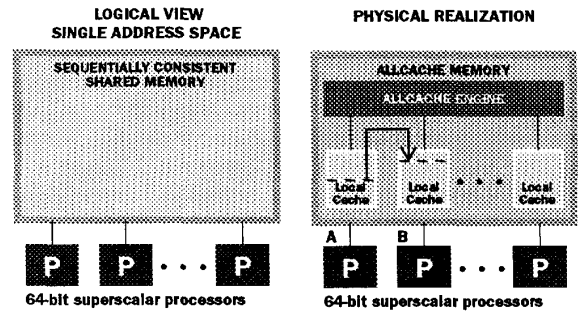
ALLCACHE physically comprises a set of memory arrays called local caches, each capable of storing 32 MBytes. There is one local cache for each processor in the system. Hardware mechanisms (the ALLCACHE Engine described below) cause SVA addresses and their contents to materialize in the local cache of 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 "SOMECACHE" 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 2.

---

[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.

Figure 2 ALLCACHE Memory System: 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.[2] 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:[3]

- 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.

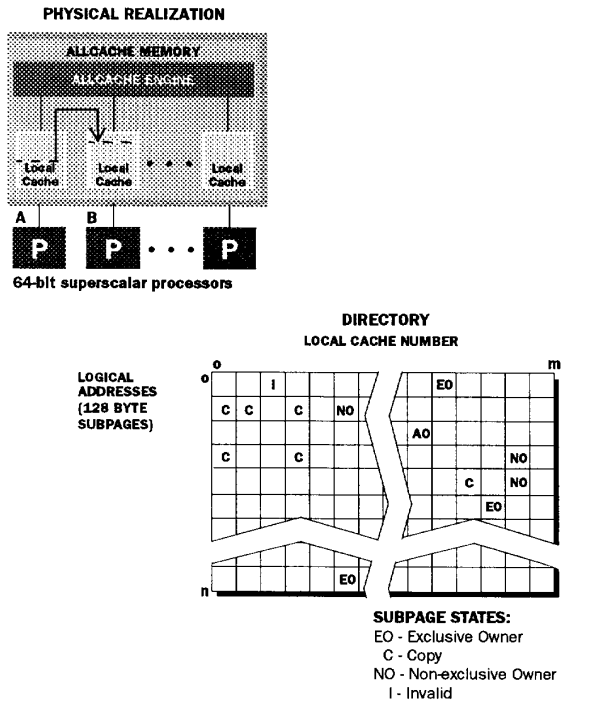
The ALLCACHE Engine manages a directory that determines which one or more local caches contain an instance of each subpage. This directory is physically stored in a distributed and compressed form but its logical function is illustrated in Figure 3. The directory is logically a matrix consisting of a 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.

---

2. 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.

3. Additional states, including, Atomic states are actually used in the implementation of ALLCACHE.

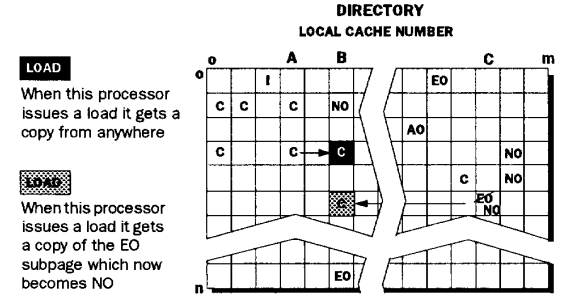
Figure 3 **ALLCACHE Physically distributed directory**



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 4. 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).

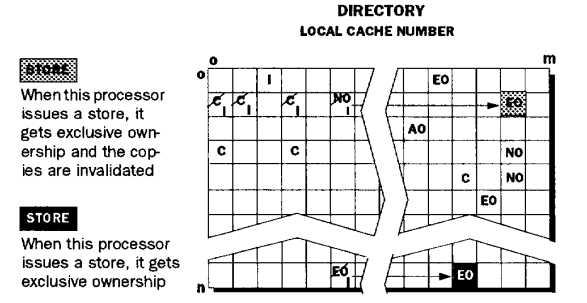
Note that the program that issues the load or store has no knowledge of the respective physical locations of the local caches. The ALLCACHE Engine transparently manages the routing, based on the subpage address and the directory. Memory allocation is handled by the respective local caches.

Figure 4 **ALLCACHE Engine directory operation, Local Cache load miss**



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 (see Figure 5). 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 5 **ALLCACHE Engine directory operation, Local Cache store miss**



Scalability: Hierarchical Organization of the ALLCACHE Engine

The KSR1 architecture exploits locality of reference by organizing a number of ALLCACHE Engines in a hierarchy. 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.

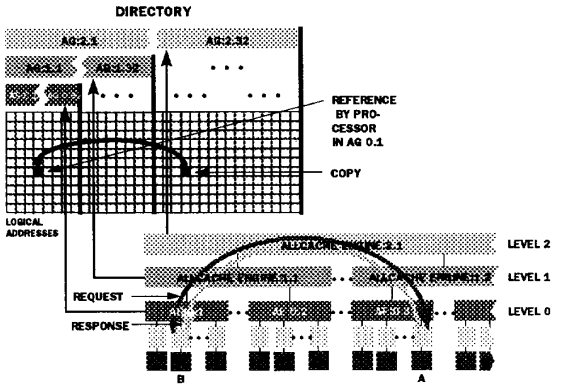
The initial KSR1 system implements two levels of ALLCACHE Engine hierarchy. The ALLCACHE Engine is constructed with a fat-tree [18] topology, so that the bandwidth increases at each level of ALLCACHE Engine. For the KSR1, ALLCACHE Engine:0 has a bandwidth of 1 GB/sec and ALLCACHE Engine:1 has a bandwidth 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.

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

Figure 6 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.

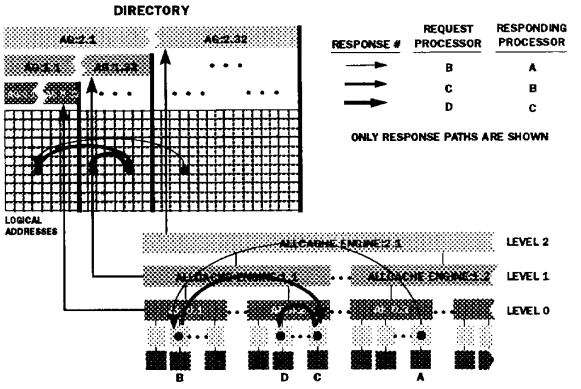
Figure 6 Hierarchical Organization of the ALLCACHE Engine – Search Path Through the Hierarchy



The maximum length of the request path is proportional to the log of the number of processors.

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 7, 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 7 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: [19]

- 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.

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 high hit rates.

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.

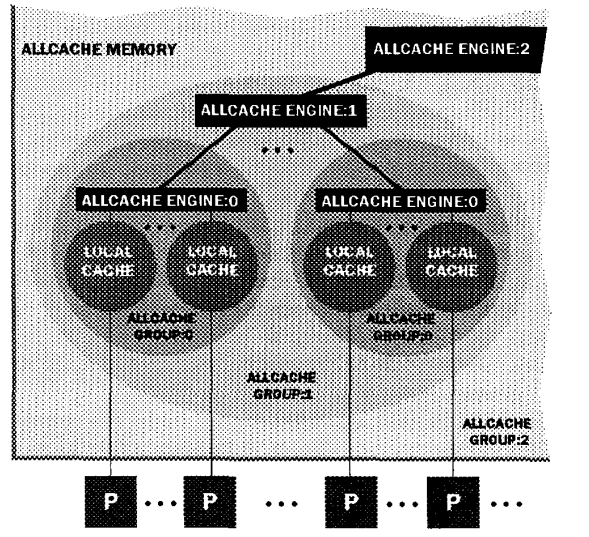
Another way to look at this phenomenon is as a hierarchy of "working sets." For each processor, the addresses most likely to require reference lie in the closest and smallest working set, which is realized in the local cache of that processor. The next most likely addresses to be referenced lie in the 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 8 provides an illustration of the hierarchical organization of the ALLCACHE Engine.

Figure 8 Hierarchical Organization of ALLCACHE Engine exploits hierarchical locality of reference by implementing a hierarchy of working sets (caches).



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.

KSR1 System

ALLCACHE forms the foundation which enables the KSR1 to provide the high performance and easy of programming of a conventional shared memory combined with scalability and low cost of parallel processing. The KSR1 provides a balanced scalability across computation, memory and I/O as illustrated in Table 1.

Table 1 KSR1 Balanced Scalability

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

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:0. The ALLCACHE Engine:1 consists of ALLCACHE Router Directory cells (ARD).

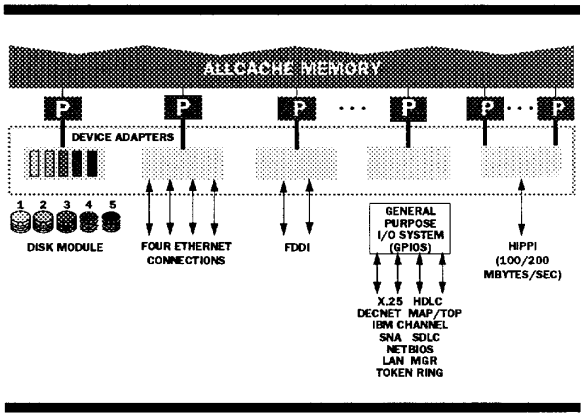
The KSR1 processor employs 64-bit address, 64-bit integer, and 64-bit floating point data types, and it can perform IEEE standard 64-bit floating point operations at a peak rate of 40 mflops. The processor executes two instructions per cycle: one integer or floating point execute and one memory reference or branch.

Each processor supports 30 Mbytes/sec I/O transfers to external sources and users data. A KSR1-32 configured with 32 APRD cells thus achieves an aggregate I/O rate of 450 Mbytes/sec. A KSR1-1088 with up to 510 I/O channels has an aggregate I/O capacity of 15,300 Mbytes/sec. Figure 9 shows connections between KSR1 systems and external devices. The KSR1 supports several types of direct I/O adaptors including Multiple Channel Disk, Multiple Channel Ethernet, Multiple Channel FDDI and Single Channel HiPPI. The VME Channel Controller provides an open interface for other networking and customer specific requirements.

The Multiple Channel Disk adaptor provides five differential SCSI channels to support disk array (RAID) mass storage subsystems. The KSR OS implements disk striping across multiple adaptors.

A companion paper [20] discusses KSR OS system software.

Figure 9 Scalable I/O Bandwidth Combined with Standards



Shared Memory Model Plus ALLCACHE Mean Higher Performance

By building on the well understood shared memory and virtual memory programming models, ALLCACHE enables the KSR1 to provide a conventional shared memory programming model. A standards based programming environment facilitates porting existing applications and writing new ones. An even greater benefit of the ALLCACHE implementation of the shared memory model is its superior scalability and performance over mainframes, and message passing MPPs.

The performance advantages of ALLCACHE and shared memory can be characterized by its machine and programming aspects. Breit, etal [21] discuss these advantages with respect to examples from numerically intensive computing and Reiner, etal [22] discusses these advantages with respect to database management and inquiry. The key machine dependent aspects are:

- Memory allocation and data movement are handled automatically by the ALLCACHE Engine and incur no processor overhead. In contrast, for MPPs, memory allocation and data movement are handled explicitly by the programmer and are executed on the processor, incurring overhead to the application.
- Dynamic memory allocation and data movement of ALLCACHE requires less bandwidth and aggregate storage than MPPs – by dynamically optimizing locality.
- Hardware based parallel directory, routing and coherency management of the ALLCACHE Engine (as opposed to these functions executed on a conventional processor) achieve higher efficiency data sharing.

- ALLCACHE Engine supports efficient movement and sharing of fine granularity of data, which is well matched to a wide range of application requirements.

These characteristics result in higher effective bandwidth to a processor and between processors at the application level. The key programming aspects are:

- Incremental parallelization using the 90/10 rule: since the majority of the execution time occurs in a small percentage of the program, the shared memory programmers can spend all their time optimizing the part of the code that counts, and ignoring the remainder, since shared memory automates memory allocation and data movement. In contrast, for MPPs, programmers spend most of their time on the code that does not have any performance leverage.
- Over the past 15 years, algorithmic improvements in many fields have proceeded at a rate equal to machine improvements. [23] Shared memory's ease of programming allows programmers to implement the latest, most computationally efficient algorithm.
- For many new applications the ultimate Amdahls Law is the elapsed time for the program to execute the first time. The conventional programming environment enabled by shared memory provides the shortest route to solution.

Although the shared memory programming model is well known for its ease of programming, the most significant advantage of the Kendall Square implementation of a scalable shared memory model is higher performance.

References

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. Smith, Alan J. Cache Memories ACM Computing Surveys, 14 (3): 473-530, September 1982.
4. Organic, E.I., "The Multics System: An Examination of Its Structure," Cambridge, MA: MIT Press, 1972.
5. Bell, C. Gordon. "Multis: A New Class of Multiprocessor Computers," Science, Vol. 228, pps. 462-467, 26 April 1985.
6. Wulf, William A. and Bell, C. Gordon. "C.mmp-A multi-miniprocessor," Proceedings, AFIPS 1972 Fall Joint Computer Conference, 41, pp. 765-777, 1972.
7. Frank, Steven J. "Tightly Coupled Multiprocessor System Speeds Memory Access Times," Electronics, pps. 164-169, 1984.
8. Seitz, Charles L. "The Cosmic Cube," Communications of the ACM, 28-1, pps. 22-33, January, 1985.
9. Dally, William L. "The J-Machine: A Fine-Grain Concurrent Computer," MIT VLSI Memo 89-532, May, 1989.
10. 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.
11. Minnich, Ronald G. and Farber, David J. "The Mether System: Distributed Shared Memory for SunOS 4.0," (private communication).
12. Swan, R., Fuller, S., and Siewiorek, D. "Cm*- A modular, multi-microprocessor," Proceedings AFIPS 1977 Fall Joint Computer Conference, 46, pps. 637-644, 1977.
13. 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.
14. 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.
15. Lenoski, Daniel, Laudon, James, Gharachorloo, Kourosh, Wolf-Dietrich Weber, Gupta, Anoop, and Hennessy, John. "Overview and Status of the Stanford DASH Multiprocessor," Proceedings of International Symposium on Shared Memory Multiprocessing, pps. 102-108, April, 1991.
16. Kendall Square Research Corporation, "Technical Summary," 1992.
17. 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.
18. 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.
19. Leiserson, Charles E. "VLSI Theory and Parallel Supercomputing," Pasadena, CA: Proceedings of the 1989 Decennial Caltech Conference, March, 1989.
20. Burke, E. "An Overview of System Software for the KSR1," *Compcon '93 Proceedings.*
21. Breit, S., Pangali, C. and Zirl, D. "Technical Applications on the KSR1: High Performance and Ease of Use," *Compcon '93 Proceedings.*
22. Reiner, D., Miller J. and Wheat, D. "The Kendall Square Query Decomposer," *Compcon '93 Proceedings.*
23. "Grand Challenges: High Performance Computing and Communications: The FY 1992 U.S. Research and Development Program," Committee on Physical, Mathematical, and Engineering Sciences, 1991.