Interleaving and Hashing for Start System

Computation Structures Group Memo 335
August 10, 1991

Tetsuhide Senta

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.
The author is also with the NEC Computer Engineering Division, Fuchu-city, Tokyo.

Interleaving And Hashing For Start System

Tetsuhide Senta
CSG, Laboratory for Computer Science
Massachusetts Institute of Technology

August 10, 1991

Abstract

Start is a newly proposed multiprocessor system consisting of up to 64 thousand processing elements (PEs) interconnected with a network. Each PE has local memory which can also be addressed by a unique global address. In this paper an effective way to distribute global structures onto multiple PEs is discussed. First, interleaving, which is a mechanism to distribute the structures on 2^n PEs is introduced. Then hashing, which enhances the memroy utilization under constant stride access by irregulary permutating the distribution, is introduced. Some simulation was done to decide a fitting hash pattern. With that hash pattern, the over all utilization of the memory, under regular access pattern with constant strides, improved from that of interleaving without hashing. Then the mechanisms were extended in two ways to distribute global structures on $2^n - k$ PEs. The mechanism using re-interleaving has less loss of memory compared to the other using modulo table.

Contents

List of Figures

List of Tables

1 Introduction

Start is a newly proposed multiprocessor system consisting of up to 64 thousand processing elements (PEs) interconnected with a network. Each PE has local memory which can also be addressed by a unique global address. This paper describes interleaving and hashing mechanism, which makes global to local address translation efficient for the Start system. In Chapter 2 the global memory access of Start system is described. In the following chapters, interleaving, a mechanism to distribute a segment on multiple processors, and hashing, a mechanism to irregularly permutate the distribution, are explained. First, interleaving and hashing on 2^n processors is described in Chapter 3. A simulation model and its results on several hashing functions are also introduced in this chapter. Then, the mechanism is extended to support any numbers of processors in Chapter 4. Finally Chapter 5 gives a conclusion.

2 Global Memory Access On Start

This chapter describes the overview of Start remote memory access. The Start system has a distributed global memory system. That is to say, it has local memory distributed to the PEs. The local memory can not only be accessed within a PE, but can also be addressed with global addresses from other PEs. Frames active on PE are located in the local memory space so that access to them is very fast. On the other hand, there is a need for global data structures shared among PEs. Start implements global data structures by distributing them among the PEs. When a PE accesses a global data structure, it has to send a request to another PE through the network. This is called remote memory access.

Figure 1 shows the mechanism of remote memory access. When the data processor wants to access a global data structure, it issues a remote memory access to the sync processor. The sync processor takes remote memory access command from the data processor and performs address translation from global virtual address, which consists of segment number and offset, to PE number and local virtual address. This translation will be explained in detail in the following two chapters. After address translation, the sync processor sends a remote memory request packet into the network. The packet includes a continuation, which gives information on where to send the data back, as well as a destination PE number and local virtual address. If the destination PE happens to be the source PE itself, instead of sending off the packet into the network, it passes the request to the RMEM processor which is a part of sync processor. Otherwise, the network takes the request and delivers it to the destination PE according to the PE field of the packet. When the destination PE receives the remote memory request packet, the RMEM processor translates the local virtual address into a local real address. This is done the same way the address translation is done on the data processor for the local frame access. The RMEM processor fetches the data from local memory with the local real address, and sends it back to the continuation as a message. In Figure 1, the message is sent back to the source PE. However, the continuation need not point back to the source PE.

The rest of this paper describes a proposal for an efficient global to local address translation mechanism.

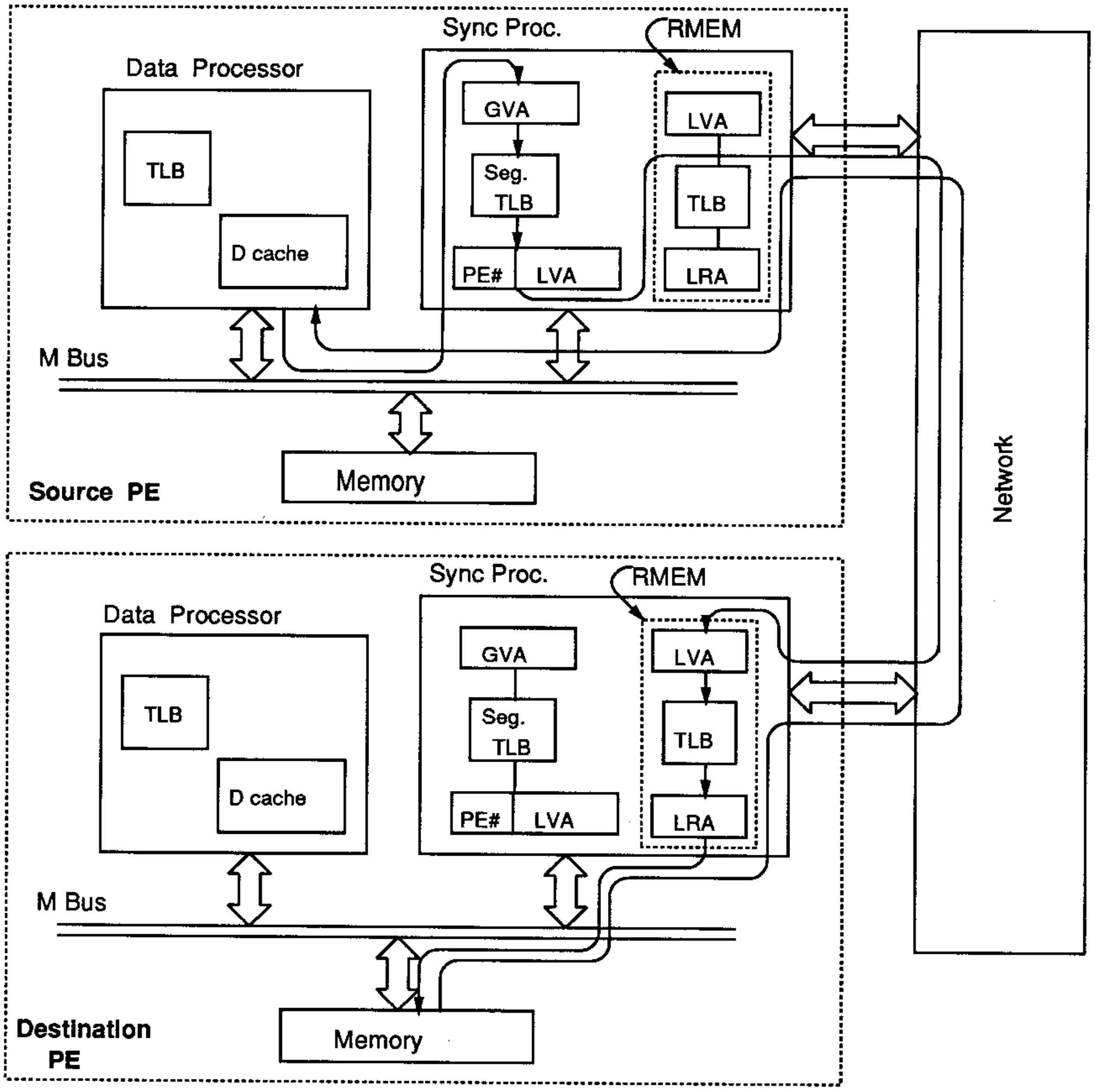


Figure 1: Remote Memory Access of Start

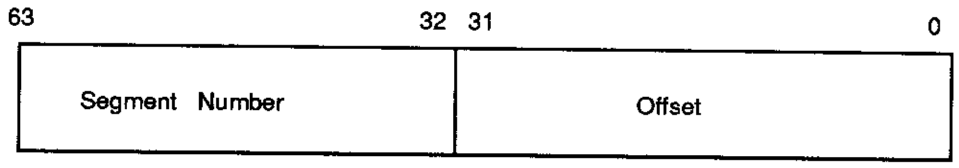


Figure 2: Global Virtual Address

3 Interleaving and Hashing For 2^n Processors

3.1 Interleaving

On Start, global data structures, which are shared among PEs, are physically distributed on PEs. It is convenient to give these structures unique names so all the PEs can access the structures similarly. The global virtual address space is the collection of these names.

Start has hardware support for translation from a global virtual address to a PE number and a local virtual address. A global virtual address consists of 32 bits of segment number and 32 bits of offset (Figure 2). A local virtual address is a 32 bit address, addressing the PE's local memory (Figure 3). There are two good reasons for having global virtual addresses, other than giving a unique naming space for the system. Firstly, to give data processor (which only has 32 bit addressing) a means to access a 64 bit address space by extending the address with a 32 bit segment number. Secondly, to give segments their own characteristics. For example, segments can be protected from destruction by making them read only.

Before explaining address translation in detail, usage of segments is presented to give a general image of the global virtual address. Each segment has its own base address and interleave value. The base address shows the location of a segment within the global logical address space (which consists of logical PE number and local virtual address). The interleave value tells among how many PEs the segment is interleaved. Being interleaved, the segment is divided into 16 byte blocks (D-cache line size) and distributed among PEs (Figure 4). This balances the memory request to each PE. Some segments can be interleaved on multiple PEs, while others are not interleaved and reside on a single PE (Figure 5) at the same time. Data shared among multiple processors should be allocated in interleaved segments, while the data locally used by single PE should be allocated in segments which are not interleaved.

Now let's look at the address translation in detail. First, the segment descriptor is explained. A segment descriptor is shown in Figure 6. Segment descriptors has entries attributes, hash bit, interleave, LVA (Local Virtual Address) base and segment size. The attributes entry includes information used to check whether the request to the segment is appropriate or not. For example, segments with protection bit on can only be accessed with kernel mode access, segments with read only (RO) bit on can only be read. I-structure access and M-structure access can only be issued to segments with I bit and M bit on accordingly. When these checks are violated, sync processor will cause a trap and the software on it will handle the exception. The IO bit in the attribute controls whether the segment is interleaved on IOPEs or not (refer to Chapter 4). The hash bit controls whether hashing is

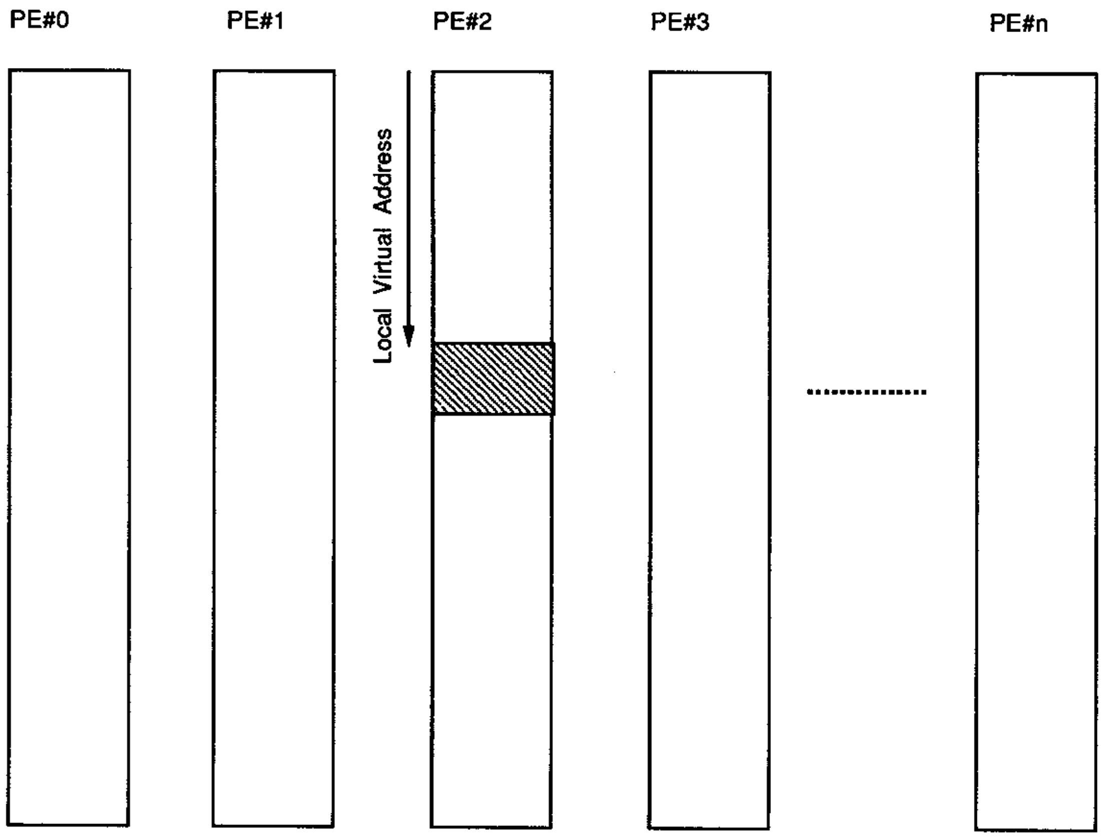


Figure 3: Local Virtual Address

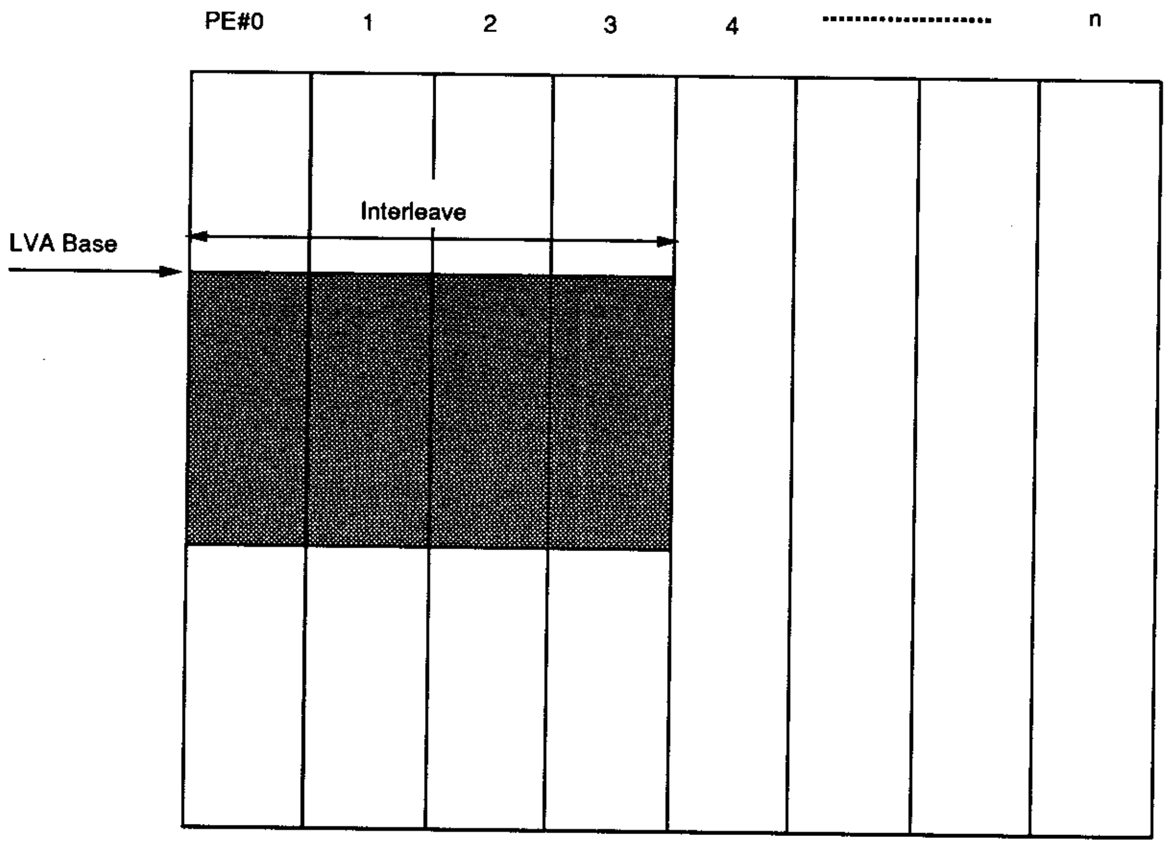


Figure 4: LVA Base, Interleave, Segment Size

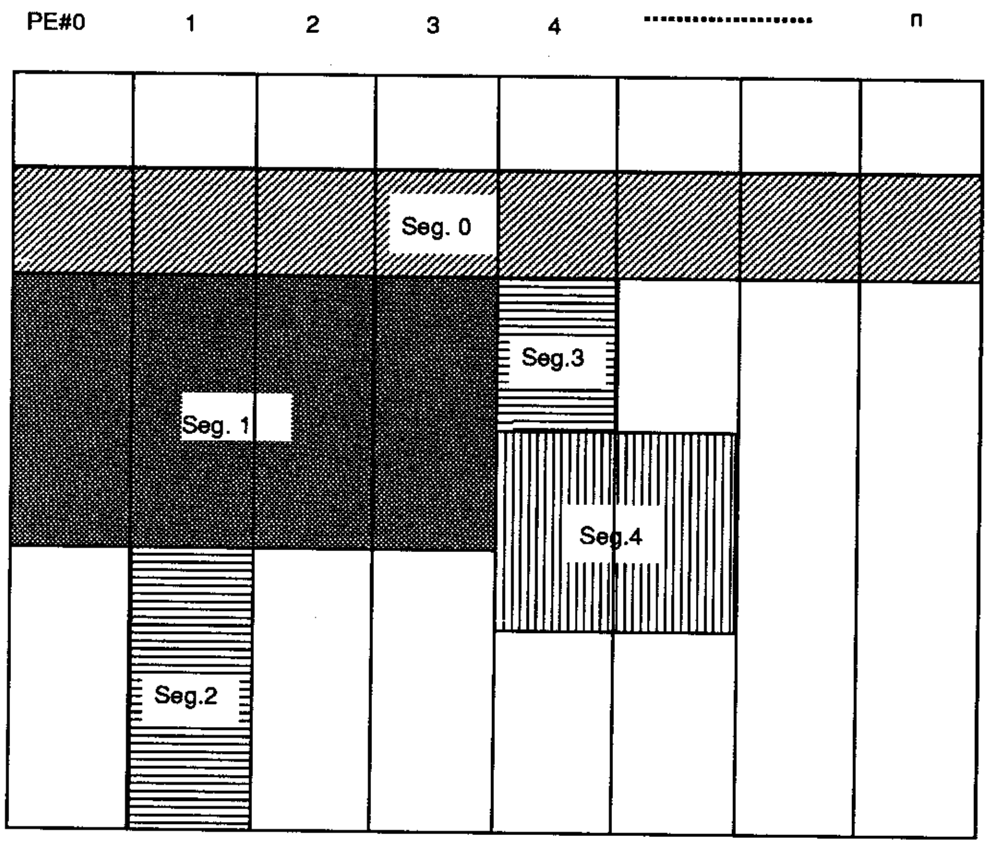


Figure 5: Interleaved Segment

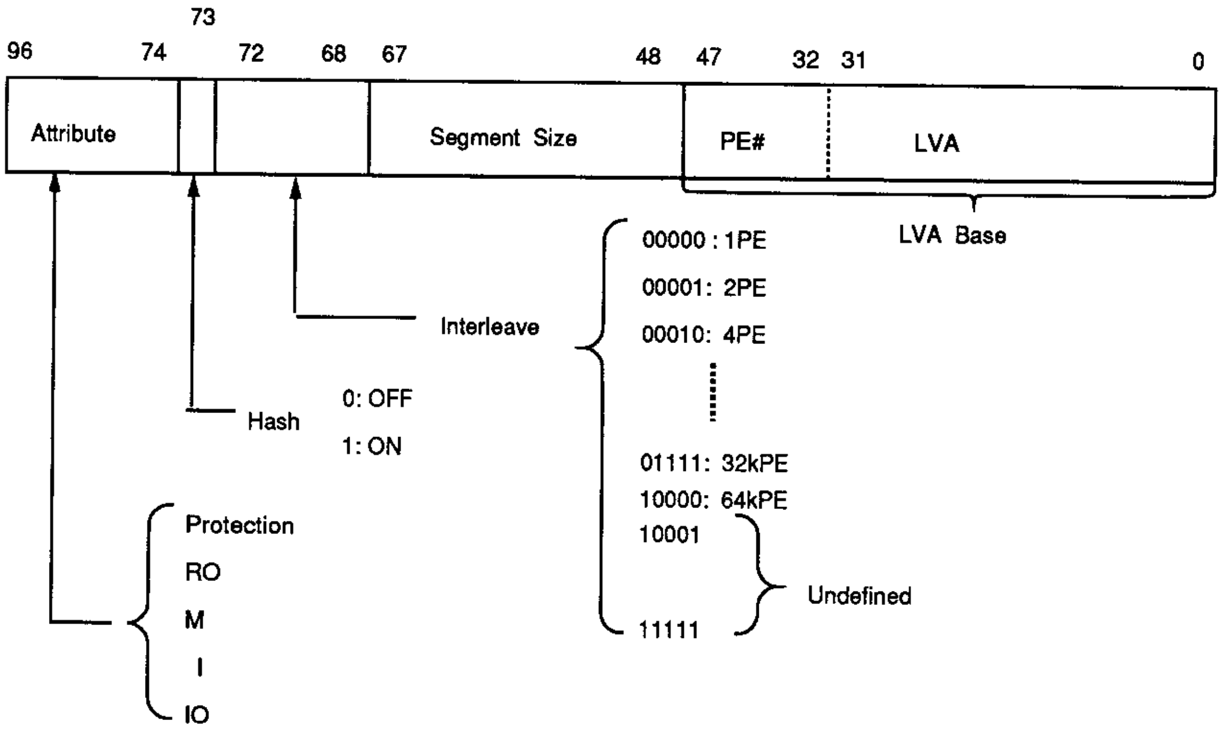


Figure 6: Segment Descriptor

performed for that segment or not. Hashing will be described in Section 3.2. 5 bit interleave value gives information showing the number of PEs among which the segment is interleaved. Interleaving among 1 to 64 thousand PEs can be encoded as shown in the figure. The 16 bit PE number and 32 bit LVA in the 48 bit LVA base entry is used to define where in the global address space the segment is located. The content of the LVA base entry will be explained in the next paragraph. The 20 bit segment size is used for checking access beyond the segment boundary. This check is necessary to allow variable segment size. Segment size holds the upper 20 bits of the 32 bit actual segment size. The actual segment size has to be a multiplier of 4k bytes (page size).

Address translation with segment descriptor is performed as shown in Figure 7. Input for the translation is a global virtual address sent to the sync processor from the data processor along with a remote memory access request. The segment number which is the upper 32 bits of global virtual address, is used to index the segment descriptor. Limited numbers of segment descriptors are held in a hardware buffer called the segment TLB. In case of a segment TLB miss, sync processor causes a trap and software on the sync processor can take care of it as an exception. If the segment descriptor is found in segment TLB, then the attributes, hash bit, interleave, LVA base, and segment size are read from the segment TLB. Most of the attributes are sent to the violation checker. As shown in Figure 8, the start address of the segment is rotated and stored in the LVA base entry of the segment descriptor. This rotation should be done in advance by software, either at initialization or at TLB miss handling. The amount of this rotation is determined by the interleave value of the segment. The offset which is the lower 32 bits of the global virtual address is arithmetically added to the LVA base (the carry from LVA base to the PE number is ignored). At the same time, the offset is compared with segment size. If the offset is greater, a trap occurs on the sync processor causing an exception. Otherwise, the result of addition is XORed with the hash pattern generated by hash pattern generator to create the hashed PE number. Hashing is described in detail in the next section. Finally the result of addition is rotated back with the result of hashing to generate the PE number and LVA. The amount of the rotation here is determined by the interleave value from the segment TLB. This way, cache lines with consecutive offsets will be assigned to different PEs.

With the above mechanism, segments can be interleaved as shown in Figure 5 according to their descriptors.

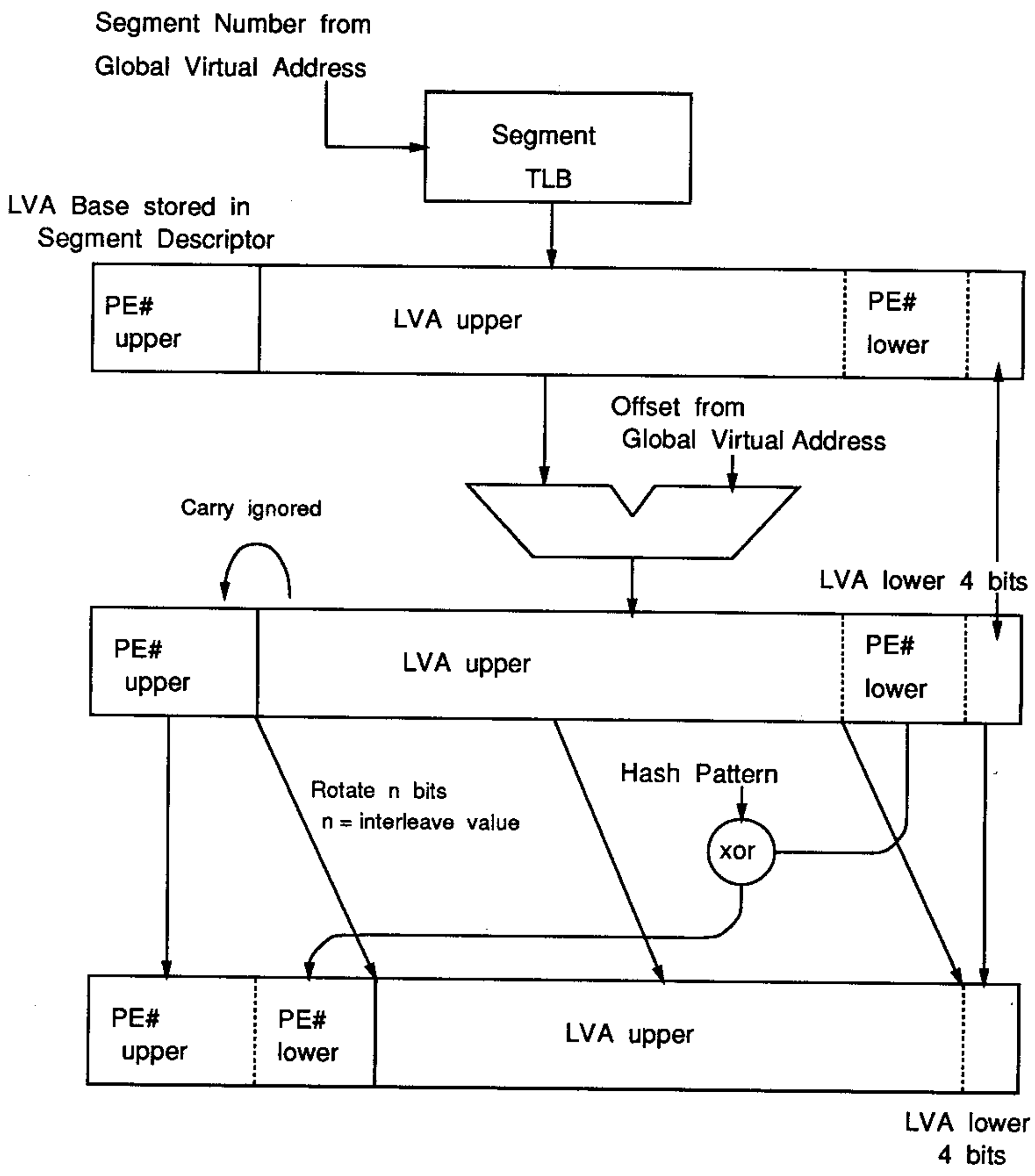


Figure 7: Address Translation

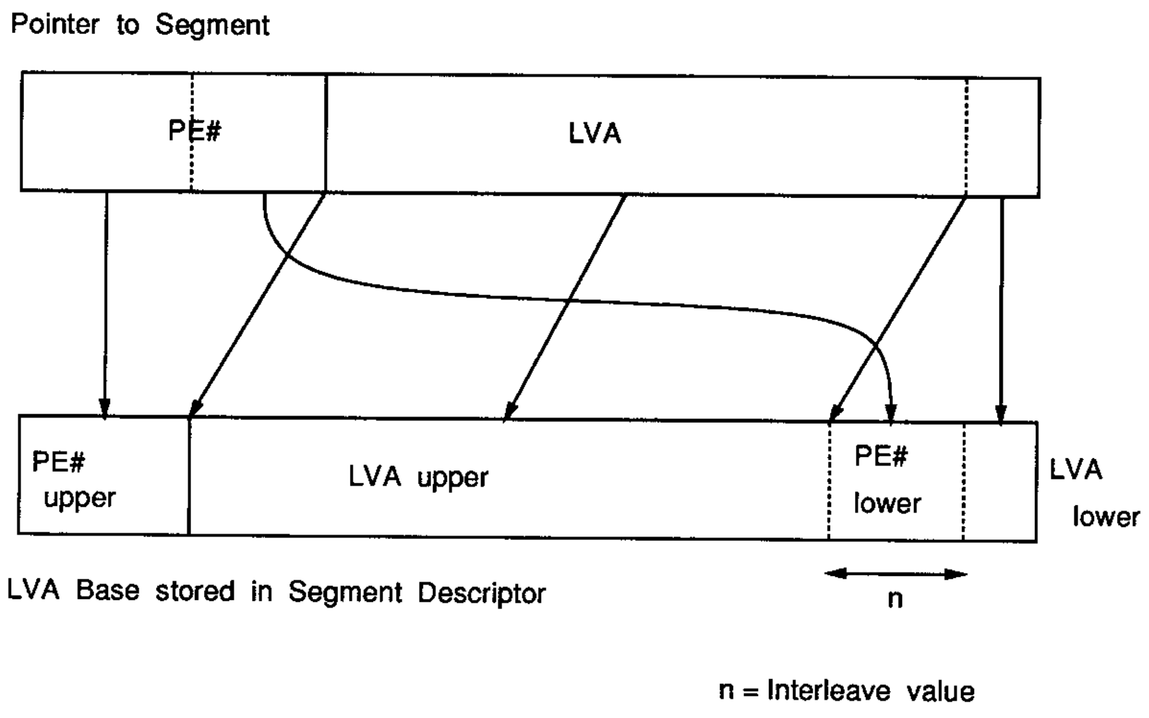


Figure 8: LVA Base in Segment Descriptor

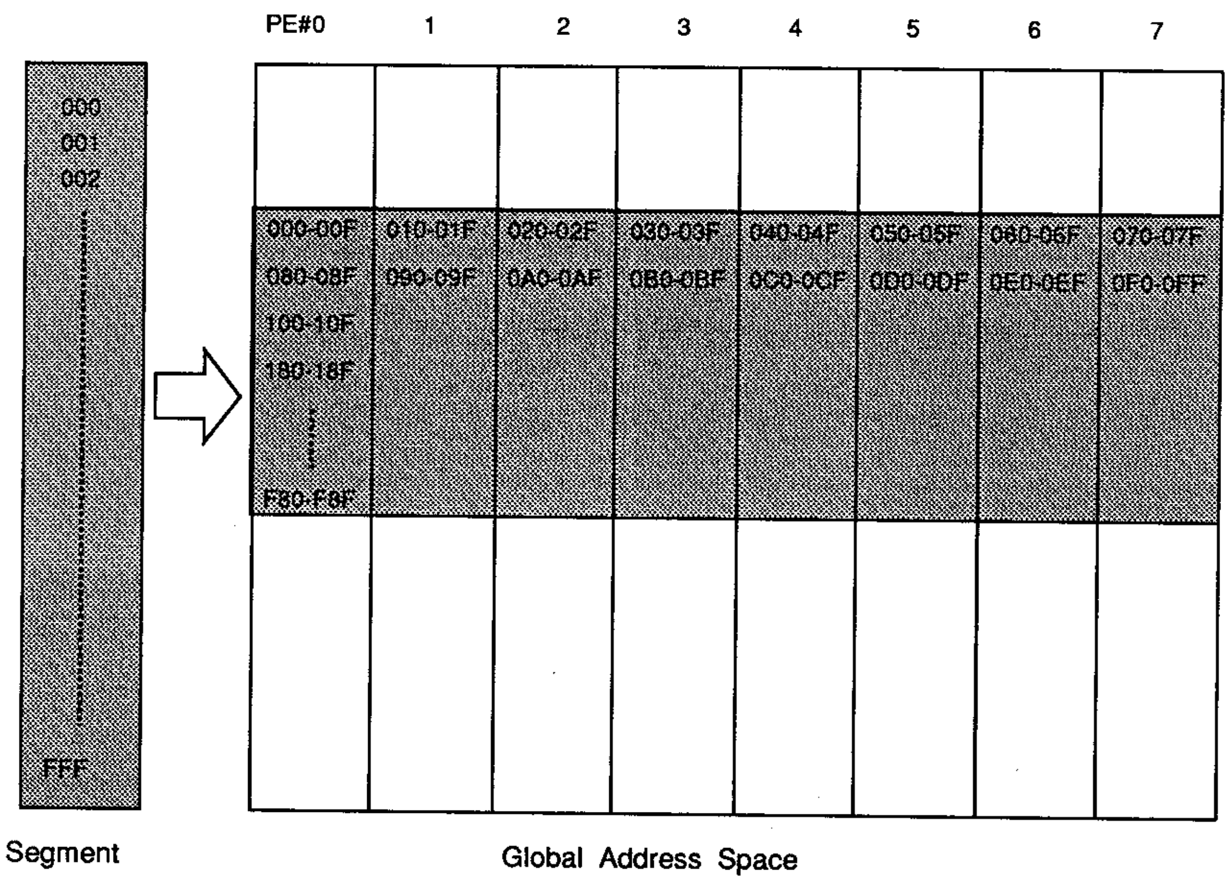


Figure 9: Interleaving

3.2 Hashing

Interleaving gives a means of distributing data structures among multiple PEs. But since it assigns PEs to each cache line in turns, the assignment is very regular (Figure 9). The regularity in interleaving causes a problem when the access pattern on the segment is also regular, such as accessing a column of an array by incrementing the index with a constant stride. When the interval is a multiple of a power of 2, the accessed elements are distributed only on a subset of the PEs assigned for the segment. This makes only part of the PEs busy, while others remain idle. For example, in Figure 9, every byte address which is a multiple of 80 hex is located on PE#0. Even if the segment is interleaved among 8 PEs, if the segment is accessed with a stride of 80 hex, only one out of the eight PEs will be busy while others get no requests. The load balancing effect of interleaving is not working in this case.

This can be a problem for sequential machines, but for parallel systems like Start, it is more severe. Each iteration of a loop can be executed on different PEs at the same time, and in a system with split phase memory access, like Start, the number of outstanding requests is much higher than in sequential machine. The performance of the system can dramatically decrease if only one PE has to process all of these outstanding requests.

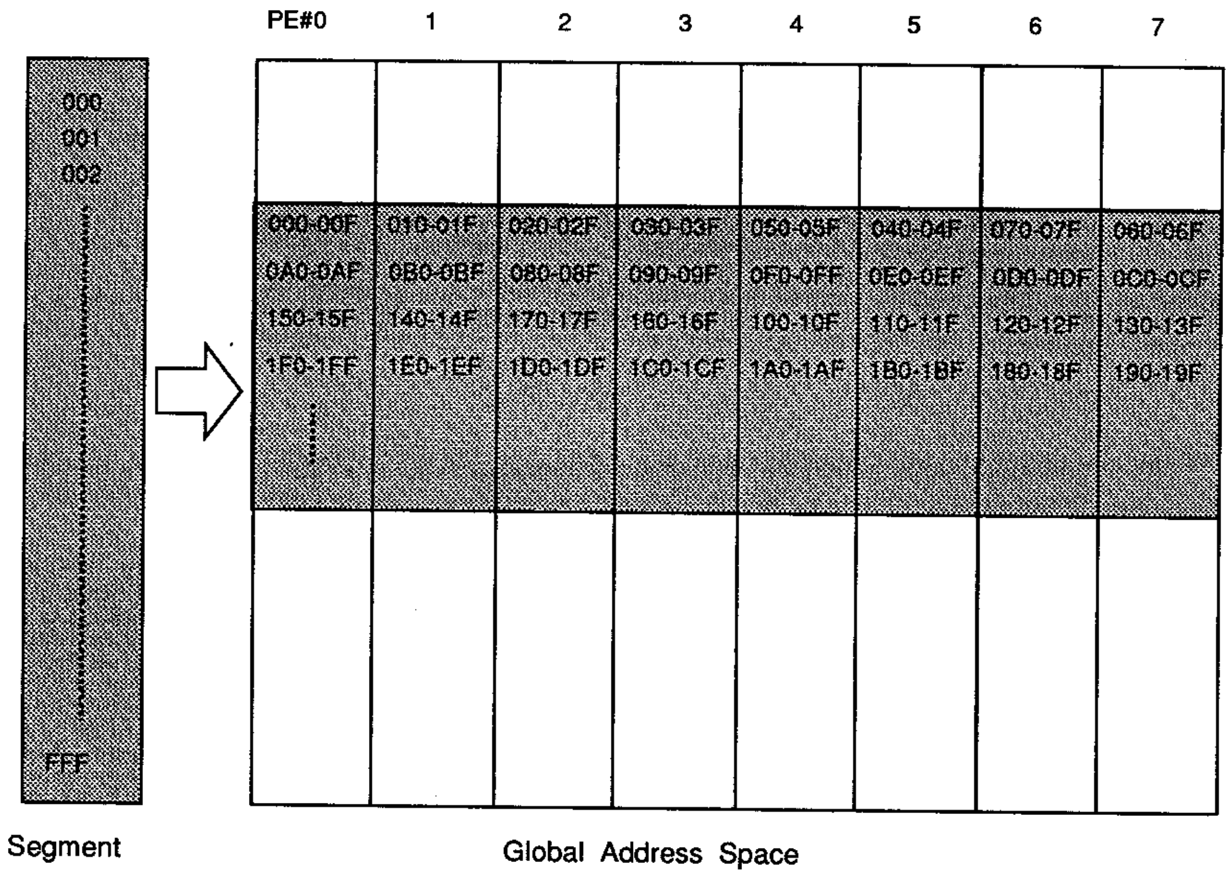


Figure 10: Hashing

Hashing is a way to assign a PE to a cache line irregularly by permutating the order of assignment. Figure 10 shows interleaving with hashing (which uses hash pattern 'fn3' in the following simulation). This has a better distribution compared to interleaving without hashing. Any function that gives a permutation can be a hashing function. However when we think of simplicity in hardware implementation, generating a hash pattern from the address and XORing it to the interleaved PE number seems to be a good solution. Figure 11 shows the idea of such a hashing function. Hash patterns can be generated by simply taking some of the bits in the address. From now on, bit locations of the cache line address are used. As the cache line size in Start is 16 bytes, bit 0 of the cache line address is equivalent to bit 4 of the byte address. Let's take bits 10..3 of the cache line address for an example hash pattern. The hash pattern changes as bits 10..3 in the address change. Each hash pattern represents a permutation pattern of 16 cache lines. With 256 PEs, if all 8 bits in a hash pattern change, it gives 256 kinds of permutation patterns, which is good. There are two reasons for some part of the bits to remain unchanged. First, when the stride constant is large, lower bits of hash pattern never change. This becomes more of a problem in systems with small numbers of PEs, because only the lower bits of hash pattern are used for hashing. In such case,

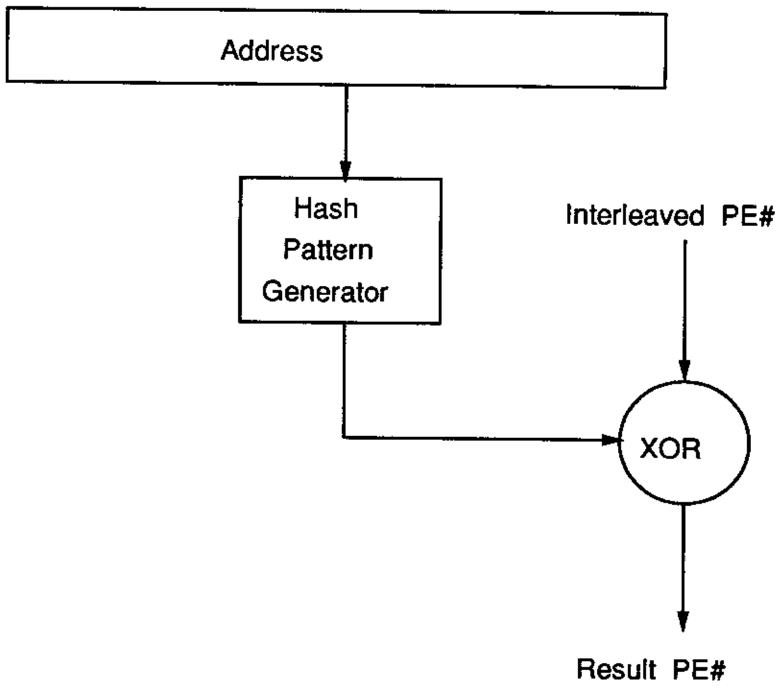


Figure 11: Hashing Function

the hash pattern will never change and the distribution is as bad as interleaving without hashing. The other case is when the stride constant is small and the number of requests is not large enough so the upper bits of the hash pattern do not change. In the first case, it is good to use the higher bits for the hash pattern, for example, bit 15..8. In the second case, it is better to use the lower bits, like bit 8..1. Is there a way to solve these contradictory requirements? One simple idea is to have two different hash pattern, one for a long stride and the other for a short stride. But, hashing has to be consistent for all requests to the same segment. This makes it impossible to have two hash patterns for the same segment.

Instead, the proposal in this paper is to XOR two or more bit patterns to generate a hash pattern. Instead of using bits like 10..3 or 15..8 as a hash pattern, we will use something like (bits 10..3) XOR (bit15..8). Comparison beween these hash patterns is made in Sec.3.3.

3.3 Simulations and Results

The following modeling of the system was done to evaluate several hash patterns.

- There is no network delay or confliction.
- m PEs can issue requests simultaneously. m is called issue rate.
- There can be only k outstanding requests in the whole system at a time (k bounded loop model). When there is k outstanding requests, no more requests can be issued until the data comes back.
- It takes t cycles from request issue to get the data even if the destination is idle. t is called access delay.
- If the destination is busy, request will be enqueued and taken care of after the destination becomes available.

Table 1 shows the memory utilization for different hash patterns with strides of power of 2. Utilization is defined as (Processing time required in the ideal case)/(Actual processing time). Processing time required in the ideal case is the time required to process all the requests when all the PEs are most effectively used in turn. Utilization decreases as some PEs receives more requests than others. Table 1 shows the utilization for four kinds of hash patterns. "No hash" is interleaving without hashing. "Low" uses bits 10..3 of cache line address as a hash pattern. "High" uses bits 15..8, and "Xor" uses (bits 10..3) XOR (bits 15..8). Both the number of outstanding requests and the issue rate are the same as the number of PEs. Access delay is also assumed to be the same as the number of PEs. Unit of strides in all the simulations is cache line (stride 2 means 2×16 bytes).

The results showed that "No Hash" has the worst and "Xor" has the best utilization in all cases. "Low" showed better utilization than "High" with 4PEs, stride 4. In other cases, "High" showed better utilization than "Low". "Xor" has the utilization same as "Low" in 4PEs, stride 4, and same as "High" in other cases, taking good parts of both.

Table 2 shows the memory utilization for odd strides. Both number of outstanding requests and issue rate are the same as the number of PEs. Access delay is also assumed to be the same as the number of PEs. The result showed that for odd strides, "No hash" shows the best utilization. It showed perfect utilization in all cases. Comparison between other three hash patterns is not so obvious. "High" shows better utilization than "Low" for 4PEs. For 32PEs, "Low" is better than "High" for stride 3 and 5. For stride 7 and 11, "High" was better. "Xor" does not have better utilization than "Low" or "High" like in power of 2 strides.

Table 1: Memory utilization for power of two strides

Hash pattern	Number of PE	Number of Requests	Utilization (%) Stride 4	Utilization (%) Stride 256
No hash	4	2*PE#	25.0	25.0
		32*PE#	25.0	25.0
	256	2*PE#	25.0	0.4
		32*PE#	25.0	0.4
Low	4	2*PE#	50.0	25.0
		32*PE#	72.7	25.0
	256	2*PE#	50.0	3.1
		32*PE#	72.7	3.1
High	4	2*PE#	25.0	100.0
		32*PE#	25.6	100.0
	256	2*PE#	100.0	100.0
		32*PE#	100.0	100.0
XOR	4	2*PE#	50.0	100.0
		32*PE#	72.7	100.0
	256	2*PE#	100.0	100.0
		32*PE#	100.0	100.0

Access delay: PE#
Issue rate : PE#
Number of outstanding requests : PE#

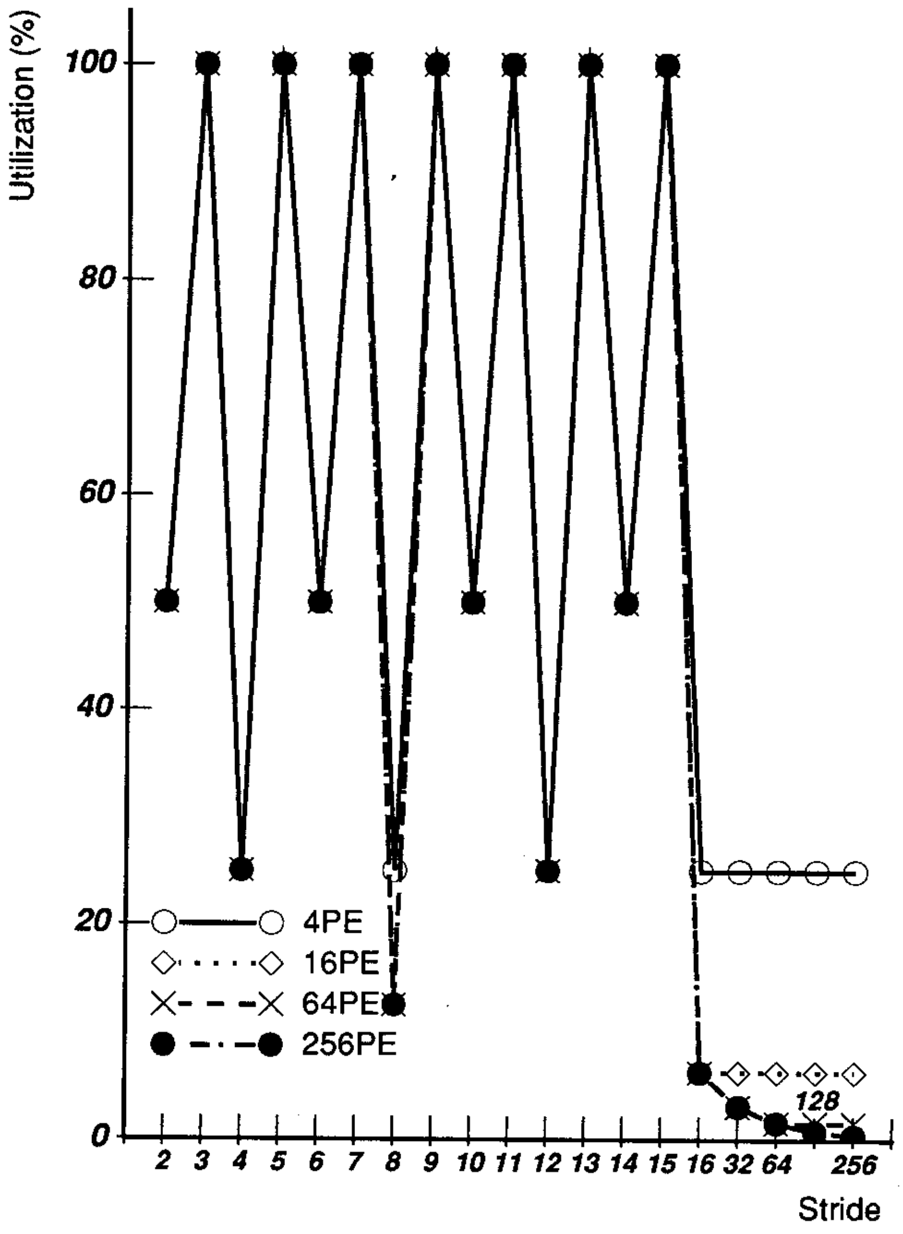
Table 2: Memory utilization for odd strides

Hash pattern	Number of PE	Number of Requests	Utilization (%) Stride 3	5	7	11
No hash	4	32*PE#	100.0	100.0	100.0	100.0
	256	32*PE#	100.0	100.0	100.0	100.0
Low	4	32*PE#	78.0	66.7	49.2	66.7
	256	32*PE#	65.3	60.4	53.3	58.2
High	4	32*PE#	97.0	97.0	97.0	97.0
	256	32*PE#	33.3	20.0	65.3	65.3
XOR	4	32*PE#	78.0	66.7	49.2	66.7
	256	32*PE#	62.7	60.4	60.4	55.2

Access delay: PE#
Issue rate : PE#
Number of outstanding requests : PE#

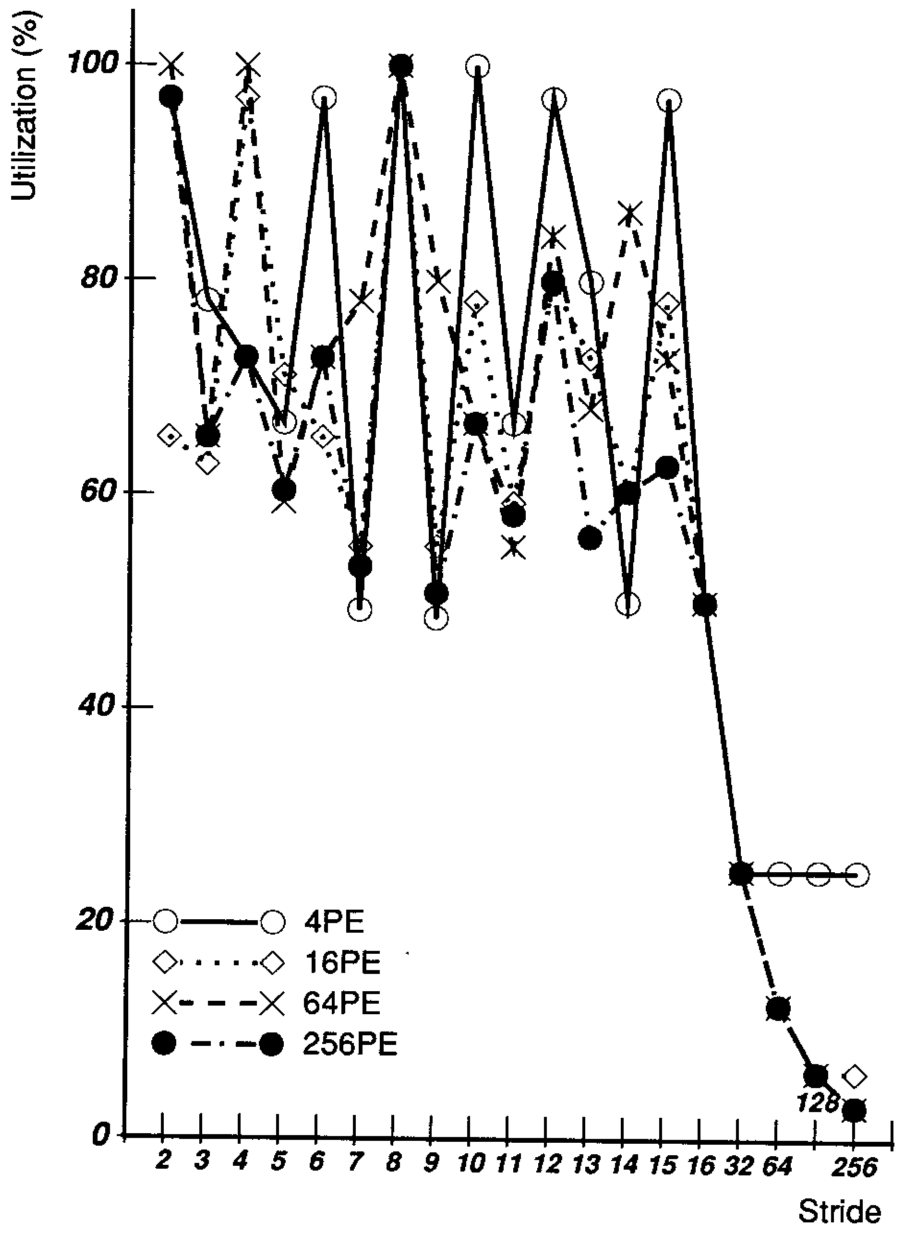
Figure 12 to 15 shows the memory utilization for the four hash patterns with 4,16,64,256PEs.

"Xor" has the best utilization as a whole. "No hash" has better utilization for odd strides but shows poor utilization for large numbers of PEs and the worst as a whole.



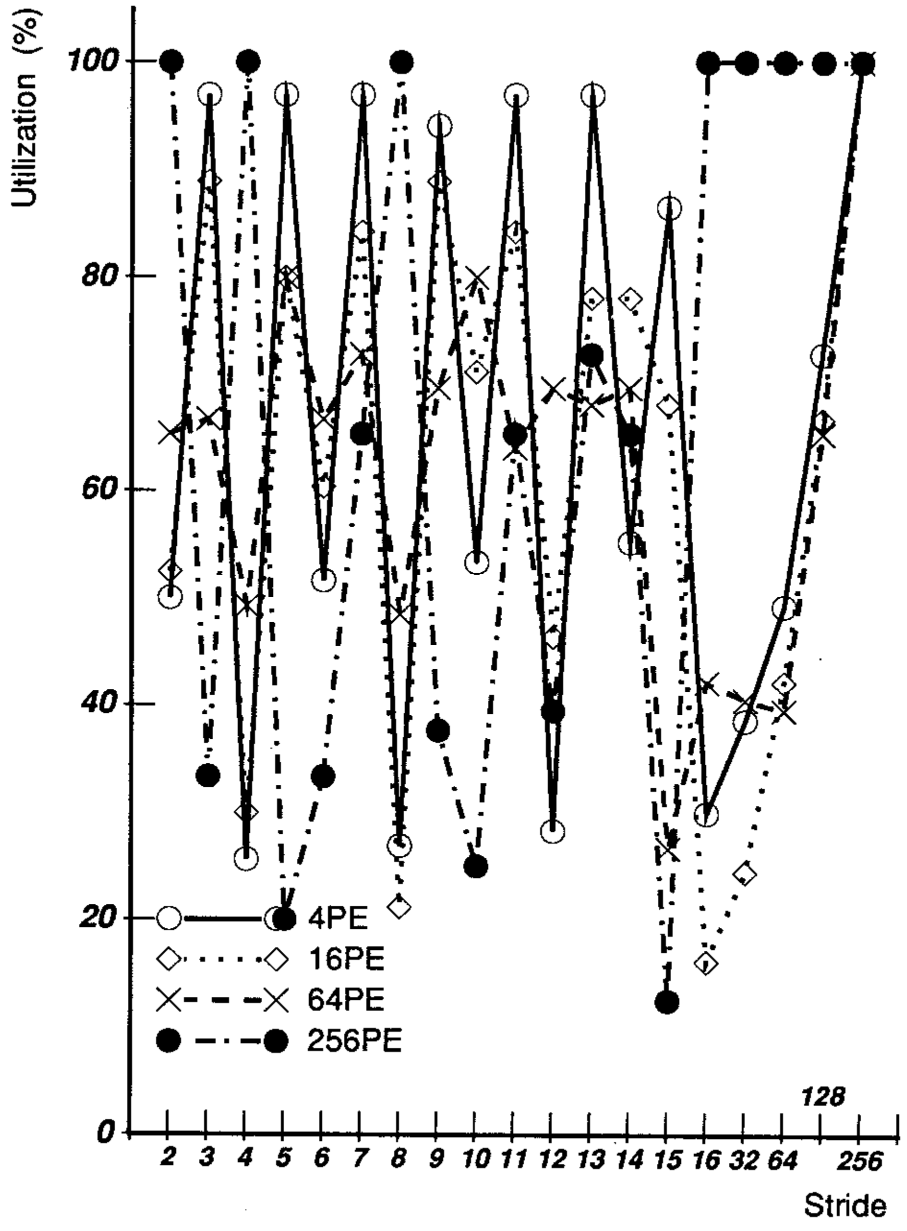
Hash Pattern : No hashing
Access delay : 16 cycles
Issue rate : PE#
Outstanding requests : PE#
Total number of requests : 32×PE#

Figure 12: Utilization for different numbers of PE



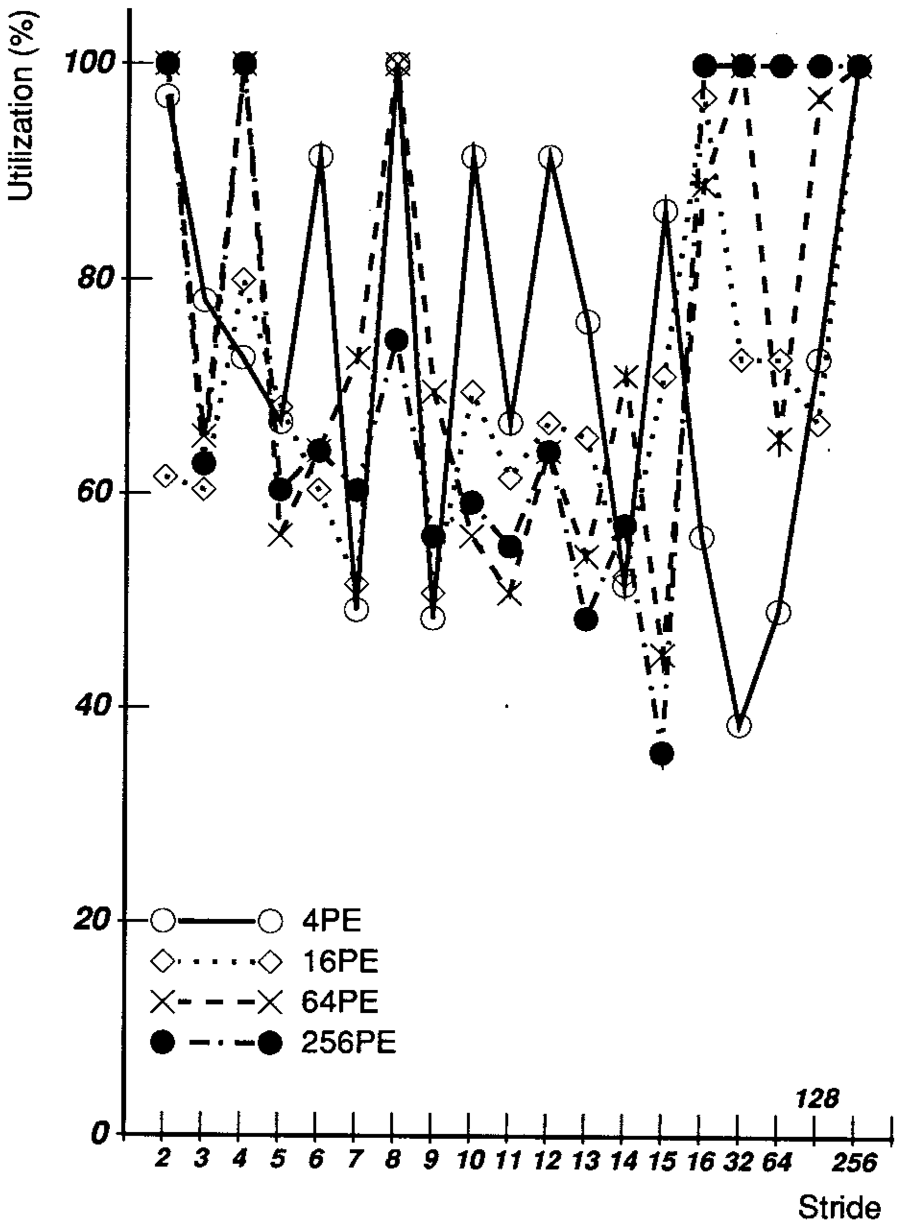
Hash Pattern : bit 10..3 (Low)
Access delay : 16 cycles
Issue rate : PE#
Outstanding requests : PE#
Total number of requests : 32×PE#

Figure 13: Utilization for different numbers of PE



Hash Pattern : bit 15..8 (High)
Access delay : 16 cycles
Issue rate : PE#
Outstanding requests : PE#
Total number of requests : 32×PE#

Figure 14: Utilization for different numbers of PE



Hash Pattern : (bit 10..3) XOR (bit 15..8) (Xor)
Access delay : 16 cycles
Issue rate : PE#
Outstanding requests : PE#
Total number of requests : 32×PE#

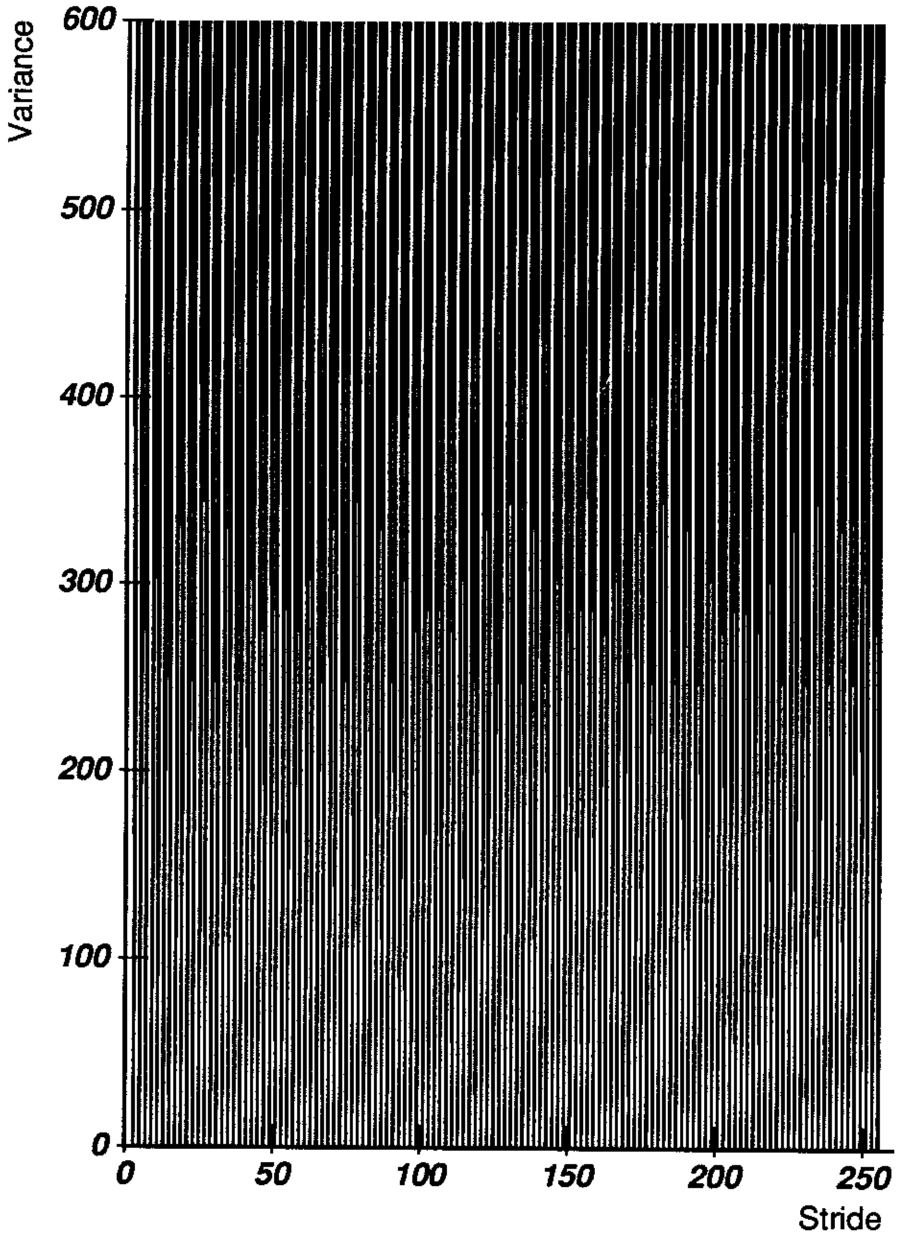
Figure 15: Utilization for different numbers of PE

Figure 16 to 19 shows variance in numbers of requests per PE.
Variance is defined as follows.

$$\sum_{for\ all\ PEs} \delta^2$$

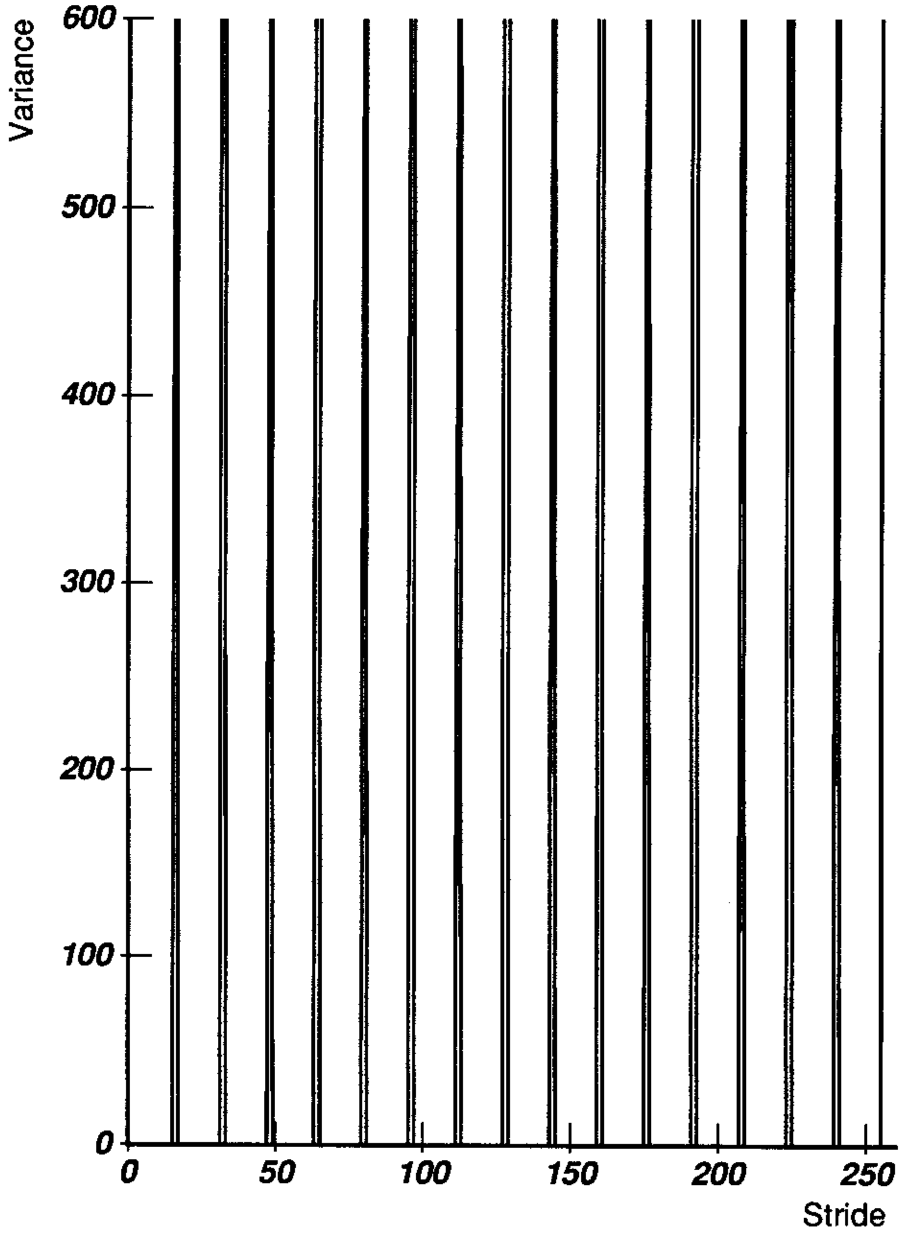
Where δ = (Total number of requests)/(Number of PEs)− (Occurrence of request on the PE). Even distribution of requests on all PEs gives zero variance.

"No hash" has very high variance for even strides and zero variance for odd strides. "Low" has high variance for strides multiples of 16. "High" has high variances for only few strides and better that "Low". "Xor" has the lowest variances of the four hash patterns. This means that with "Xor" hash pattern, the number of occurrence per PE is quite even for most of the strides.



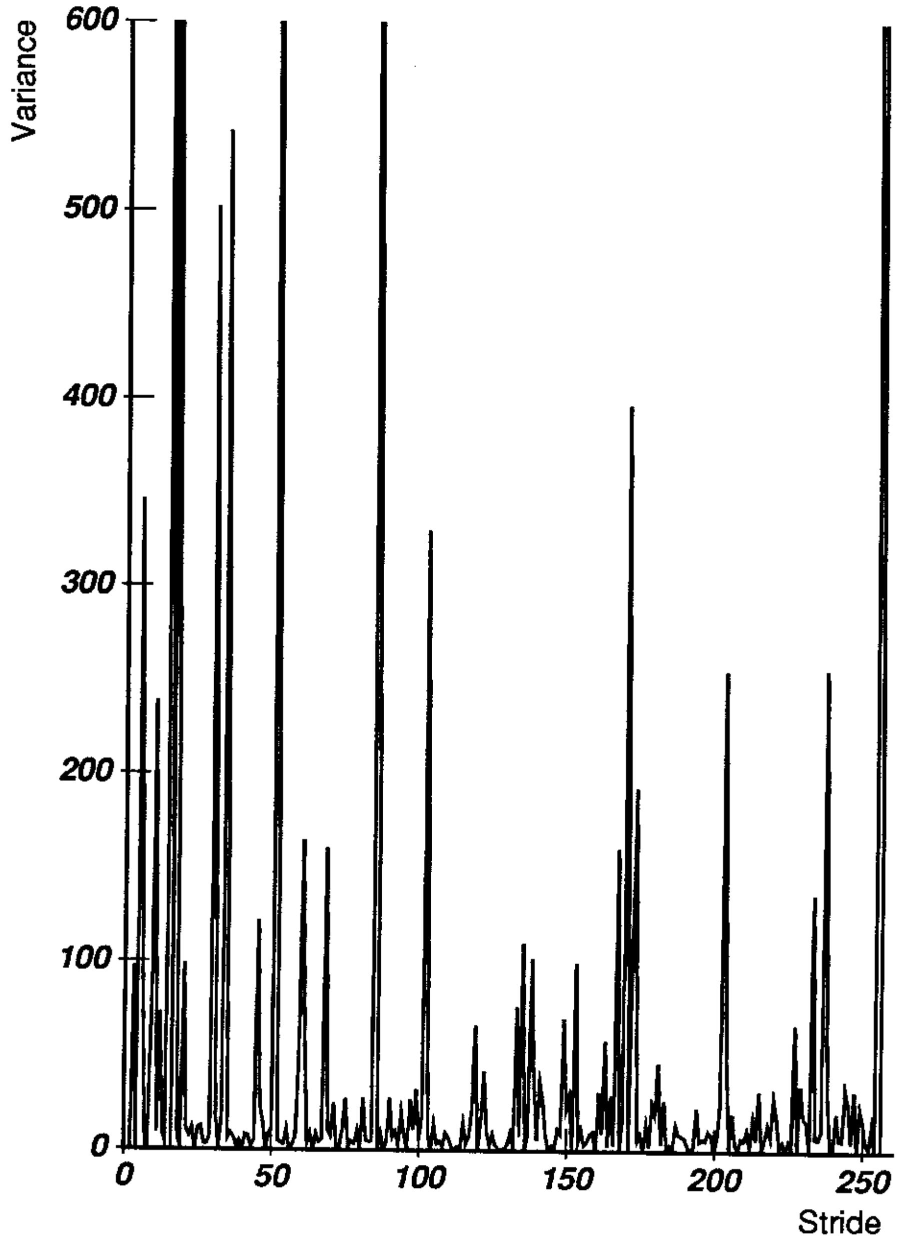
Hash Pattern : No Hashing
Number of PEs : 256
Access delay : 16 cycles
Issue rate : 256
Outstanding requests : 256
Total number of requests : 8096

Figure 16: Variance in numbers of requests per PE



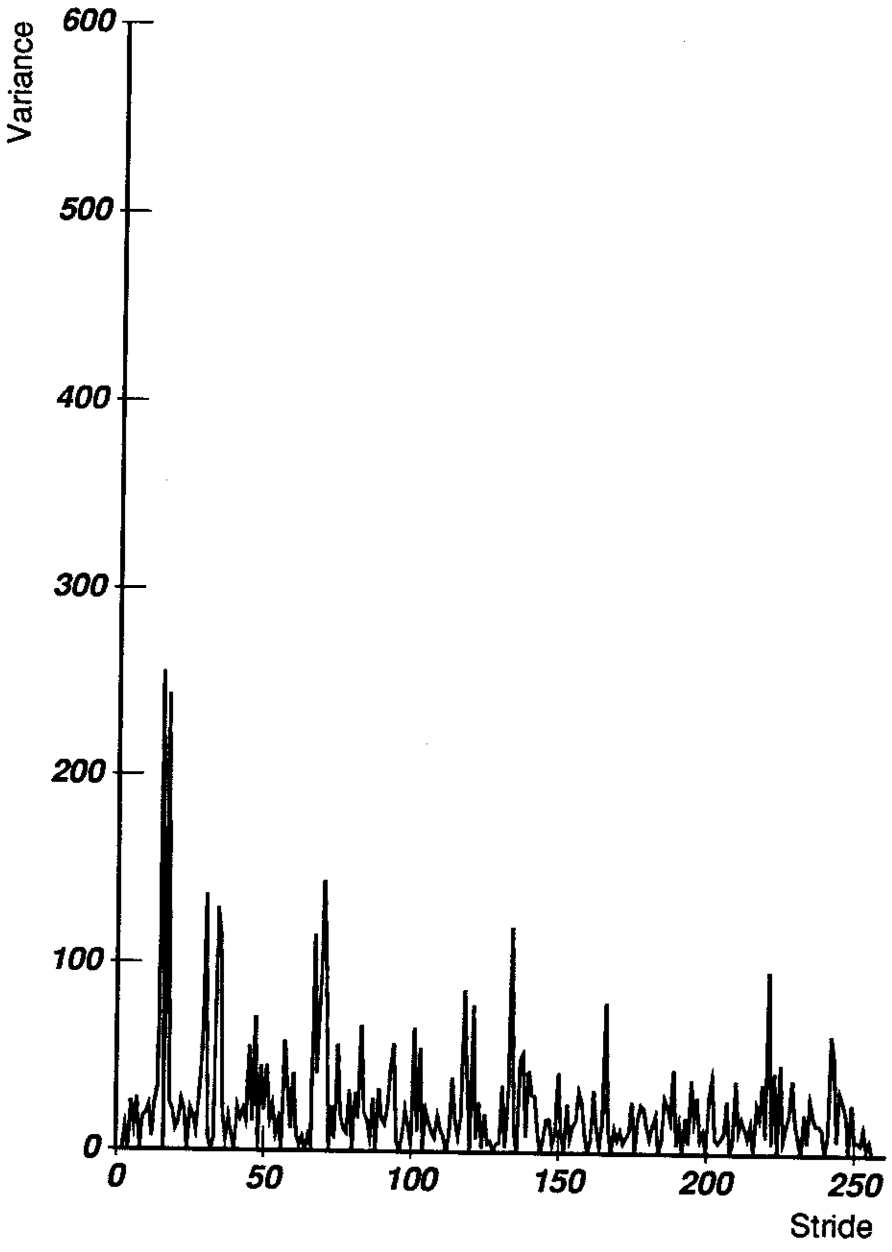
Hash Pattern : bit 10..3 (Low)
Number of PEs : 256
Access delay : 16 cycles
Issue rate : 256
Outstanding requests : 256
Total number of requests : 8096

Figure 17: Variance in numbers of requests per PE



Hash Pattern : bit 15..8 (High)
Number of PEs : 256
Access delay : 16 cycles
Issue rate : 256
Outstanding requests : 256
Total number of requests : 8096

Figure 18: Variance in numbers of requests per PE



Hash Pattern : (bit 10..3) XOR (bit 15..8) (Xor)
Number of PEs : 256
Access delay : 16 cycles
Issue rate : 256
Outstanding requests : 256
Total number of requests : 8096

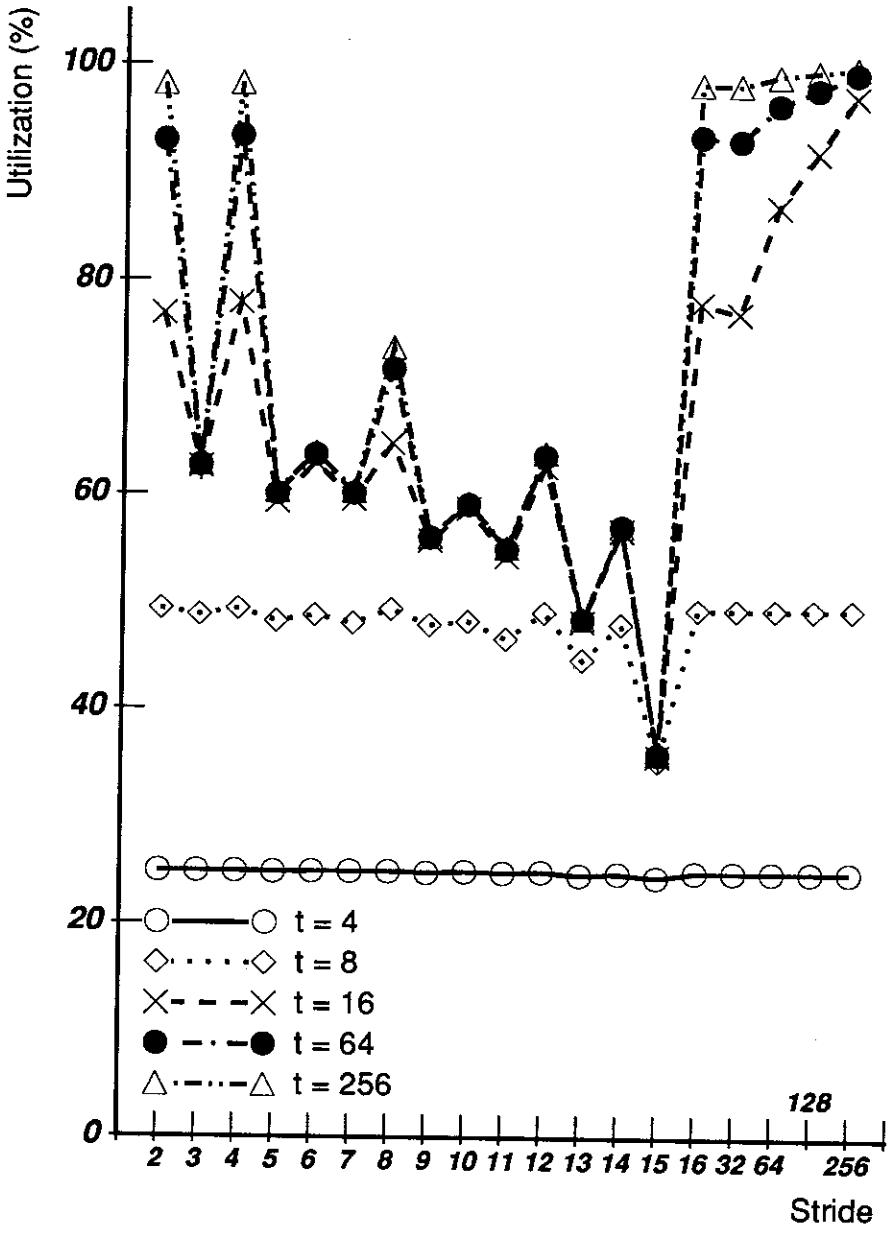
Figure 19: Variance in numbers of requests per PE

Figure 20 to 22 shows how the change in some simulation parameters affects the memory utilization for the hash pattern "Xor" .

Figure 20 shows the memory utilization for different access delays. As access delay decreases, the utilization decreases. This is because the issue rate here is smaller than the number of PEs. This way, if the requests terminate quickly, low request issue rate cannot keep the number of outstanding request high. For access delay smaller than 16, memory issue cannot keep up with request termination so the utilization is limited. With a issue rate equal to the number of PEs, the difference in the access delay did not affect the utilization at all.

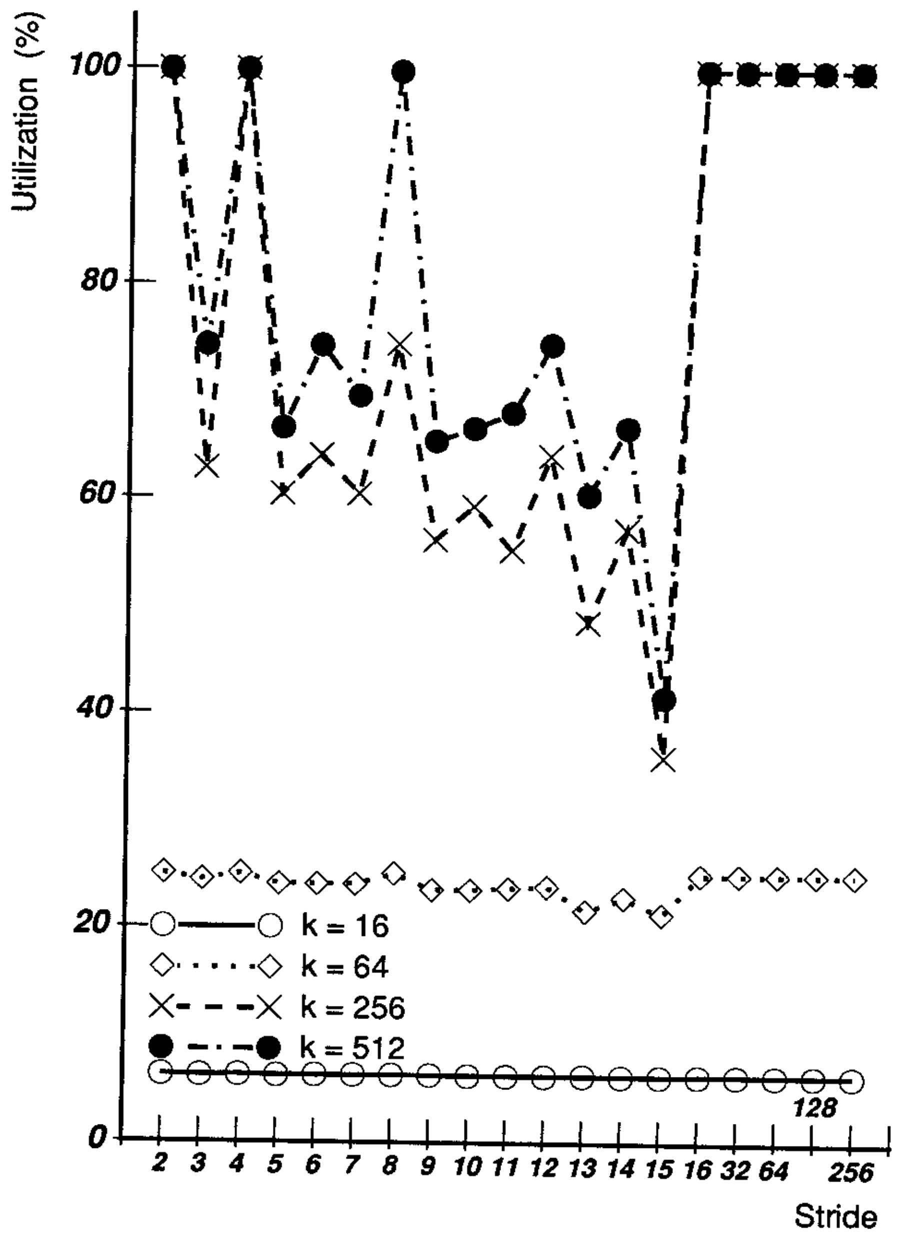
Figure 21 shows the memory utilization for different numbers of outstanding requests. The result shows some improvement when the number of outstanding requests increases from 256 to 512. With the number of outstanding requests smaller than the number of PE, the system cannot make full use of all the PEs, and the utilization is limited.

Figure 22 shows the memory utilization for different issue rate. Change of issue rate from 256 to 64 did not affect the utilization very much. With issue rate smaller than 16, memory issue cannot keep up with request termination and the utilization is limited.



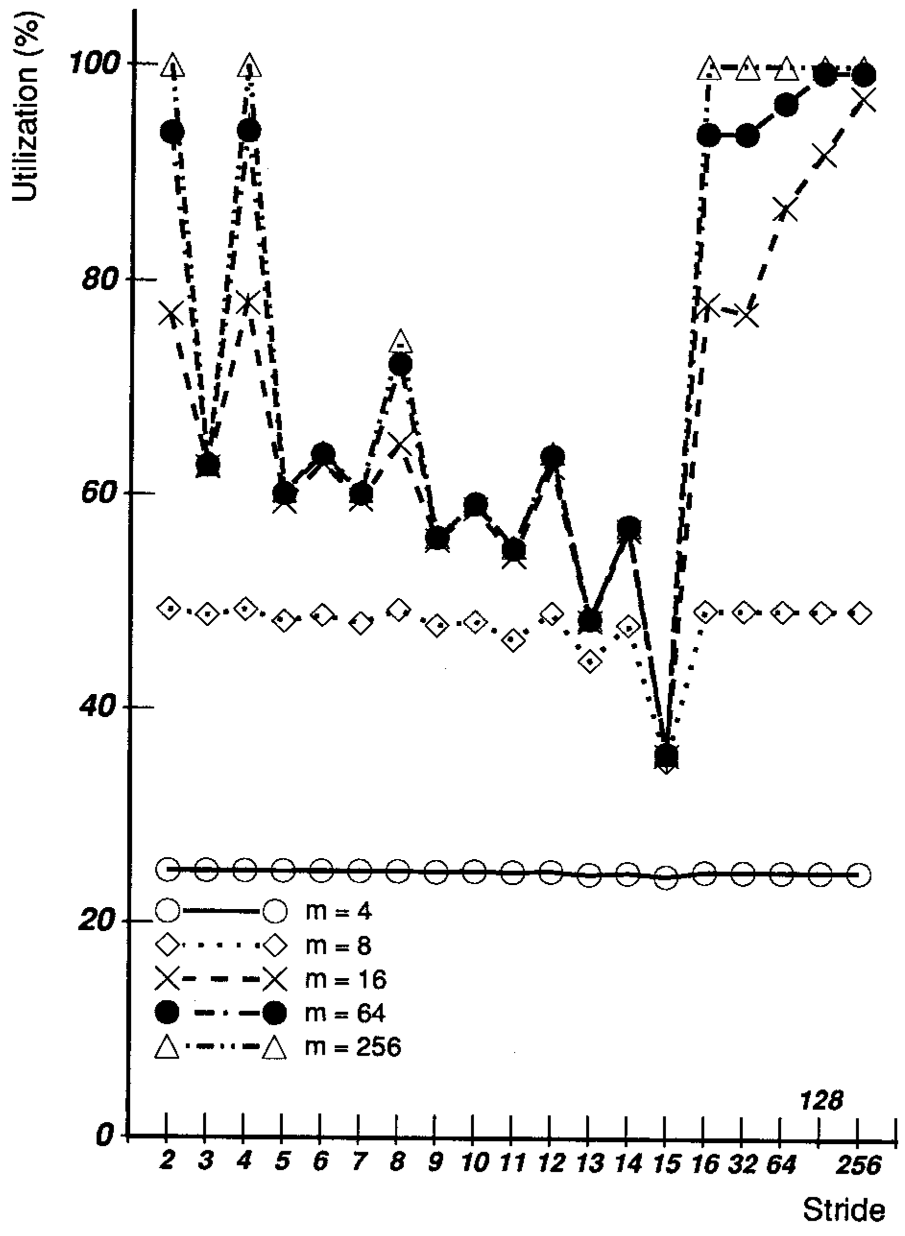
Hash Pattern : (bit 10..3) XOR (bit 15..8) (Xor)
Number of PEs : 256
Issue rate : 16
Outstanding request : 256
Total number of requests : 8096

Figure 20: Utilization for different access delay



Hash Pattern : (bit 10..3) XOR (bit 15..8)   (Xor)
Number of PEs : 256
Access delay : 16 cycles
Issue rate : 256
Total number of requests : 8096

Figure 21: Utilization for different numbers of outstanding requests



Hash Pattern : (bit 10..3) XOR (bit 15..8) (Xor)
Number of PEs : 256
Access delay : 16 cycles
Issue rate : 256
Total number of requests : 8096

Figure 22: Utilization for different issue rates

Table 3 and 4 shows the average utilization for various hash patterns.

The average was calculated as harmonic means for the number of PEs of 4, 16, 64, 256, strides of 2, 3, 4, 5, 7, 8, 16, 64, 256, issue rates of 1/4, 1/2, 1, 2 times the number of PEs. Access delay is the same as the number of the PEs. The simulation was done with two numbers of outstanding request, the number of requests, and the number of PEs. The former gives unlimited bounding and the result is equivalent to static analysis of the maximum number of requests on a single PE.

As a conclusion, with all these trials, no other hash patterns better than "Xor" was found.

Table 3: Memory utilization for various hash patterns

Hash pattern name	Hash pattern description	Average Utilization (%)	
		Unlimited outstanding requests	Outstanding requests < PE#
No hashing		6.0	6.0
Low	bits10..3	25.0	23.6
High	bits15..8	77.0	49.0
Xor	bits10..3 XOR bits 15..8	91.9	74.6
fn2	bits10..3 XOR bits 15..8 XOR 85 hex	91.9	74.6
fn3	bits9..2 XOR bits 15..8 XOR bits12..5	88.4	73.6
fn4	bits9..2 XOR bits 15..8 XOR bits 10,11,12,9,7,8,5,6	88.5	70.0
fn5	bits9..2 XOR bits 15..8	91.3	69.1
fn6	bits9..2 XOR bits 8,9,10,11,12,13,14,15	66.0	40.3
fn7	bits9..2 XOR bits 4,5,6,7,8,9,10,11	58.1	45.9
fn8	bits9..2 XOR bits 3,4,5,6,7,8,9,10	39.8	34.2
fn9	bits9..2 XOR bits 8,9,10,3,4,5,6,7	20.0	18.8
fn10	bits9..2 XOR bits 14,13,12,11,10,5,4,3	39.8	36.9
fn11	bits16,14,12,10,8,6,4,2 XOR bits17,15,13,11,9,7,5,3	45.8	38.8
fn12	bits6,2 XOR bits7,3 XOR bits 8,4 XOR bits9,5	11.9	11.7
fn13	bits14,12,10,6,4,2 XOR bits15,13,11,7,5,3 XOR bits8,6,4 XOR bits9,7,5	51.7	41.7
fn14	bits5,4,3,2 XOR bits 12,9,6,3 XOR bits13,10,7,4 XOR bits14,11,8,5	37.2	31.8
fn15	bits9..2 XOR bits24,21,18,15,12,9,6,3 XOR bits25,22,19,16,13,10,7,4 XOR XOR bits26,23,20,17,14,11,8,5	73.6	49.5
fn16	bits10..3 XOR bits29,23,17,11,5 XOR bits31,25,19,13,7 XOR bits27,21,15,9	64.8	50.5
fn17	bits9..2 XOR bits23,22,17,16,11,10,5,4 XOR bits25,24,19,18,13,12,7,6 XOR bits27,26,21,20,15,14,9,8	76.9	57.7
fn18	bits9..2 XOR bits26,24,22,20,18,16,10,4 XOR bits12,6 XOR bits14,8	65.4	39.2
fn19	bits10..3 XOR bits23,22,17,16,11,10,5,4 XOR bits25,24,19,18,13,12,7,6 XOR bits27,16,21,20,15,14,9,8	86.3	67.9

Table 4: Memory utilization for various hash patterns (Continued)

Hash pattern name	Hash pattern description	Average Utilization (%)	
		Unlimited outstanding requests	Outstanding requests < PE#
fn20	bits9..2 XOR bits23,22,17,16,11,10,5,4 XOR bits25,24,19,18,13,12,6,7 XOR bits27,26,21,20,15,14,9,8	82.4	60.4
fn21	bits10..3 XOR bits21,20,17,16,11,10,5,4 XOR bits23,22,19,18,13,12,7,6 XOR bits25,24,23,22,15,14,9,8	88.5	67.9
fn22	bits10..3 XOR bits21,20,17,16,11,10,5,5 XOR bits23,22,19,18,13,12,6,7 XOR bits25,24,23,22,15,14,9,8	87.0	63.8
fn23	bits10..3 XOR bits21,20,17,16,11,10,5,4 XOR bits23,22,19,18,13,12,11,6 XOR bits25,24,23,22,15,14,14,9	88.9	65.6

4 Hashing and Interleaving For Any Numbers of Processors

An effective method to distribute segments on multiple PEs was proposed in Chapter 3. However the interleaving and hashing work only for systems with 2^n PEs. In real life, some of the PEs may be out of order (in a system with 64 thousand PEs, the probability that some of the PEs are faulty is very high) or some PEs connected to IO(IOPE) may be exclusively used for IO service. In such cases, it would be convenient if segments could be distributed on $2^n - k$ PEs, where k is the number of unavailable PEs, which is small compared to the total number of PEs, 2^n.

4.1 Interleaving Using a Modulo Table

Straight forward interleaving is done by assigning PEs in turn for each cache line as we did in Section 3.1. Although this was efficient for 2^n PEs, it is quite inefficient for power of $2^n - k$ PEs. Interleaving is a division of address by number of PEs. The remainder becomes the PE number, and the result of division becomes the local virtual address. The division by 2^n is trivial, but division by $2^n - k$ is quite complicated. Instead of doing full division, Figure 23 shows a slightly different implementation of this idea. In the figure, one PE out of 8PEs is unavailable. The segment has to be interleaved among 7PEs. First, the segment is divided into blocks. In the figure, the size of the block is 32 cache lines. It is convenient to make the size of blocks a power of 2 to keep the division for obtaining block number and offset in the block trivial. Each cache line in the block is assigned one of the available PEs in turn. Since $2^n - k$ cannot be a factor of the block size, there is unused space as a result of the transformation. Instead of having a large hardware mechanism to do the division, a compromise has been made to lose some of the memory.

This block transformation can be done by looking up a table called modulo table. The modulo table for the example is shown in Figure 24. Each entry of the table contains a new PE number and a new position. The transformation using the modulo table is performed as follows. First, the old address, which is the sum of LVA base and offset, is divided by the block size to obtain the block number. The remainder becomes the offset in the block. Then the offset in the old block is used to index the modulo table to read the new PE number and the new position. To calculate the new LVA, block number has to be multiplied by the new block length and the new position in the block has to be added.

For example, let's follow the transformation of cache line 11 hex in Figure 23. Cache line 11 is in block 0, and offset is 11 hex. The entry of the table addressed by the offset 11 hex has PE number 3, position 2. PE number 3 means the cache line is located on PE 3. Local Virtual Address of the cache line is calculated by $5 \times 0 + 2 = 2$, where 5 is the new block length, 0 is the old block number, and 3 is the position read from the table.

This table is general and can be used for any number of unavailable PEs, but it has to be initialized first, according to the number of unavailable PEs. In the example, the table is initialized so that all available PEs are used in turn, but it can be more sophisticated. Just as hashing in Chapter 3, usage of PEs stored in the table can be permutated at initialization to increase utility under constant stride access. Strides which are a multiple of the block size

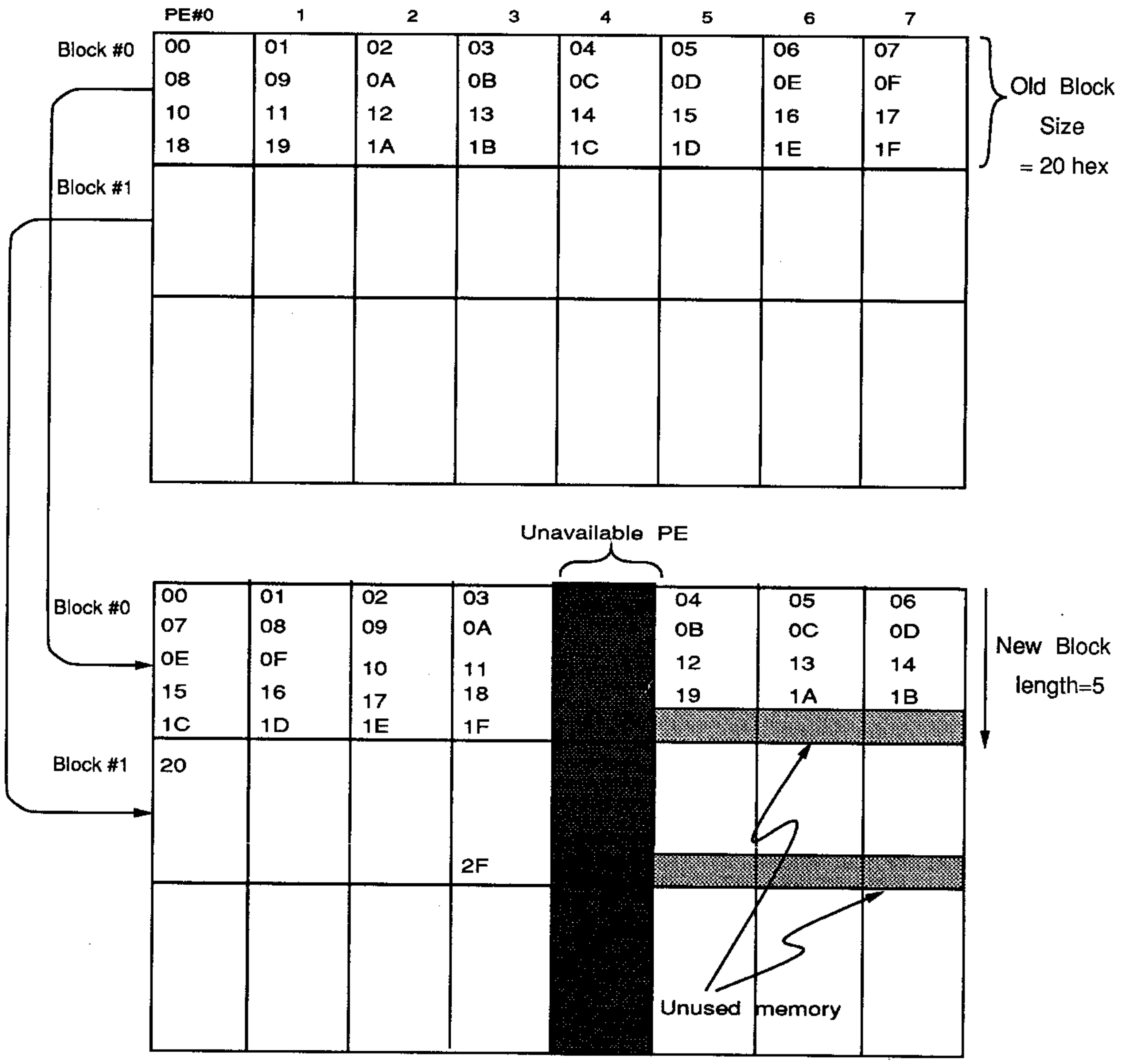


Figure 23: Hashing Using Modulo Table

	PE#	Position
00	0	0
01	0	1
02	0	2
03	0	3
04	0	5
05	0	6
10	2	2
11	3	2
1F	3	4

Figure 24: Modulo Table

Table 5: Memory loss due to one unavailable PE

(1k entry table)

Total Number of PEs	4	8	16	32	64	128	256
Memory Loss(%)	0.2	0.5	1.1	2.8	4.4	10.4	19.7
New Block Length	342	142	69	34	17	9	5

Table 6: Memory loss for different fractions of unavailable PEs

(Total 256PE, 1k entry table)

Fraction of Unavailable PEs	1/4	1/8	1/16	1/32	1/64	1/128	1/256
Memory Loss (%)	11.1	8.6	14.7	17.4	18.7	19.4	19.7
New Block Length	6	5	5	5	5	5	5

need some other hashing capability.

One difficulty of implementing this method is the multiplication of the new block length. The new block length for the system depends on the block size, the total PE number and number of unavailable PEs. Implementing a full multiplier requires too much hardware and the delay is large. A better way is to limit the multiplier to some subset of integers. By choosing an appropriate block size, the new block length can be controlled to be the sum of two power of 2 numbers. It is not so hard to built a multiplier in this case. It is only a simple shift and add operation.

Some estimation has been done on the loss of usable memory resulting from unused space in the block. Table 5 shows the loss for 1 unavailable PE in different total PE numbers. The table size is 1k entries for all PE numbers. It shows that the loss rate is high for large numbers of PEs. As table size adjustment was not made, 4, 8, and 16 PE have new block size which makes it hard to do the multiplication.

Table 6 shows the loss for a 256 PE system with different fractions of unavailable PEs. The table size is also 1k entries. With the same new block size, the loss rate is low for large number of unavailable PEs. When 1/4 of the PEs are unavailable, loss is higher than 1/8 case because the block size increases.

Table 7 is same as Table 6 with 64PE. Compared to 256PEs, loss rate is lower.

Table 8 is loss for 64PE system with 256 entry table. Loss rate is identical to Table 6. To obtain the same loss rate, larger system requires larger table.

The above estimation shows some problems with this method. First, it requires a large table for a large number of PEs. Second, it shows high loss for small numbers of unavailable PEs. The second doesn't fit the real situation, because the number of unavailable PEs in the system is rather small.

Table 7: Memory loss for different fractions of unavailable PEs

(Total 64PE, 1k entry table)

Fraction of Unavailable PEs	1/4	1/8	1/16	1/32	1/64
Memory Loss (%)	3.0	3.8	5.2	2.8	4.4
New Block Length	22	19	18	17	17

Table 8: Memory loss for different fractions of unavailable PEs

(Total 64PE, 256 entry table)

Fraction of Unavailable PEs	1/4	1/8	1/16	1/32	1/64
Memory Loss (%)	11.1	8.6	14.7	17.4	18.7
New Block Length	6	5	5	5	5

4.2 Re-interleaving into a Subset of Available PEs

In this section, another way of distributing data structure among $2^n - k$ PEs is introduced. In this method, we use a virtual PE number. By virtualizing the PE number, we can keep the lower half of PEs ($< 2^{n-1}$) always available. This is done by giving all the unavailable PEs large virtual PE number ($\geq 2^{n-1}$). We first interleave and hash as discussed in Chapter 3. Whenever the result of hashing comes on an unavailable PE, we re-interleave into the lower half of the PEs. In Figure 25, two unavailable PEs are each assigned virtual PE numbers of $m-2$ and $m-1$ ($m = 2^n$). When the result of interleaving and hashing among m PEs result on PE $m-2$ or $m-1$, it is re-interleaved into PE 0 to $m/2-1$ using special segment descriptor. This special segment descriptor shows a segment where the memory space of PE $m-2$ and $m-1$ is re-interleaved. This segment has to be reserved on PE0 to $m/2-1$. This way, the area marked unused in the figure cannot be used for interleave among all PEs, but it can still be used for interleave between $m/4$ or less PEs. By the use of a virtual PE number, the test of availability is easy and the result of re-interleave is assured to be available.

Figure 26 shows the hardware mechanism to support re-interleaving. In addition to the original hash and interleave mechanism, there are two selectors before interleaving and hashing, a matching mechanism to detect unavailable PE number, and an offset generator. The added hardware is dotted in the figure. During the first hashing, selectors select output from the segment TLB and the offset from global virtual address. Then the result PE number of interleave and hashing is tested for availability. Using a virtual PE number, this test can be simple matching of some bits or arithmetic compare, instead of full associative compare. The result of the test is used to control the selectors. If the PE is available, the result of the first interleaving and hashing is directly sent to the message formatter. If the PE is unavailable, re-interleaving is performed. This is done by sending the special segment descriptor to the interleave unit along with an offset from the offset generator. The offset generator generates the offset from the result PE number and LVA address of the first interleave and hashing. Finally, the result of the second interleave and hashing will be sent to the message formatter. With this hardware, only when the result PE number of the first interleaving and hashing was unavailable, it takes more cycles than in the normal case. First hashing plays an important role here, because if all the requests goes to unavailable PEs, all the requests will take additional cycles for re-interleaving. This will slow down the system. It is important that this situation is avoided by the hashing.

In both cases, whether the result of the first interleaving and hashing has to be re-interleaved or not, the result virtual PE number has to be translated into a real PE number. For example, in Figure 27, real PE2, 5 are unavailable. In order to keep virtual PE0 to 3 available, real PE 2, 5, 6, 7 are given virtual PE numbers of 6, 7, 2, 5. The rest of the PEs have the same virtual PE number as their real PE number. PE5 does not have to be virtualized because it is in the last half of the PE, but the virtualizing makes the availability test simple. Translation from virtual to real PE number can be done before message formatting with fully associative memory. The number of entries in this memory limits the number of unavailable PEs, which is supposed to be small, like 5% of the total PE number.

Sometimes, it is convenient to have some of the segments interleaved on IOPEs, while others are not. This can be implemented by having two classes of unavailable PEs, fault and

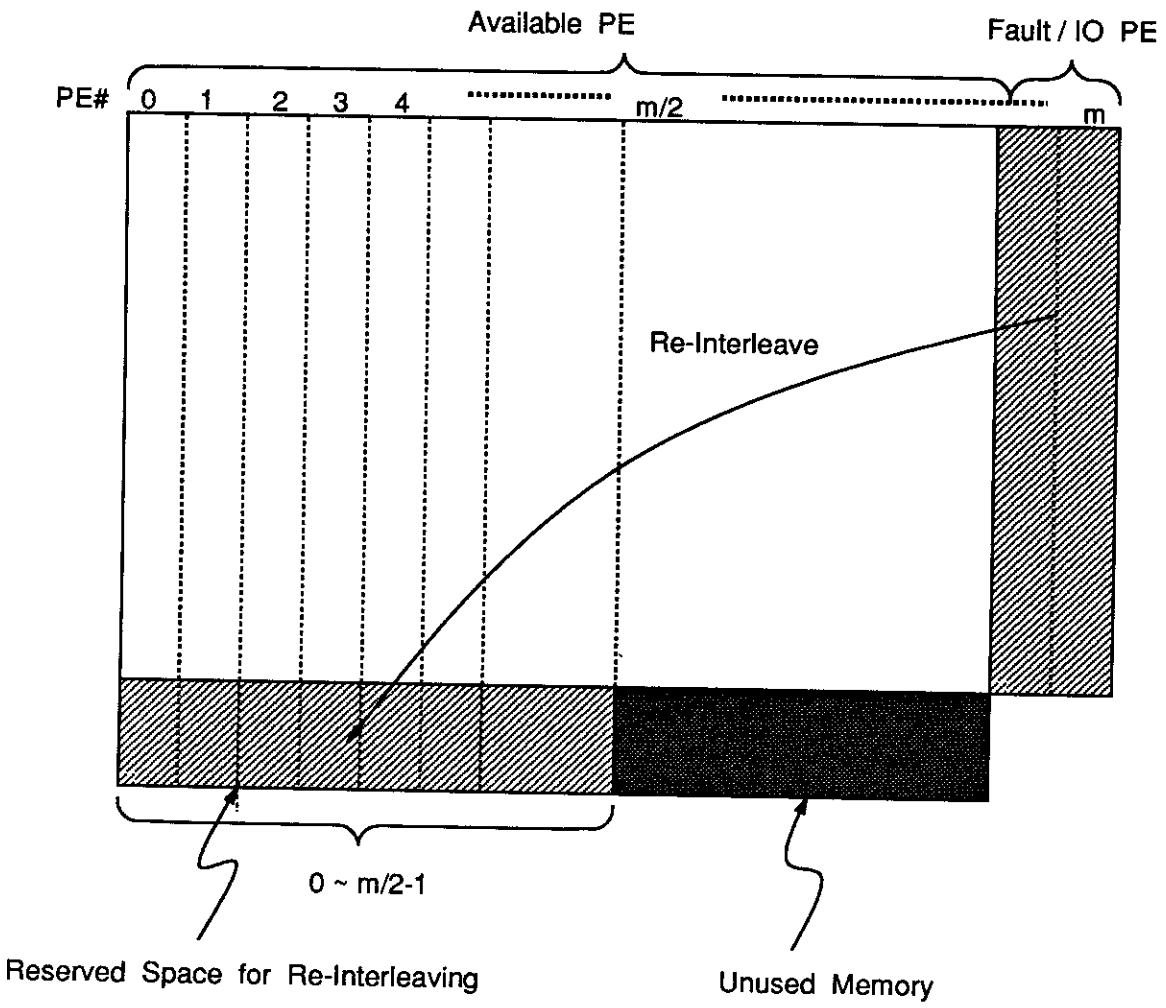


Figure 25: Re-interleave into Power of 2

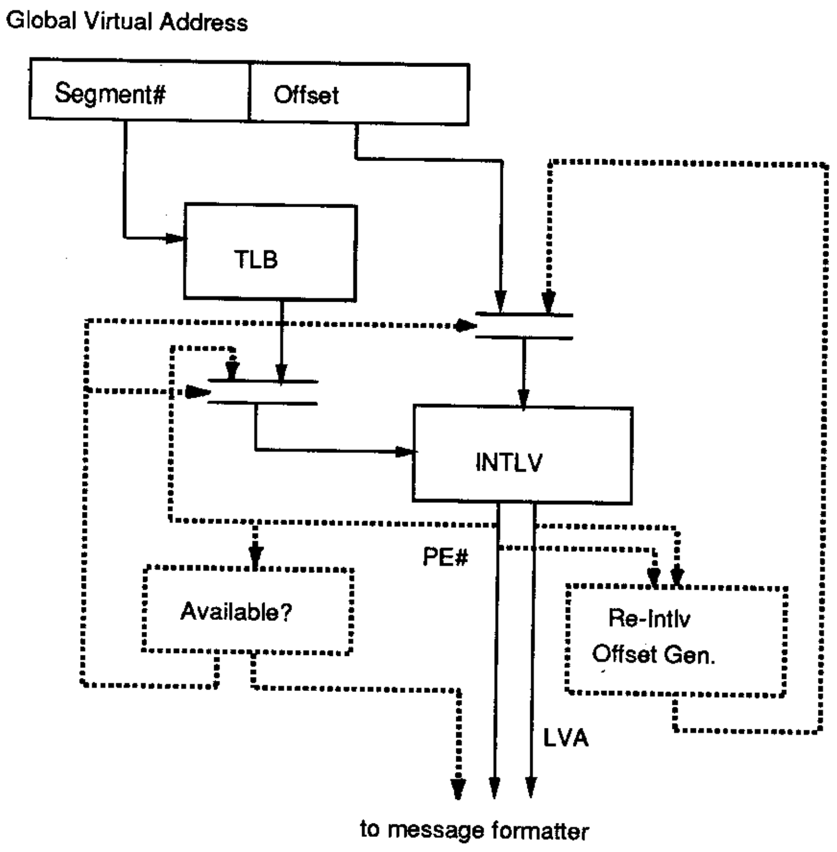


Figure 26: Hardware for Re-Interleaving

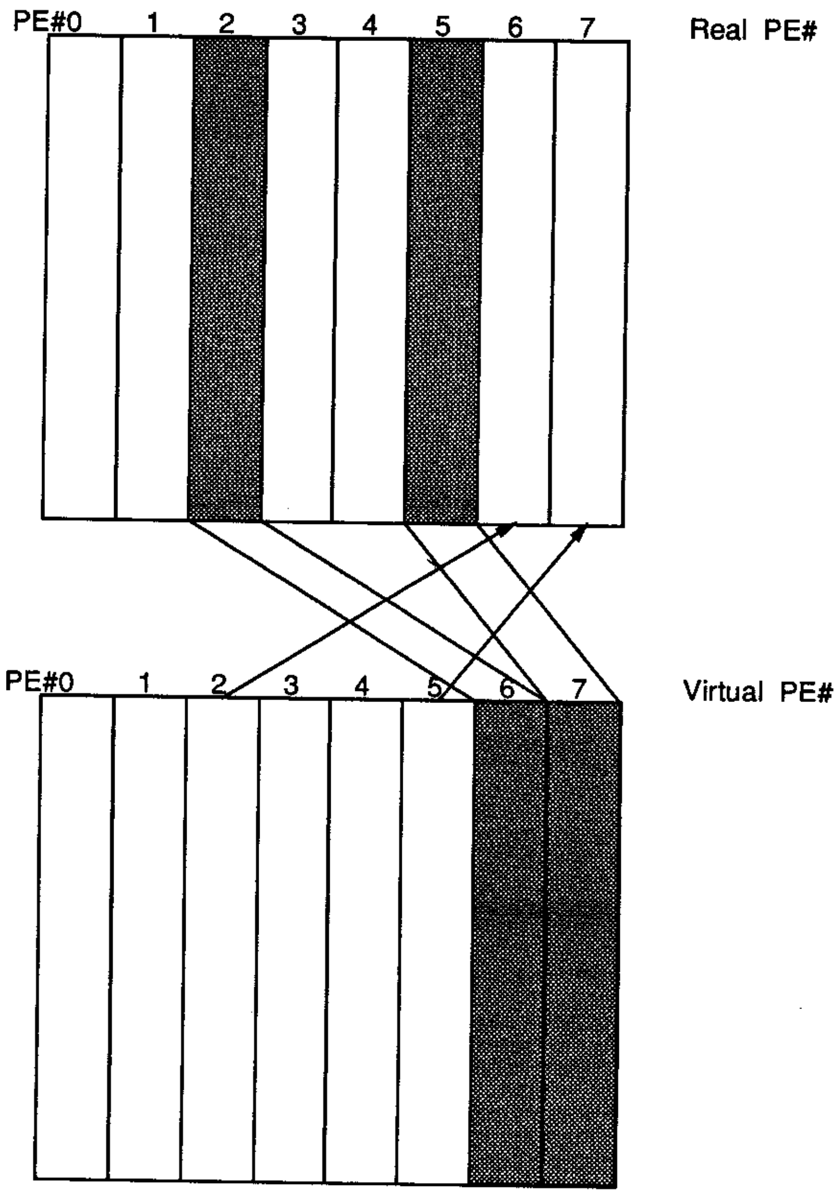


Figure 27: Virtual and Real PE Number

Table 9: Memory loss for re-interleaving

Fraction of Unavailable PEs	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256
Memory Loss (%)	0.0	11.1	8.6	5.2	2.8	1.6	0.8	0.4

IOPE. Instead of having an available/unavailable test, we can have an available/fault/IOPE test. When the result PE number of the first interleaving and hashing was IOPE, only the segments with IO bit off are re-interleaved. If the result PE number of the first interleaving and hashing was fault, the segment is re-interleaved regardless of the IO bit.

Table 9 shows the estimation of loss for re-interleaving. This loss rate is true for any total number of PEs. The result shows that loss is small for small numbers of unavailable PEs.

5 Conclusion

In this paper an effective way to distribute global structures onto multiple PEs was discussed. First interleaving and hashing mechanism for 2^n PEs was introduced. Then simulation was done to decide a fitting hash pattern. With that hash pattern, the over all utilization of the memory , under regular access pattern with constant strides, is better compared to that of interleaving without hashing. Two mechanisms to distribute global structures on $2^n - k$ PEs were then introduced. The mechanism using re-interleaving has less loss of memory compared to the other using modulo table.

6 Acknowledgments

I would like to thank Prof. Gregory M. Papadopoulos for giving me chance to work on the Start project and for his close advice on this work. I would like to also thank Prof. Arvind for accepting me as a visiting scientist in his group . Dana Henry and Christopher Joerg gave helpful comments and suggestions on this work. The grammatical quality of this paper improved by comments by my office mate Stephen Glim.