Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2012, Article ID 365091, 12 pages
doi:10.1155/2012/365091

Research Article

Memory Map: A Multiprocessor Cache Simulator

Shaily Mittal[1] and Nitin[2]

[1] *Department of Computer Science & Engineering, Chitkara University, Baddi, Solan 174103, India*
[2] *Department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology, Waknaghat, Solan 173234, India*

Correspondence should be addressed to Nitin, delnitin@ieee.org

Received 7 February 2012; Accepted 31 May 2012

Academic Editor: Vivek Kumar Sehgal

Nowadays, Multiprocessor System-on-Chip (MPSoC) architectures are mainly focused on by manufacturers to provide increased concurrency, instead of increased clock speed, for embedded systems. However, managing concurrency is a tough task. Hence, one major issue is to synchronize concurrent accesses to shared memory. An important characteristic of any system design process is memory configuration and data flow management. Although, it is very important to select a correct memory configuration, it might be equally imperative to choreograph the data flow between various levels of memory in an optimal manner. Memory map is a multiprocessor simulator to choreograph data flow in individual caches of multiple processors and shared memory systems. This simulator allows user to specify cache reconfigurations and number of processors within the application program and evaluates cache miss and hit rate for each configuration phase taking into account reconfiguration costs. The code is open source and in java.

1. Introduction

In the memory hierarchy, cache is the first encountered memory when an address leaves the central processing unit (CPU) [1]. It is expensive, relatively small as compared to the memories on other levels of the hierarchy and provides provisional storage that supplies most of the information requests of the CPU, due to some customized strategies that control its operation.

On-chip cache sizes are on the rise with each generation of microprocessors to bridge the ever-widening memory-processor performance gap. According to a literature survey in [2], caches consume 25% to 50% of total chip energy, while covering only 15% to 40% of total chip area, whereas designers have conventionally focused their design efforts on improving cache performance as these statistics and technology trends visibly indicate that there is much to be gained from making energy and area, as well as performance, front-end design issues.

Embedded systems as they occur in application domains such as automotive, aeronautics, and industrial automation often have to satisfy hard real-time constraints [3]. Hardware architectures used in embedded systems now feature caches, deep pipelines, and all kinds of conjecture to improve average case performance. The speed and size are two concerns of embedded systems in the area of memory architecture design. In these systems, it is necessary to reduce the size of memory to obtain better performance. The speed of memory plays an important role in system performance. Cache hits usually take one or two processor cycles, while cache misses take tens of cycles as a penalty of miss handling, so the speed of memory hierarchy is a key factor in the system. Almost all embedded processors have in-chip instructions and data caches. Scratch-pad memory (SPM) has become an alternative for the design of modern embedded system processors [4, 5].

Multiple processors on a chip communicate through shared caches embedded on a chip [6]. Integrated platforms for embedded applications [7] are even more assertively pushing core-level parallelism. SoCs with tens of cores are commonplace [8–11] and platforms with hundreds of cores have been proclaimed [12]. In principle, multicore architectures have the advantages of increased power-performance scalability and faster design cycle time by exploiting replication of predesigned components. However, performance and power benefits can be obtained only if applications exploit a high level of concurrency. Indeed, one of the toughest challenges to be addressed by multicore architects is how to help programmers expose application parallelism.

Thread level parallelism brings revolution in MPSoC [13]. As multiple threads can be executed simultaneously, it makes the real advantage of multiple processors on a single chip [14]. However, this leads to a problem of concurrent access to cache by multiple processors. When more than one processor simultaneously wants to access the same shared cache then there is a need of synchronization mechanism [15]. This paper presents memory map, a fast, flexible, open source, and robust framework for optimizing and characterizing the performance, hit and miss ratio of low-power caches in the early stages of design. In order to understand the description of simulator and related work that follows, one must be aware of the terminology used to describe caches and cache events.

Caches can be classified into three possible ways depending on the type of information stored. An instruction cache stores CPU instructions, a data cache stores data for the running application, and a unified cache stores both instructions and data. The basic operations to a cache are reads and writes. If the location specified by the address and generated by CPU is stored in the cache, a hit occurs, otherwise, a miss and the request is promoted to the next memory in the hierarchy. A block is the smallest unit of information present in the cache. Based on possible locations for a new block, three categories of cache organization are possible. If the number of possible locations for each block is one, the cache is said to be direct mapped. If a block can be placed anywhere in the cache, the cache is said to be fully associative and if a block can be placed only in one of a restricted set of n places, the cache is said to be n-way set associative. When a miss occurs, the cache must select a block to be replaced with the data fetched from the next-level memory. In a direct-mapped cache, the block that was checked for a hit is replaced. In a set associative or fully associative cache, any of the blocks in the set may be replaced.

Associativity is one of the factors that impinge on the cache performance. Currently, modern processors include multilevel caches with increased associativity. Therefore, it is critical to revisit the effectiveness of common cache replacement policies. When all the lines in a cache memory set become full and a new block of memory needs to be replaced into the cache memory, the cache controller must replace it with one of the old blocks in the cache. We have used the same procedure for SPM. The modern processors employ various policies such as LRU (Least Recently Used), Random, FIFO (First in First Out), PLRU (Pseudo LRU), and N-HMRU.

Least recently used [16] cache replacement policy rejects the least recently used items first. This algorithm keeps track of what was used when and which is expensive to make sure the algorithm always discards the least recently used item. Random cache replacement policy randomly selects a candidate item and discards it to make space when required. This algorithm does not keep any information about the access history. FIFO cache replacement policy is the simplest page replacement algorithm. This algorithm requires slight book keeping on the part of the operating system. The operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the first arrival in front. When a page needs to be replaced, the page at the front of the queue, that is, the oldest page is selected. Although, FIFO is cheap and intuitive, it performs poorly in practical application. Hence, it is rarely used in its unmodified form. PLRU [17] maintains a tree of cache ways instead of linear order as in case of LRU. Every inner tree node has a bit pointing to the subtree that contains the leaf to be replaced next when required. The H-NMRU [18] cache replacement policy can be described using a multiway tree, where the leaf nodes represent the lines in the set. Each intermediate node stores the value of its most recently used (MRU) child. During a cache hit, the tree is traversed to reach the accessed line at the leaf node. On the way, the value of the nodes is updated to point to the path of traversal. In this way, the most recently used branches are stored at each node of the tree. While on a cache miss, the tree is traversed selecting a random value unlike from the MRU value stored in the node. From each level a non-MRU path is selected. Hence, this algorithm points to a leaf node which has not been accessed in recent times.

A write through a cache modifies its own copy of the data and the copy stored in main memory at the time of the write. In a copy-back cache, it modifies its own copy of the stored information at the time of the write, but it updates the copy in main memory only when the modified block is selected for eviction. Read misses usually result in fetching the requested information into the cache; while write misses do not necessarily require that the cache fetch the modified block. The new block is loaded on a write miss if the cache is using the write allocate strategy, otherwise the write request is simply forwarded and the modified data is not loaded into the cache. The cache is said to be nonallocating in the above case.

The rest of the paper is organized as follows. Section 2 describes working, benefits, and drawbacks of various currently available memory processor simulators in the field of embedded systems. An approach with experimental results for multiprocessor synchronization is described in Section 3 followed by an overview of proposed memory map multiprocessor simulator architecture in Section 4. Section 5 describes our simulation environment, and experimental results are explained in Section 6. Lastly, our work is concluded in Section 7.

2. Survey and Motivation

A number of simulators are available for multiprocessor shared memory architecture evaluation. We are discussing some of them with their features and problems that lead to the need of memory map multiprocessor simulator. SMP-Cache [19] is a trace-driven simulator for SMP (symmetric multiprocessor) memory consisting of one windows executable file, associated help files, and collection of memory traces. SMPCache is used for the analysis and teaching of cache memory systems on symmetric multiprocessors. It has a full graphic and friendly interface, and it operates on PC systems with Windows 98 or higher. However, SMPCache is a trace-driven simulator; however, we need a certain tool to generate memory traces.

OpenMP [20] is a de facto standard interface of the shared address space parallel programming model using OpenMP directives. For C and C++ programs, programs/directives are provided by the OpenMP API to control parallelism through threads. OpenMP supports parallel programming using compiler directives, however, lacks tool to gather memory access statistics.

SimpleScalar [21] is C-based simulation tool that models a virtual computer system with CPU, cache, and memory hierarchy. SimpleScalar [22] is a set of tools through which users can build modeling applications that simulate real programs running on a range of modern processors and systems. The tool set embraces sample simulators ranging from a fast functional simulator to a dynamically scheduled processor model that supports nonblocking caches, speculative execution, and state-of-the-art branch prediction. In addition to simulators, the SimpleScalar tool set takes account of statistical analysis resources, performance visualization tools, debugging, and verification infrastructure. However, the problem is that SimpleScalar does not support multiprocessors.

M-Sim [23] is a multithreaded simulation environment for concurrent execution based on SMT model. M-Sim extends the SimpleScalar 3.0 d toolset. M-SIM supports single-threaded execution, SMT (simultaneous execution of multiple threads), and number of concurrent threads MAX_CONTEXTS. For executing the program, we need to write a statement:

./sim-outorder-num_cores 3-max_contexts_per_core 3

Cache:dl1 dl1:1024:8:2: l-cache:dl2 dl2:1024:32:2:l hello.arg

An argument file contains alpha binary executable will be produced

1000000000 # ammp < ammp.in > ammp.out

M-Sim supports multiprocessors but requires separate program per core. M-Sim requires alpha binaries executables using DEC compiler, which is not a freely available compiler.

Class library in SystemC, including the source code, is free and available to the public through SystemC portal [24, 25]. In addition to standard Linux C++ development and shell tools, GTKWave waveform viewer and Data Display Debugger (DDD) were used. However, the major shortcoming for software development of this tool is that standard software development tools are debugging the software of the model and not the software running on the model. Moreover, there is no linker available for SystemC. Hence, the semantics of SystemC build on top of C++ syntax is not checked within the compilation process that in turn results in illegal semantics that are syntactically correct and will not produce any compiler errors or warnings. In these circumstances, the programs will cause a run-time error, which are typically harder to locate than compile-time errors. In addition, unfathomable error messages are produced by standard C++ compiler with the illegal use of SystemC semantics and generate a syntactical error within the SystemC library. Interaction with other software environments and native C/C++ and SystemC can also be niggling.

3. Alternate Approach for Multiprocessor Synchronization

3.1. Memory Interleaving. Memory interleaving [26] is a technique for compensating the relatively slow speed of DRAM. Alternative sections can be accessed immediately by CPU without waiting for memory to be cached. Multiple memory banks take turns to supply data. An interleaved memory is said to be n-way interleaved, if there are n banks and memory location i would reside in bank number i mod n. One way of mapping virtual addresses to memory modules is to divide the memory space into contiguous blocks. The CPU can access alternate sections immediately, without waiting for memory to catch up (through wait states). Interleaved memory is one technique to compensate relatively slow speed dynamic RAM (DRAM). Interleaved memories are the implementation of the concept of accessing more words in a single memory access cycle. This can be achieved by partitioning the memory into N separate memory modules. Thus, N accesses can be carried out to the memory simultaneously.

We have implemented memory interleaving with respect to merge sort algorithm to avoid any synchronization issue in n process scenario. In general, the CPU is more likely to access the memory for a set of consecutive words (either a segment of consecutive instructions in a program or the components of a data structure such as an array, the interleaved (low-order) arrangement shown in Figure 1 is preferable as consecutive words are in different modules and can be fetched simultaneously.

Instead of splitting the list into 2 equal parts, the list is accessed by n processors simultaneously using memory interleaving, thus ensuring that they never access the same memory locations as described through Figure 2. While merging, all the memory points being merged at a time will be at contiguous locations. Due to this, all the locations pointed by different processors are brought into cache simultaneously; merge module can access all the elements in cache, hence increasing cache hit and performance of sorting algorithm. Merge sort algorithm has been modified accordingly as shown in Algorithm 1. According to the modified algorithm, merging operation will become highly efficient as values to be merged will be at contiguous location and will be brought to cache simultaneously.

In order to increase the speed of memory reading and writing operation, the main memory of $2^n = N$ words can be organized as a set of $2^m = N$ independent memory modules where each containing 2^{n-m} words. If these M modules can work in parallel or in a pipeline fashion, then ideally an M fold speed improvement can be expected. The n-bit address

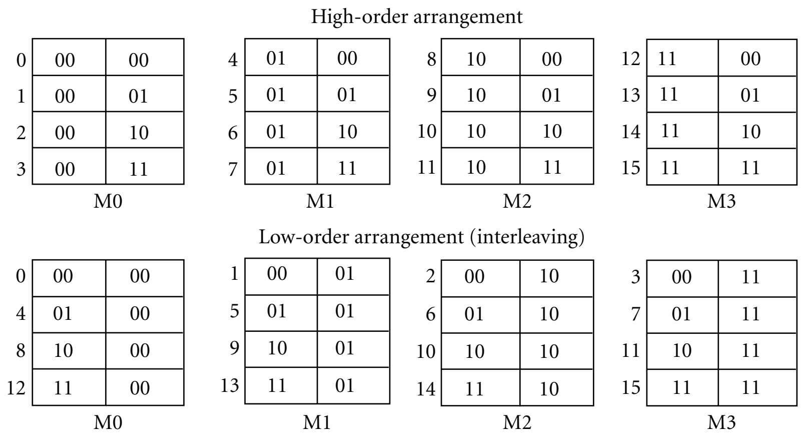
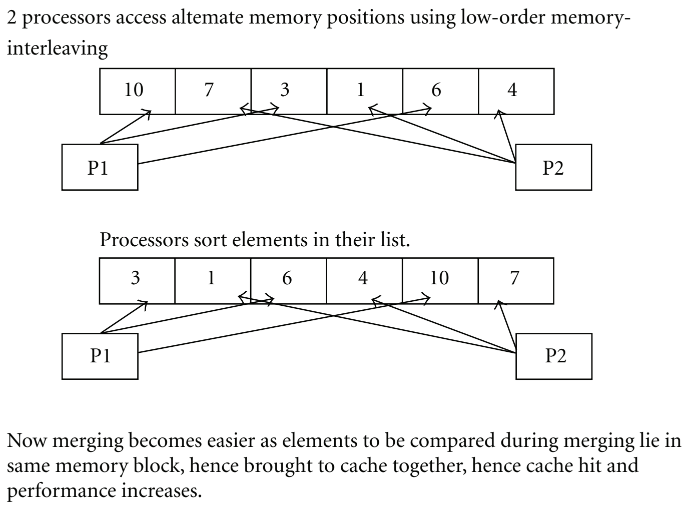


Figure 1: Interleaved structure.


2 processors access altemate memory positions using low-order memory-interleaving

Processors sort elements in their list.

Now merging becomes easier as elements to be compared during merging lie in same memory block, hence brought to cache together, hence cache hit and performance increases.

Figure 2: Multiple processors accessing memory simultaneously.

is divided into an m-bit field to specify the module, and another $(n-m$-) bit field to specify the word in the addressed module.

(1) Interleaving allows a system to use multiple memory modules as one.

(2) Interleaving can only take place between identical memory modules.

(3) Theoretically, system performance is enhanced because read and write activity occurs nearly simultaneously across the multiple modules.

In our experiment, we have taken the size of data array as of 30 elements and LRU as the data replacement policy. As far as cache configuration is concerned, we have taken SPM as a 16 bit 2-way set-associative cache and an L2 cache of 64 bit 2-way set-associative cache.

3.2. Observations. We have used SimpleScalar functional simulators sim-cache and sim-fast to implement the above modified merge sort algorithm. We use a system running the Linux operating system. We evaluated and compared cache hit ratio and cache miss rate. The percentage of data accesses that result in cache hits is known as the hit ratio of the cache. Figure 3(a) shows the hit ratio for SPM in case of memory interleaving and comparing it with the normal execution of the merge sort benchmark. As it is clear from the graph hit ratio is increased from 98.55 (normal sorting) to 99.87 (sorting with memory interleaving). This is a considerably good achievement as hit ratio is very close to 100%. L2 cache hit ratio is shown in Figure 3(b). It has also been proved that cache hit ratio also increased from 96.54 in normal sorting to 99.74 in case of memory interleaving for modified merge sort algorithm. Moreover, obtaining 100% hit ratio is practically impossible.

Miss Rate is measured as 1-hit ratio. The first two bars in Figure 4(a) show the SPM miss rate with normal and interleaved execution. As shown, miss rate decreases from 1.46 for normal sorting to 0.12 for sorting with memory interleaving. Similarly, in case of L2 cache miss rate also decreases from 1.46 to 0.26. This is a considerably

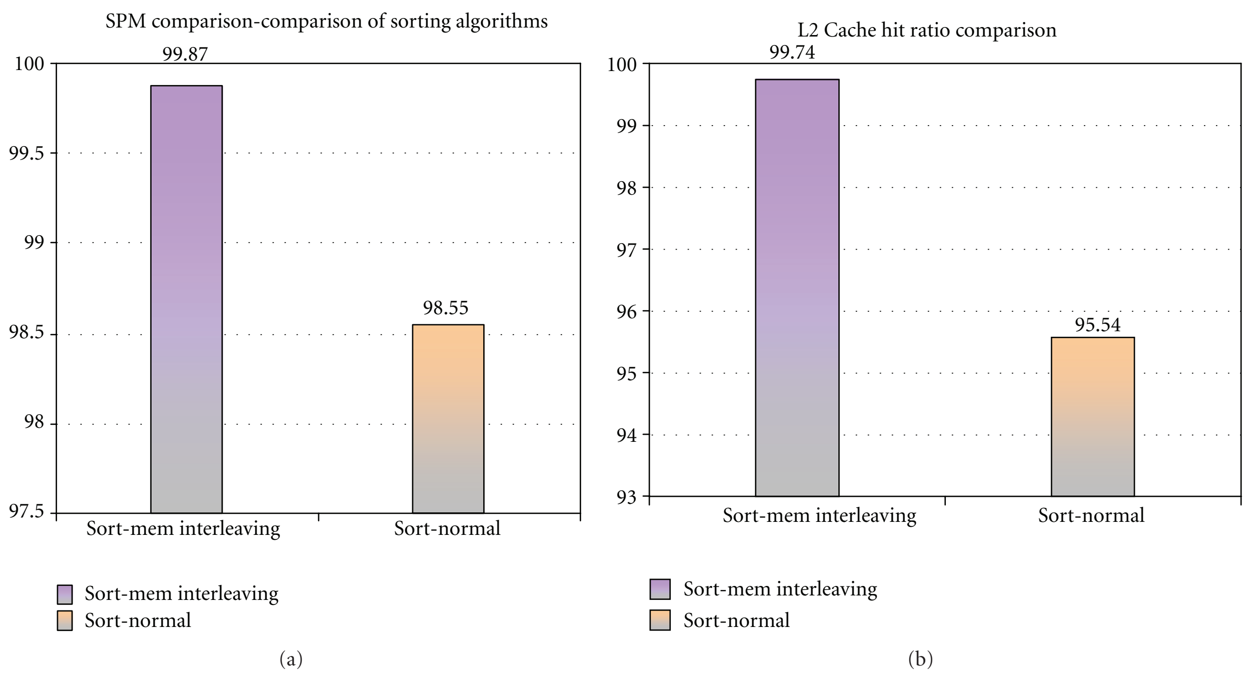


Figure 3: (a) SPM and (b) L2 cache hit ratio.

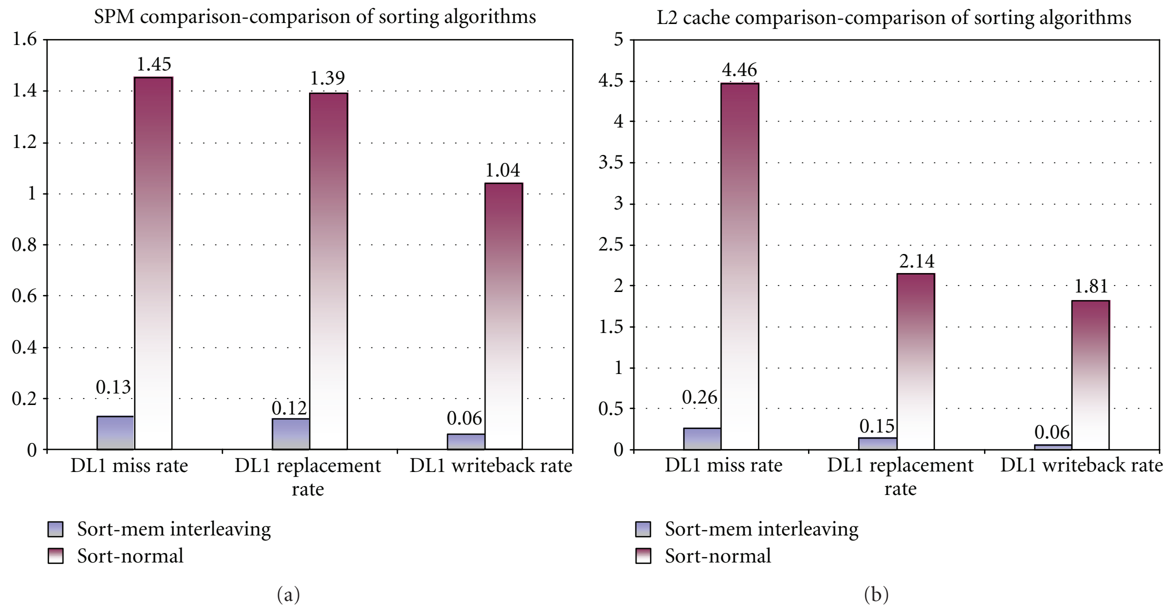


Figure 4: (a) SPM and (b) L2 cache miss rate, replacement rate and, writeback rate.

good achievement as obtaining nil miss rate is practically impossible. We have deemed over DL1 cache. The next two bars in the graph are showing the comparison of replacement rate for normal execution to the execution with interleaved memory. It also shows a significant decrease from 1.39 to 0.12 for SPM and from 2.14 to 0.15 for L2 cache. Replacement rate indicates the rate of replacement due to cache aliasing and cache miss. Lesser replacement rate is due to high cache hit, which lowers down effective memory access time. Likewise, the last two bars, are explaining the writeback rate for SPM as well as for L2 cache. As shown in the bars, it has also been decreased from 1.04 to 0.06 for SPM and from 1.81 to 0.06 for L2 cache. Low writeback rate indicates the rate of replacement due to cache aliasing and cache miss is low.

3.3. Limitations. Although memory interleaving has decreased the miss rate and shows significant improvement, this approach has some limitations. The current study

```
Sorting algorithm (modified merge sort)
#define MaxSize 7000
Algorithm ($A$ [][], left, right, $M$, inter)

//$A$ is array of MaxSize elements, which need to be sorted from left to right position, $M$ // is the number of processors, which
will sort elements of array in parallel, inter is degree // of interleaving.

factor = (right − left + 1)/Pow ($M$, inter);
IF (factor > 3) THEN

        //This loop partitions the elements into $M$ processors

        FOR ($i = 0$ to $M$ DO)
                                Sort Individual elements in the partitioned array with
                                starting position as $I$ and end position till factor or
                                factor + 1, each element placed with "inter" positions
                                next to previous elements.
        Merge ($A$, left, right, $M$, inter);
ELSE
          InsertionSort ($A$, left, $n$, right, Pow ($M$, inter − 1));
END
Algorithm of insertion sort is modified so that each element is placed with "inter" positions next to previous element
int $j$, $p$, Tmp, count;
FOR ($P = i$ + inter, count = 1; count < $N$, $P$ <= right; $P = P$ + inter, ++count)
      Tmp = $A[P]$;
FOR ($j = P$; $j$ >= $I$ && $j$ − inter >= $i$ && $A[j$ − inter$]$>Tmp; $j = j$ − inter)
      $A[j] = A[j$ − inter$]$;
$A[j]$ = Temp;
Modified Merge Algorithm
Algorithm Merge ($A$, left, right, $M$, inter)
{
 pInter = Pow($M$, inter);
 FOR ($i = 0$; $i < M$; $i$++) pointer [$i$] = left + $r*$pInter/$M$;
      $N$ = (right − left + 1);
 FOR $i = 0$ to $n$ do
       FOR ($j = 0$; $j < M$; + + $j$)
             IF((pointer[$j$] ! = −1) && (TmpArray[$i$]> $A$[pointer[$j$]))
                   TmpArray[$i$] = A[pointer[$j$];
                   indexJ = $j$;
             IF(pointer[index $J$] + pInter > $n$ − 1)
                   pointer[index $J$] = −1;
             ELSE pointer[index $J$] + = pInter;
 END For($i$)
      FOR($i = 0$; $i < n$; + + $i$)
         $A[i]$ = TmpArray[$i$];
}
```

Algorithm 1: Modified algorithms.

does not involve studying algorithmic complexity of new proposed algorithms, as their complexity is high. It is solely based on assumption that we have increased CPU power with high computation rate; only reduced parameter is memory access time. Moreover, a current study is based on SimpleScalar simulator, which can be implemented on SimpleScalar architecture. In addition, as this simulator simulates system calls to capture memory image, there are some system calls that are not allowed by simulator, which leads to inhibit the use of certain advanced system features. Furthermore, we cannot implement multiple processors on a single benchmark as required by our problem statement.

4. Architecture of Memory Map Simulator

The target architecture is a shared memory architecture for a multiprocessor system [27]. The platform consists of

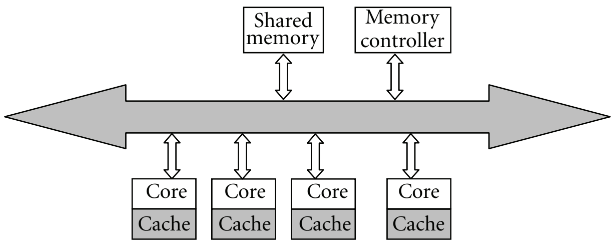


Figure 5: Proposed simulator architecture.

computation cores and private cache (one for each processor) and of a shared memory for interprocessor communication. The multiprocessor platform is homogeneous and consists of data caches. Figure 5 shows the architectural template of the multiprocessor simulation platform to be used in this study. It consists of

(1) a configurable number of processors,

(2) their private cache memories,

(3) a shared memory,

(4) memory controller.

We have implemented different features in this multiprocessor memory simulator. These features are discussed as follows.

We have used physical cache addressing model, that is, physical address is used to map cache address. Direct addressing scheme is used to map cache block to memory block. Different processors use shared memory to interact among each other. It allows concurrent execution of single program. It maintains the exclusive access to parts of memory by different processors to avoid cache coherence and uses writeback strategy to maintain cache and memory in synchronization [28].

Multiple processors or threads can concurrently execute many algorithms without requiring access to memory at simultaneous parts. However, it allows processes to interact using shared memory. For example, merge sort requires processors to sort the data independently, however later these processors need to merge the data by mutual interaction. Moreover, meta data related to variable is mapped with logical address; it facilitates easy access to cache, memory [29]. It consists of the following attributes:

(1) variable name,

(2) variable type,

(3) variable size,

(4) variable index,

(5) invalid flag,

(6) dirty flag,

(7) processor flag.

We have also implemented some general operations on memory map multiprocessor simulator, which are described below.

(1) Memory allocation: memory is represented as an array of byte stream. It maintains the free memory pool and used block list. While allocating space for a variable, it looks for first free pool blocks that has equaled or of more size. It uses Java Reflection API for calculation of size of various primitive types and classes. It converts variables to byte and type depending upon the type of variable. Variable is converted to stream of byte array and vice versa. Moreover, an invalid bit flag has also been set and reset. Before any value is set into memory, it is marked as invalid.

(2) Memory query: for any query, it first maps physical address to logical address. Then processor cache is checked whether block containing physical address is present. If yes, it is a hit, and data is read from cache. If no, it is a miss, then it finds out whether cache block occupies other memory block or not. If yes, block needs to be set into memory. Hence, it fetches the data from memory.

(3) Memory update: in memory update operation, firstly physical address will be mapped to logical address. Processor cache is checked for the presence of block containing physical address. If block is present, it is a hit, and hence after updating cache, dirty flag is set to 1. If block is absent, it is a miss. Therefore, it finds out whether cache block occupies other memory block with dirty flag set, and if yes, block needs to be set into memory. Dirty flag of replaced block is zero. It updates the data in memory and brings the block to cache.

Flowchart shown in Figure 6 describes the whole operation in detail. It explains the working flow of memory map multiprocessor simulator.

5. Testbed and Experimental Setup

In the implemented architecture on our memory map simulator, we have taken private cache [30] associated with each processor and no shared cache with one shared main memory as shown in Figure 7. The configuration of private cache is direct mapped L2 cache of block sizes 32, 64, and 128 bits. We have implemented 2-way set-associative LRU and FIFO replacement policies. We have taken three benchmarks to run on this architecture, namely, merge sort, bubble sort

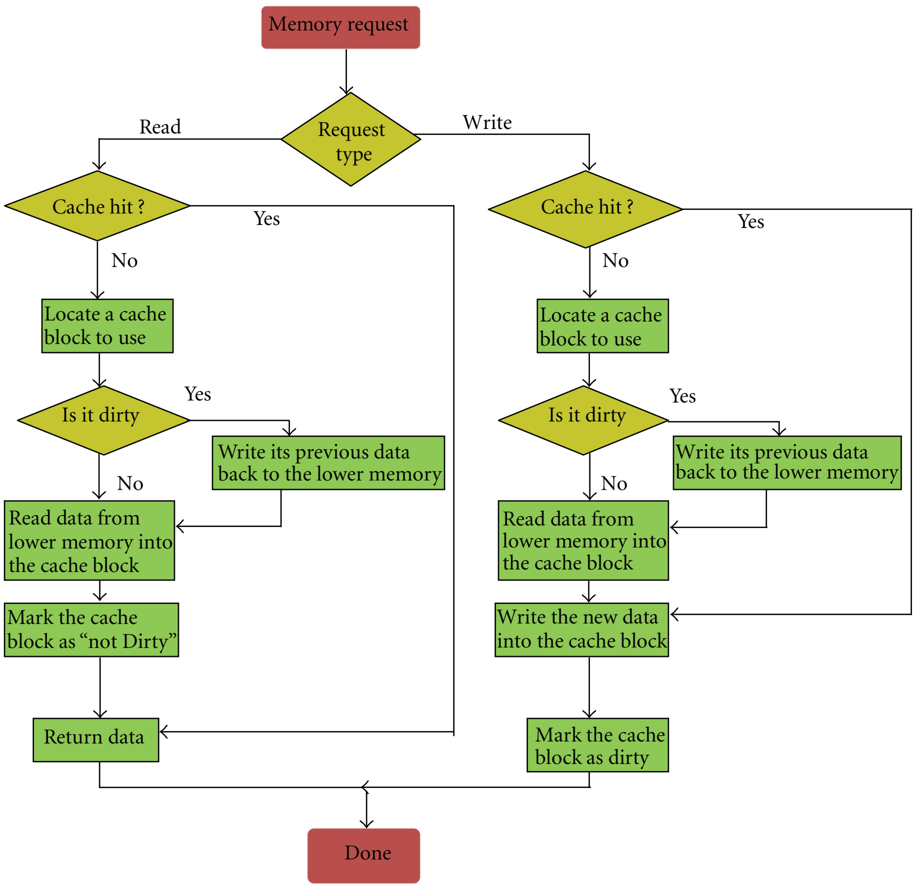

Memory request
Request type
Read
Write
Cache hit ?
Yes
No
Locate a cache block to use
Is it dirty
Yes
No
Write its previous data back to the lower memory
Read data from lower memory into the cache block
Mark the cache block as "not Dirty"
Return data
Cache hit ?
Yes
No
Locate a cache block to use
Is it dirty
Yes
No
Write its previous data back to the lower memory
Read data from lower memory into the cache block
Write the new data into the cache block
Mark the cache block as dirty
Done

Figure 6: Flowchart showing operations in memory map.

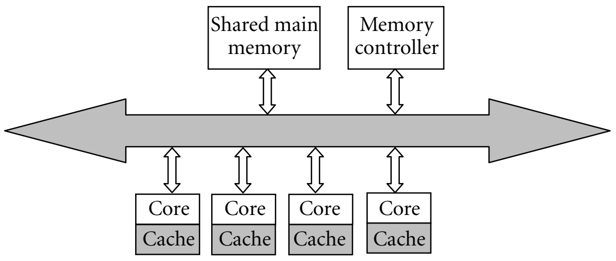


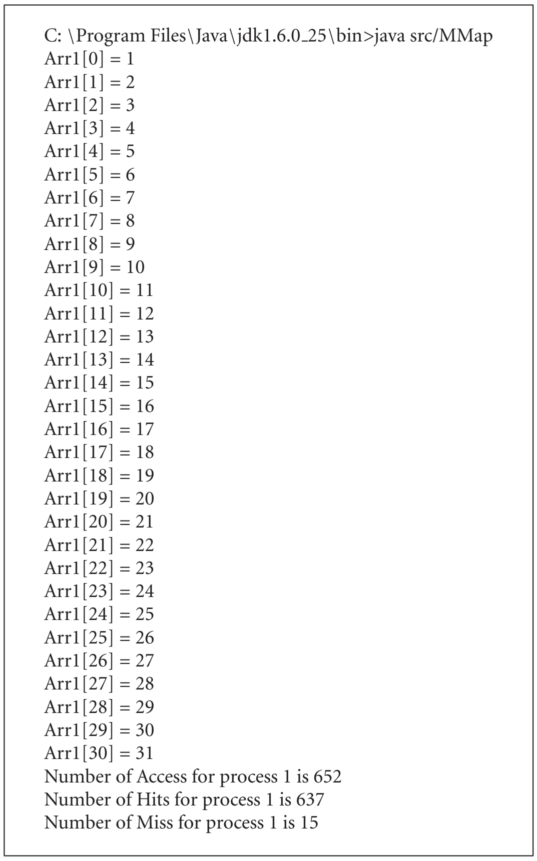
Figure 7: Proposed private cache architecture.

and average calculation in an array of 30 elements. In the similar fashion, we implemented and compared memory interleaving with normal execution of these benchmarks. We run simulations several times in order to find the correct value.

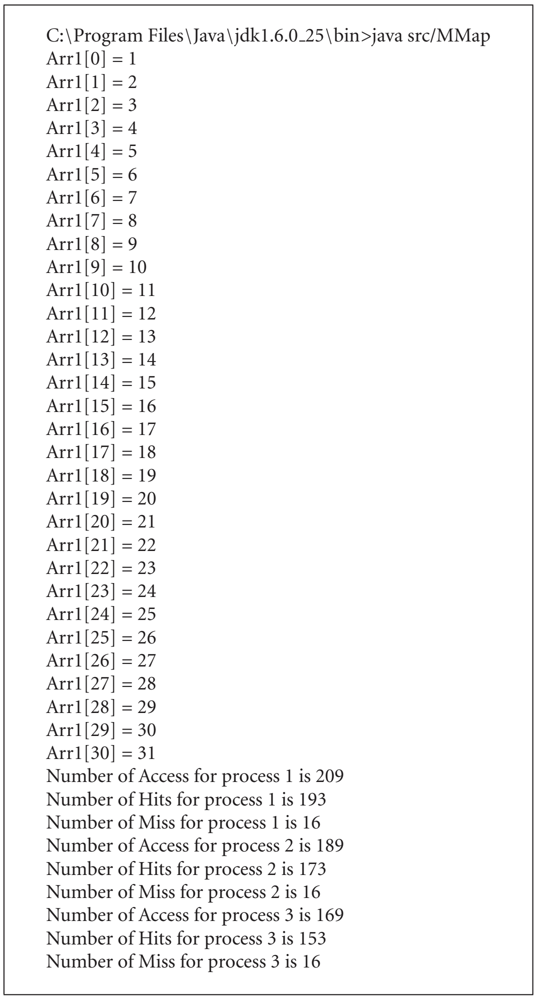
We have used Java technology (i.e., JDK 1.6) for simulations and Microsoft Excel for drawing the graphs on Windows XP platform. We have used the java.lang.reflect class to determine the size of various data types. We have used our custom convert utility package for converting various data types into bytes and vice versa. Free pool class acts as memory controller, it looks for free space which can be allocated and uses Hashmap to provide 2 levels of hashing : association of variable with memory position and association of variable with metadata, maintaining attributes related to variable, cache coherence, process association, and memory. Using CMap class, we map memory address to cache address using direct addressing mode technique.

6. Experimental Results

As shown in Figure 7, the basic system configuration consists of a variable number of cores (each having direct-mapped L2 cache of 32 bits, 64 bits, and 128 bits block sizes), one-shared memory (256 KB). The interconnect is an AMBA-compliant communication architecture. The above given architecture



```
C: \Program Files\Java\jdk1.6.0_25\bin>java src/MMap
Arr1[0] = 1
Arr1[1] = 2
Arr1[2] = 3
Arr1[3] = 4
Arr1[4] = 5
Arr1[5] = 6
Arr1[6] = 7
Arr1[7] = 8
Arr1[8] = 9
Arr1[9] = 10
Arr1[10] = 11
Arr1[11] = 12
Arr1[12] = 13
Arr1[13] = 14
Arr1[14] = 15
Arr1[15] = 16
Arr1[16] = 17
Arr1[17] = 18
Arr1[18] = 19
Arr1[19] = 20
Arr1[20] = 21
Arr1[21] = 22
Arr1[22] = 23
Arr1[23] = 24
Arr1[24] = 25
Arr1[25] = 26
Arr1[26] = 27
Arr1[27] = 28
Arr1[28] = 29
Arr1[29] = 30
Arr1[30] = 31
Number of Access for process 1 is 652
Number of Hits for process 1 is 637
Number of Miss for process 1 is 15
```



```
C:\Program Files\Java\jdk1.6.0_25\bin>java src/MMap
Arr1[0] = 1
Arr1[1] = 2
Arr1[2] = 3
Arr1[3] = 4
Arr1[4] = 5
Arr1[5] = 6
Arr1[6] = 7
Arr1[7] = 8
Arr1[8] = 9
Arr1[9] = 10
Arr1[10] = 11
Arr1[11] = 12
Arr1[12] = 13
Arr1[13] = 14
Arr1[14] = 15
Arr1[15] = 16
Arr1[16] = 17
Arr1[17] = 18
Arr1[18] = 19
Arr1[19] = 20
Arr1[20] = 21
Arr1[21] = 22
Arr1[22] = 23
Arr1[23] = 24
Arr1[24] = 25
Arr1[25] = 26
Arr1[26] = 27
Arr1[27] = 28
Arr1[28] = 29
Arr1[29] = 30
Arr1[30] = 31
Number of Access for process 1 is 209
Number of Hits for process 1 is 193
Number of Miss for process 1 is 16
Number of Access for process 2 is 189
Number of Hits for process 2 is 173
Number of Miss for process 2 is 16
Number of Access for process 3 is 169
Number of Hits for process 3 is 153
Number of Miss for process 3 is 16
```

FIGURE 8: Screen shots for plain and interleaved merge sort execution on memory map simulator block size of 128 bits.

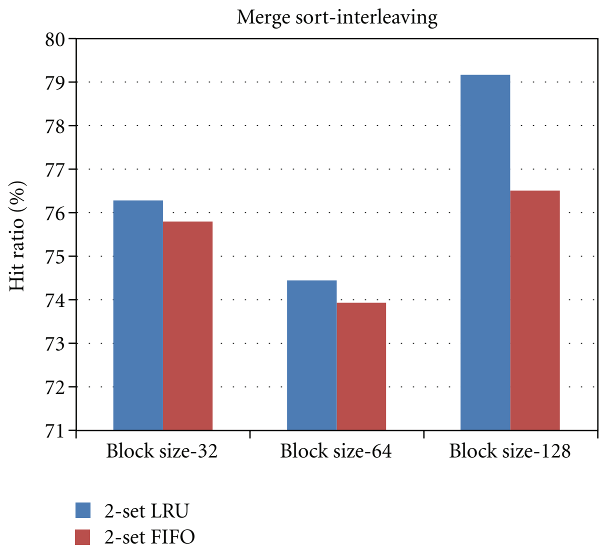


FIGURE 9: Hit ratio of replacement policy comparison—merge interleaving.

has been implemented on memory map multiprocessor simulator. Final value has been taken after running simulation 10 times.

Figure 8 shows the screen shots for merge sort execution on memory map simulator with or without using interleaving. As seen in the figure, we have taken array size of 30 elements. The simulator calculates number of accesses to the memory, number of hits, and number of misses in the memory when implementing simple merge sort algorithm. In a similar fashion, memory map evaluates number of memory accesses, misses, and hits when numbers of processors are three and memory is interleaved. Later on, we have evaluated hit ratio and miss ratio,

$$\text{Hit Ratio} = \frac{\text{No. of hits}}{\text{No. of memory accesses}},$$
$$\text{Miss Ratio} = \frac{\text{No of miss}}{\text{No of memory accesses}}, \tag{1}$$

as well as in interleaved execution. These two bars (As shown in Figures 9 and 10) are clearly showing the better

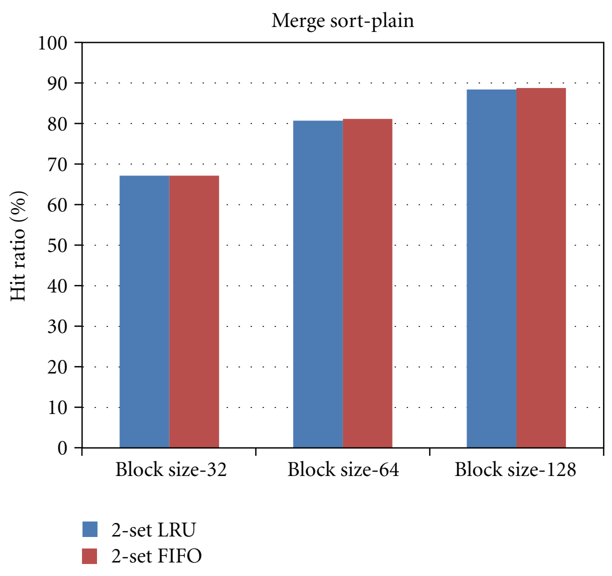
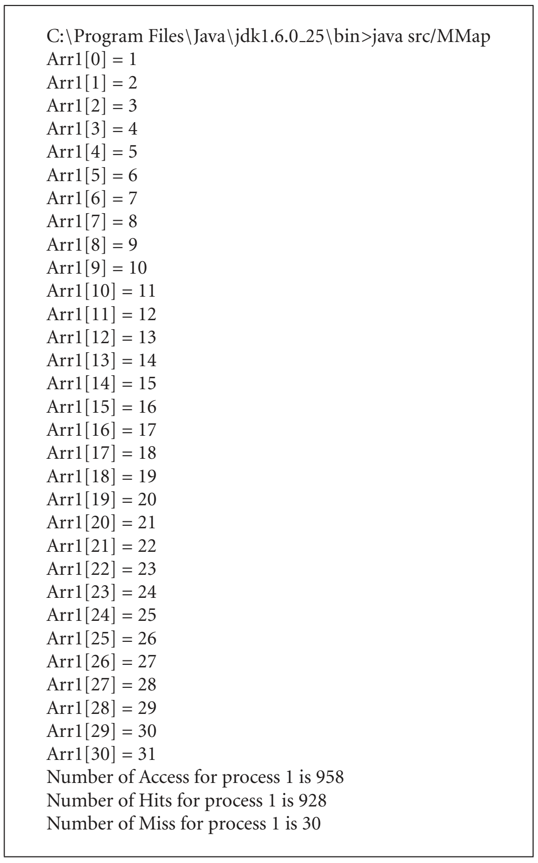


Figure 10: Hit ratio of replacement policy comparison—merge sort plain.



```
C:\Program Files\Java\jdk1.6.0_25\bin>java src/MMap
Arr1[0] = 1
Arr1[1] = 2
Arr1[2] = 3
Arr1[3] = 4
Arr1[4] = 5
Arr1[5] = 6
Arr1[6] = 7
Arr1[7] = 8
Arr1[8] = 9
Arr1[9] = 10
Arr1[10] = 11
Arr1[11] = 12
Arr1[12] = 13
Arr1[13] = 14
Arr1[14] = 15
Arr1[15] = 16
Arr1[16] = 17
Arr1[17] = 18
Arr1[18] = 19
Arr1[19] = 20
Arr1[20] = 21
Arr1[21] = 22
Arr1[22] = 23
Arr1[23] = 24
Arr1[24] = 25
Arr1[25] = 26
Arr1[26] = 27
Arr1[27] = 28
Arr1[28] = 29
Arr1[29] = 30
Arr1[30] = 31
Number of Access for process 1 is 958
Number of Hits for process 1 is 928
Number of Miss for process 1 is 30
```

Figure 11: Screen shot for bubble sort execution on memory map simulator with block size of 32 bits.

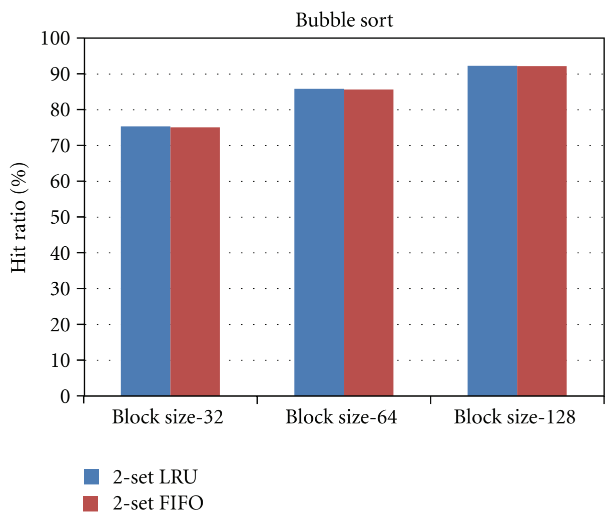


Figure 12: Hit ratio of replacement policy comparison—bubble sort.

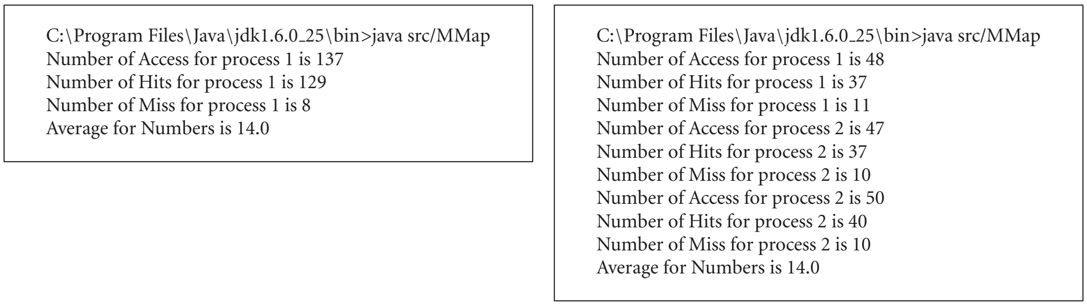
Table 1: Experimental results.

Block size	2-Set LRU	2-Set FIFO	Algorithm
Block Size-32	78.46%	76.89%	Average-Interleaving
Block Size-64	76.03%	78.58%	Average-Interleaving
Block Size-128	81.11%	83.46%	Average-Interleaving
Block Size-32	70.59%	71.17%	Average-Plain
Block Size-64	83.67%	84.42%	Average-Plain
Block Size-128	90.92%	91.77%	Average-Plain
Block Size-32	75.39%	75.14%	Bubble Sort
Block Size-64	85.90%	85.74%	Bubble Sort
Block Size-128	92.31%	92.22%	Bubble Sort
Block Size-32	76.28%	75.79%	Merge Sort-Interleaving
Block Size-64	74.44%	73.93%	Merge Sort-Interleaving
Block Size-128	79.16%	76.50%	Merge Sort-Interleaving
Block Size-32	67.12%	67.17%	Merge Sort-Plain
Block Size-64	80.67%	81.14%	Merge Sort-Plain
Block Size-128	88.40%	88.74%	Merge Sort-Plain

performance of LRU policy in comparison to FIFO when using interleaved memory and almost similar behavior in normal execution. LRU policy reaches 79% hit ratio with 128 bits block size when interleaved in a multiprocessor environment and almost 90% without interleaving with 128 bits block size.

Bubble sort execution of 30 elements on memory map simulator is shown in Figure 11. It clearly shows the values of memory accesses, number of hits, and number of misses. Hit and miss ratio are evaluated from the above given formulas and stored in Table 1.

Figure 12 drawn on the basis of results obtained in Figure 11 shows the comparison of hit ratio in two replacement policies in bubble sort using three different 32-bit, 64-bit, and 128-bit size private caches. The first two bars



```
C:\Program Files\Java\jdk1.6.0_25\bin>java src/MMap
Number of Access for process 1 is 137
Number of Hits for process 1 is 129
Number of Miss for process 1 is 8
Average for Numbers is 14.0
```

```
C:\Program Files\Java\jdk1.6.0_25\bin>java src/MMap
Number of Access for process 1 is 48
Number of Hits for process 1 is 37
Number of Miss for process 1 is 11
Number of Access for process 2 is 47
Number of Hits for process 2 is 37
Number of Miss for process 2 is 10
Number of Access for process 2 is 50
Number of Hits for process 2 is 40
Number of Miss for process 2 is 10
Average for Numbers is 14.0
```

Figure 13: Screen shot for plain and interleaved execution of average calculation on memory map simulator with block size of 32 bits.

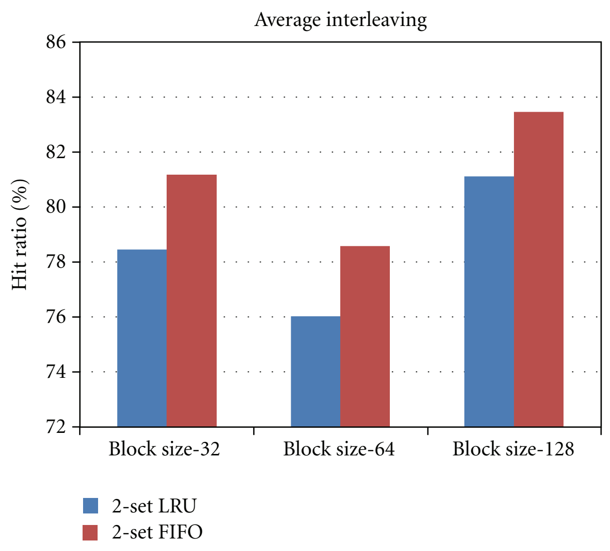


Figure 14: Hit ratio of replacement policy comparison—average interleaving.

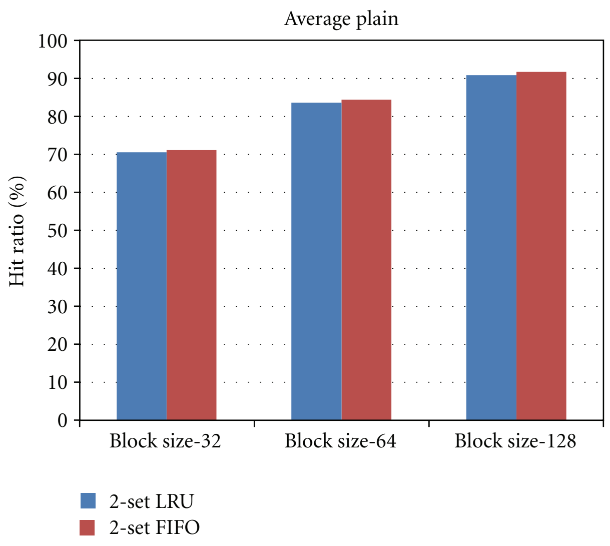


Figure 15: Hit ratio of replacement policy comparison—average plain.

are showing the result when cache size is of 32 bits and LRU outperforms FIFO when compared to cache hit rate. Similarly, when cache block size increases from 32 bits to 64 bits, cache hit rate increases from 78.3 to 85.9 and further to 92.3 when block size reaches 128 bits. There is no significance of interleaving in bubble sort due to locality of reference property.

Figure 13 clearly explains the experimental results for average calculation of 30 elements array with and without memory interleaving with 2 processors. Again the results are tabulated in Table 1. Results obtained for average benchmark are shown in Figures 14 and 15. Figure 14 shows results for interleaved memory for calculating average and results of noninterleaved execution are drawn in Figure 15 when the cache block sizes are 32 bits, 64 bits, and 128 bits. There is no increase in cache hit rate for average when not interleaved. However, when the cache size increases from 32 bits to 128 bits, cache hit significantly increases from 78% to 83% for interleaved memory. Table 1 stores the value of different scenarios.

7. Conclusion and Future Work

It has been shown that certain algorithms such as bubble sort may have little bit better cache performance than other algorithms such as merge sort due to locality of reference property. Moreover, LRU proves an outstanding performance in merge sort algorithm using memory interleaving while FIFO shows better performance in the calculation of average evaluation. This may be due to the overhead in keeping track of used pages which is not required in average calculation. In addition, as cache block size increases from 32 bits to 128 bits, hit ratio increases considerably. Sequential processes outperform concurrent processes using interleaving and private cache, when block size is more. This simulator is suitable for algorithms where no two processors need to access the same memory parts simultaneously. Further, proposed multiprocessor simulator evaluates similar results as that obtained from SimpleScalar simulator using memory interleaving proving the legitimacy of memory map multiprocessor simulator with private cache architecture.

Significant work remains to be done, as only two cache replacement algorithms LRU and FIFO are implemented till now and some other policies need to be implemented. Further, currently we are evaluating only hit and miss ratio, will implement more statistics like energy consumption, memory traces, and so forth, in future work.

References

[1] J. L. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 4th edition, 2006.

[2] M. B. Kamble and K. Ghose, "Analytical energy dissipation models for low power caches," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 143–148, August 1997.

[3] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Predictability of cache replacement policies," AVACS Technical Report no. 9, SFB/TR 14 AVACS, ISSN:18609821, 2006.

[4] R. Banakar, S. Steinke, B. S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems," in *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES '02)*, pp. 73–78, May 2002.

[5] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient utilization of scratch-pad memory in embedded processor applications," in *Proceedings of the European Design & Test Conference*, pp. 7–11, March 1997.

[6] S. Pasricha, N. D. Dutt, and M. Ben-Romdhane, "BMSYN: bus matrix communication architecture synthesis for MPSoC," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 8, pp. 1454–1464, 2007.

[7] I. Issenin, E. Brockmeyer, B. Durinck, and N. D. Dutt, "Data-reuse-driven energy-aware cosynthesis of scratch pad memory and hierarchical bus-based communication architecture for multiprocessor streaming applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 8, pp. 1439–1452, 2008.

[8] D. Cho, S. Pasricha, I. Issenin, N. D. Dutt, M. Ahn, and Y. Paek, "Adaptive scratch pad memory management for dynamic behavior of multimedia applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 4, pp. 554–567, 2009.

[9] Nomadik platform, http://www.st.com/.

[10] PC205 platform, http://www.picochip.com/.

[11] Philips nexperia platform, http://www.semiconductors.philips.com/.

[12] STMicroelectronics, http://www.st.com/internet/mcu/home/home.jsp.

[13] OMAP5910 platform, http://www.ti.com/.

[14] J. Chung, H. Chafi, C. C. Minh et al., "The common case transactional behavior of multithreaded programs," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pp. 266–277, February 2006.

[15] I. Issenin, E. Brockmeyer, B. Durinck, and N. Dutt, "Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies," in *Proceedings of the 43rd annual Design Automation Conference (DAC '06)*, pp. 49–52, 2006.

[16] B. Ackland, A. Anesko, D. Brinthaupt et al., "Single-chip, 1.6-billion, 16-b MAC/s multiprocessor DSP," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 3, pp. 412–424, 2000.

[17] Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual, http://www.cs.washington.edu/education/courses/csep501/05au/x86/24896607.pdf.

[18] S. Roy, "H-NMRU: a low area, high performance cache replacement policy for embedded processors," in *Proceedings of the 22nd International Conference on VLSI Design*, pp. 553–558, January 2009.

[19] M. A. Vega Rodríguez, J. M. Sánchez Pérez, and J. A. Gómez Pulido, "An educational tool for testing caches on symmetric multiprocessors," *Microprocessors and Microsystems*, vol. 25, no. 4, pp. 187–194, 2001.

[20] W. C. Jeun and S. Ha, "Effective OpenMP implementation and translation for Multiprocessor System-On-Chip without using OS," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '07)*, pp. 44–49, January 2007.

[21] T. D. C. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," Tech. Rep. CS-TR-1997-1342, University of Wisconsin, Madison, Wis, USA, 1997.

[22] http://www.simplescalar.com/.

[23] Jason Loew, http://www.cs.binghamton.edu/~msim/.

[24] T. Rissa, A. Donlin, and W. Luk, "Evaluation of systemC modelling of reconfigurable embedded systems," in *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, pp. 253–258, March 2005.

[25] Open SystemC Iniative OSCI, SystemC documentation, 2004, http://www.systemc.org/.

[26] C. L. Chen and C. K. Liao, "Analysis of vector access performance on skewed interleaved memory," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 387–394, June 1989.

[27] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[28] D. A. Patterson and J. L. Hennesy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kauffmann, 1994.

[29] M. Farrens and A. Park, "Dynamic base register caching: a technique for reducing address bus width," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 128–137, May 1991.

[30] A. Samih, Y. Solihin, and A. Krishna, "Evaluating placement policies for managing capacity sharing in CMP architectures with private caches," *ACM Transactions on Architecture and Code optimization*, vol. 8, no. 3, 2011.