# Effective Memory Access Optimization by Memory Delay Modeling, Memory Allocation, and Slack Time Management

Sultan Daud Khan, *Member, IACSIT*

***Abstract*—MPSoCs are gaining popularity because of its potential to solve computationally expensive applications. MPSoCs frequently use two kinds of memories; on-chip SRAMs and off-chip DRAMs. Processors in multi-core systems usually take many clock cycles for the transfer of data to/from off-chip memories which affects the overall system performance. While on-chip memory operation takes one or two clock cycles, an off-chip memory access takes significantly more number of clock cycles. Memory access delays largely depend on the ways of memory allocation and array binding. In this paper, memory delay modeling for finding accurate delays and two effective techniques, memory allocation and slack time management, are proposed for memory access optimization of off-chip DRAMs.**

***Index Terms*—Memory allocation, binding, scheduling, memory access optimization, memory management.**

## I. Introduction

For years, increasing clock speed delivers high performance for wide range of applications. Many applications become more and more complex and require a large amount of computation, so that a single processor cannot frequently satisfy the performance criteria and the designer needs to use multiple processors. A heterogeneous MPSoC consists of two or more independent and distinct microprocessors (cores), i.e., heterogeneous multi-core processors.

Multi-core systems typically use two kinds of memories; on-chip memories for local access and off-chip memories for a global access. Computationally expensive applications involve large amount of data and need to store them in cheap and large capacity off-chip memories. Each processor in multi-core system frequently accesses the global memory which takes many cycles (latency) for the transfer of data. Latency in digital systems has become a critical parameter. An important class of digital systems includes applications such as video image processing which is extremely memory expensive. In such applications, a significant amount of delay and power is consumed during memory access especially for off-chip dynamic random access memories (DRAMs). We need accurate methods for modeling and optimizing inter-core memory operations. Some part of work related to analyze the access time model for on-chip cache memories is reported in literatures [1], [2] while there is not adequate work on analyzing the access time model for off-chip memories. We developed memory delay model that finds cyclic accurate delays and optimize the memory access latency for off-chip DRAMs.

Several techniques have been introduced to optimize the memory bandwidth. To improve access bandwidth, modern memories are provided with access modes like page mode and read-modify-write mode. Different access modes of modern day memories (e.g., page-mode) are exploited in [3] to alleviate the memory bandwidth bottleneck. The memory access can be optimized to some extent by exploiting page and read-modify-write mode as in [4]. Ordering of memory operations determines possibility of exploiting the page and read-modify-write mode. Maximizing page and read-modify -write mode belongs to the class of NP-Complete problems [5]. Loop transformations are interesting alternatives to optimize the memory access latency. Techniques like code rewriting and loop transformations were used in performing data and memory related optimizations in embedded systems [6]. Techniques like loop morphing; loop fusion and loop alignment were reported in [7], [8], [9]. These techniques cannot utilize memory access latency optimally. As there are high variations in memory access delay depending on the ways of designing memory configuration and assigning arrays to memories, a technique for memory allocation and array binding is proposed in [10]. After memory allocation and binding, scheduling of operations is applied in [10] to exploit the page mode. In [11], a new method for memory allocation and assignment is proposed using multi-way partitioning. They use dual port memories which are expensive. In [11], the authors did not describe the method how to use partition algorithm to resolve the conflicts in conflict graph. For small applications, Branch & Bound and integer linear programming can be used to find the optimum solutions. But if the size of the application gets larger, these algorithms take a huge computation time to produce an optimum solution.

For such applications, heuristic algorithms can find the near optimum solutions with reasonable CPU time. In this paper, we use modified min-cut partitioning algorithm from [12] for memory allocation and assignment. The min-cut algorithm tends to find minimum cuts in conflict graph. We modified the conflict graph to maximize the cuts.

Maximizing the cuts results in resolving the maximum number of conflicts in the conflict graph. Furthermore, we apply slack time management technique in addition to exploiting the page mode for memory access optimization.

Manuscript received July 20, 2012; revised September 20.

S. D. Khan is with the Computer Engineering Department, Umm Al-Qura University, Makkah, Saudi Arabia (e-mail: sgkhan@ uqu.edu.sa).

## II. Memory Delay Modeling

Most of streaming applications are data dominated. The data for such applications are usually stored in off-chip double data rate synchronous DRAMs (DDR SDRAMs). DDR SDRAMs use double data rate architecture to achieve high speed operations. The memory address is split internally into a row address and a column address. Read and Write accesses to the DDR memories are burst-oriented. The burst length determines the maximum number of column location that can be accessed for a given READ or WRITE command. Before accessing the memory, PRECHARGE command is issued to a bank. The bank becomes idle after pre-charge delay (tRP) is met. Access to the memory starts with the registration of an ACTIVATE command which is then followed by READ or WRITE command. The address bits registered with ACTIVATE command are used to select the bank and row to be accessed. The address bits registered with READ or WRITE command are used to select the column to be accessed. CAS latency (CL) is the delay in clock cycles between the registration of a READ/WRITE command and the availability of the first bit of output data. The latencies calculated in [10] are based on the assumed values which may not be accurate. We have developed Memory Delay Model for DDR SDRAMs that finds accurate latency and also optimize the memory access latency.

Our Memory Delay Model receives each command and calculates the delay on the basis of certain rules in Fig. 1. For example, If the READ or WRITE command occurs within the row address to column address delay (*tRCD*) time, then it is shifted by the (*tRCD- t*) value. If two commands occur at the same clock cycle (collision), then the later command is shifted by one cycle. If two consecutive WRITE commands occur, the gap must be greater than or equal to two clock cycles according to a DDR DRAM data sheet. Commands are scheduled at the clock cycles where there is no violation of rules. In this way, all the memory operations are scheduled and the cyclic accurate delay is computed.

```
STEP 1 : If READ/WRITE command occurs beyond tRCD,
         no shifting is required.

STEP 2 : If READ/WRITE command occured within the tRCD,
         then shift the READ/WRITE command by (tRCD-t) value.

STEP 3 : If after shifting,
         two commands occur at the same clock cycle (collision),
         then shift the later command by one cycle

STEP 4 : Repeat STEP 3 until no collision.

STEP 5 : The gap between two consecutive WRITE command must be
         greater than or equal to two clock cycles.

STEP 6 : If READ command is followed by WRITE command,
         tWTR (Write to Read Delay) must be met.
```

Fig. 1. Rules for delay modeling

Fig. 2 shows a block of codes that contains the arrays stored in an off-chip memory. Assume that the size of each array in Fig.2 is 16 × 1024 and that all the arrays are of the same size, residing in different rows of the same memory, and Read/Write latency are 5/4 cycles respectively. Fig. 3(a) shows delay models for *op*1. In the simple delay modeling, the next command is executed after first command is completed. When READ *X*[*i*] command occurs, the next READ *X*[*i*+1] command cannot be executed during 5 clock cycles. In the cycle accurate delay modeling shown in Fig. 3(b), READ *X*[*i*] lies within the time slot of tRCD, so according to rules for delay modeling (STEP 1) in Fig.1 , it should be shifted by (*tRCD* – *t* = 3 – 0 = 3) clock cycles. After shifting, READ *X*[*i*] is rescheduled at the 3rd clock cycle. Next, READ *X*[*i*+1] occurred at 2nd clock cycle. As it also lies within the time slot of tRCD, it is shifted in the same way by (*tRCD* – *t* = 3 – 2 = 1) clock cycles. READ *X*[*i*+1] cannot be scheduled at the 3rd clock cycle because READ *X*[*i*] is already scheduled at the same clock cycle. READ *X*[*i*+1] is thus shifted and rescheduled at the 4th clock cycle to avoid a collision. As shown in Fig. 3(b), the total delay can be reduced from 19 to 12 clock cycles (36.8% reduction) by using our cycle accurate delay model, instead of the simple worst case delay model.

```
for ( i = 1; i < nm1; ++i )
{
     op1: d[i] = x[i+1] - x[i];
     op2: b[i] = d[i+1] + d[i];
     op3: c[i+1] = ( y[i+1] - y[i] ) / d[i];
     op4: c[i] = c[i+1] - d[i];
     op5: d[i] = d[i] * 3;
}
```

Fig. 2. Motivational example

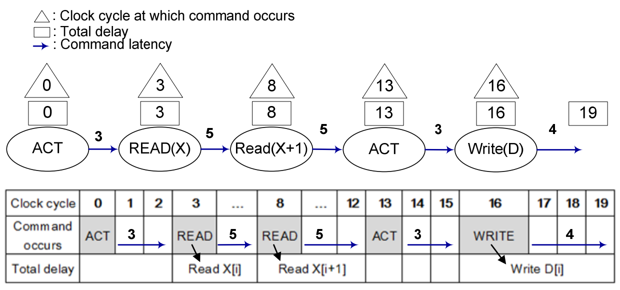


Fig. 3(a). Simple delay modeling for *op*1

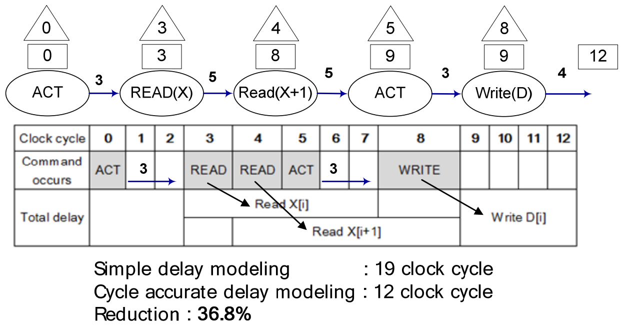


Fig. 3(b). Cycle accurate delay modeling for *op*1

If the arrays are allocated to a single memory module, it results in limiting parallel access to the memory and hence increases the overall latency. But, if the arrays are allocated to distinct memory modules, it allows the parallel access and results in reducing latency. This shows that there is a tradeoff between the area and latency.

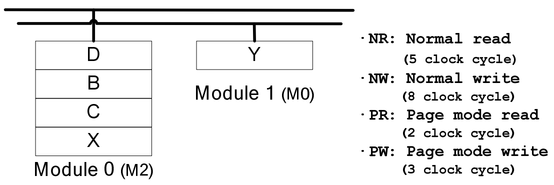
## III. Memory Allocation

In streaming application, the size of arrays is very large and usually stored in off-chip memories. In most memory intensive applications, array access is a dominant factor in the total memory access latency. Usually arrays within body of loops accessing off-chip memory result in overall memory access latency. We identify those blocks of source codes which result in frequent access to off-chip memory

TABLE I: MEMORY MODULE LIBRARY

| Memory Module | Size (Bits * Words) | Area(mm$^2$) |
|---|---|---|
| $M_0$ | 16 × 1024 | 5.432 |
| $M_1$ | 16 × 2048 | 7.658 |
| $M_2$ | 16 × 4096 | 10.830 |
| $M_3$ | 32 × 2048 | 21.661 |

As shown in Fig. 4, the memory access latency is 65 clock cycles when the arrays *B*[], *C*[], *D*[], *X*[] are allocated to Module 1($M_2$) and array *Y*[] is allocated to Module 0($M_0$). The M2 and M0 are memory module types from the memory module library shown in Table I. The total area is the sum of module sizes of $M_0$ and $M_2$. We assume that Normal Read (NR), Normal Write (NW), Page Read (PR), Page Write (PW) take five, eight, two, and three clock cycles, respectively, as in [10].


D
B
C
X
Module 0 (M2)
Y
Module 1 (M0)
·NR: Normal read (5 clock cycle)
·NW: Normal write (8 clock cycle)
·PR: Page mode read (2 clock cycle)
·PW: Page mode write (3 clock cycle)

```
op1.  Read X[i](NR)                          ; NR(5) //  5 clk
      Read X[i+1](PR)                        ; PR(2) //  7 clk
      Write D[i](NW)                         ; NW(8) // 15 clk
op2.  Read D[i](PR)                          ; PR(2) // 17 clk
      Read D[i+1](PR)                        ; PR(2) // 19 clk
      Write B[i](NW)                         ; NW(8) // 27 clk
op3.  Read D[i](NR)      Read Y[i](NR)       ; NR(5) // 32 clk
                         Read Y[i+1](PR)     ; PR(2) // 34 clk
      Write C[i+1](NW)                       ; NW(8) // 42 clk
op4.  Read C[i+1](PR)                        ; PR(2) // 44 clk
      Read D[i](NR)                          ; NR(5) // 49 clk
      Write C[i](NW)                         ; NW(8) // 57 clk
op5.  Read D[i](NR)                          ; NR(5) // 62 clk
      Write D[i](PW)                         ; PW(3) // 65 clk
```

Total latency per iteration : **65** clock cycle
Total area(mm$^2$): **16.262** ( Using memory module M0, M2 )

Fig. 4. An example of memory binding and allocation

Our approach is to analyze those blocks that result in overall memory access latency and to optimize memory access for these arrays. We group together two or more arrays and find the number of instructions in which the arrays in the selected group are simultaneously accessed. Table II shows the access information of all the arrays in Fig. 2. The second column of the table shows the most frequently accessed elements of arrays and the last column shows the total number of accesses of the arrays. After getting all the access information, the arrays are sorted on the basis of total number of accesses in decreasing order. {*D* [], *C* [], *X* [], *Y* [], *B* []} is the access order of the arrays in Fig. 2. We now group the arrays and find the number of instructions in which these arrays are simultaneously accessed. Table III shows the group of arrays and the number of instructions which contain the arrays in a group. In order to maximize the parallel access, we place the arrays in separate memories that belong to the same group with large number of instructions. At the beginning, we make a group of two arrays. By using the access information table and array grouping, we generate the conflict graph. The memory bandwidth requirement can be modeled as a conflict graph as shown in Fig. 5(a). The nodes in graph represent application arrays and edges connect arrays accessed in parallel. The weight of an edge represents the number of instructions in which these arrays are accessed in parallel. In the Fig. 5, arrays *C* and *D* should not be placed in the same memory because there is conflict between them. However, there is no edge between *B*, *X* and *Y*, hence we can place these arrays in same memory without conflict. Fig. 5 shows that there are five conflicts in conflict graph. We need to find techniques to resolve maximum conflicts using min-cut partitioning algorithm. For this purpose, complement of the conflict graph with weight re-adjustment is proposed. To obtain the complement graph, we readjust the edge weights. Let m be the maximum edge weight in the original graph, then each edge weight e is replaced by (m-e) in the complement graph. The adjusted weights of the complement graph are as shown in Fig. 5(b). If we use 2-way partitioning, the cost is 7 and we can resolve all conflicts except (*D*, *Y*) as shown in Fig. 5(b). If we use 3-way partitioning, as in Fig. 5(c), the min-cut algorithm made the maximum cut and hence all the conflicts are resolved as arrays *X*, *B* and *Y* are allocated to one memory module and *C* and *D* are allocated to other two modules. So the designer can select the number of memory modules considering area constraints and hardware cost. Fig. 5(c) shows memory allocation and binding results using min-cut partitioning algorithm for the example in Fig. 2. Fig. 6 shows that memory access latency is reduced to 46 clock cycles; by parallelizing the array access resulted from partition algorithm in Fig. 5(c). We see that, memory access latency is reduced from 65 clock cycles to 46 clock cycles (25.8% reduction) by maximizing the parallel access.

TABLE II: ACCESS INFORMATION TABLE

| Array Groups | Array Elements | # of Read Access | # of Write Access | Total # of Access |
|---|---|---|---|---|
| *D*[ ] | *D*[*i*] | 4 | 2 | 7 |
| | *D*[*i*+1] | 1 | 0 | |
| *X*[ ] | *X*[i] | 1 | 0 | 2 |
| | *X*[*i*+1] | 1 | 0 | |
| *B*[ ] | *B*[*i*] | 0 | 1 | 1 |
| *Y*[ ] | *Y*[*i*] | 1 | 0 | 2 |
| | *Y*[*i* +1] | 1 | 0 | |
| *C*[ ] | *C*[*i*] | 1 | 1 | 4 |
| | *C*[*i* +1] | 1 | 1 | |

TABLE III: ARRAY GROUPING

| Array Groups | # of Instructions | Array Groups | # of Instructions |
|---|---|---|---|
| { *D*, *C* } | 2 | { *D*, *Y* } | 1 |
| { *D*, *X* } | 1 | { *C*, *Y* } | 1 |
| { *D*, *B* } | 1 | { *D*, *C*, *Y* } | 1 |

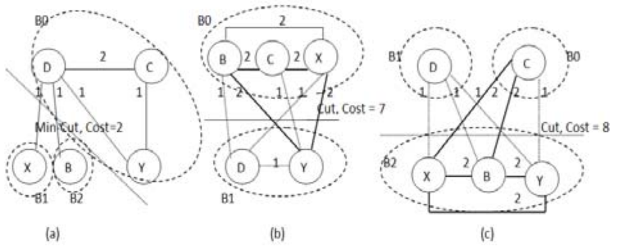
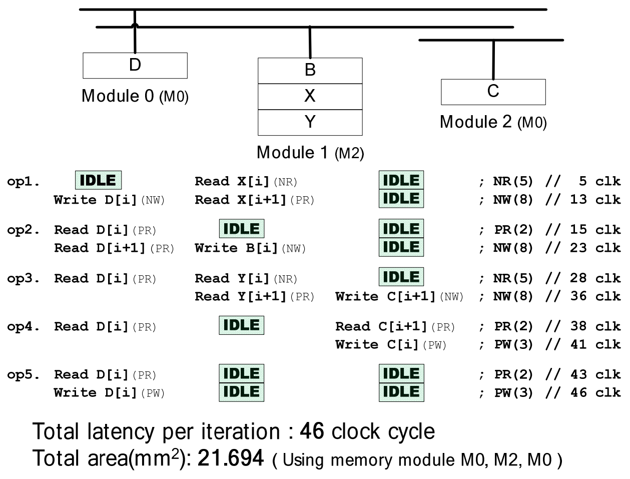


Fig. 5. (a) Original conflict graph. (b) Complimented 2-way partitioning result. (c) Complimented 3-way partitioning result.



```
op1.    IDLE              Read X[i](NR)       IDLE              ; NR(5) //  5 clk
        Write D[i](NW)    Read X[i+1](PR)     IDLE              ; NW(8) // 13 clk

op2.    Read D[i](PR)     IDLE                IDLE              ; PR(2) // 15 clk
        Read D[i+1](PR)   Write B[i](NW)      IDLE              ; NW(8) // 23 clk

op3.    Read D[i](PR)     Read Y[i](NR)       IDLE              ; NR(5) // 28 clk
                          Read Y[i+1](PR)     Write C[i+1](NW)  ; NW(8) // 36 clk

op4.    Read D[i](PR)     IDLE                Read C[i+1](PR)   ; PR(2) // 38 clk
                                              Write C[i](PW)    ; PW(3) // 41 clk

op5.    Read D[i](PR)     IDLE                IDLE              ; PR(2) // 43 clk
        Write D[i](PW)    IDLE                IDLE              ; PW(3) // 46 clk
```

Total latency per iteration : 46 clock cycle
Total area(mm$^2$): 21.694 ( Using memory module M0, M2, M0 )

Fig. 6. Proposed array binding and memory allocation using partitioning algorithm using the complement graph

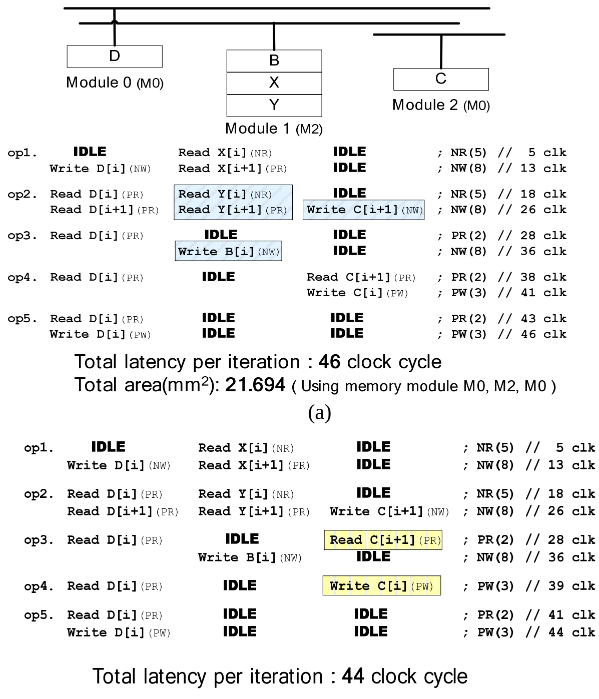
## IV. Slack Time Management

In this section, slack time management is introduced to further optimize the memory access latency. Arrays are assigned to memories after memory allocation, in order to maximize the parallel access and resolve the conflicts up to maximum extent. After memory allocation, we exploit the page mode as much as possible. The memory access latency can be reduced to some extent by exploiting the page mode as reported in [3], [4]. But exploiting the page mode is NP-Complete [4]. In some cases, exploiting the page mode cannot optimize the memory access up to maximum extent as shown in Fig. 7 (a). Fig. 7 (a) shows worst case example where memory access latency remains same after exploiting the page mode. In this way, we introduce slack time management technique to further optimize the memory access latency. During execution of streaming applications different arrays are accessed at different times. Therefore, some of memories remain idle while other memories are accessed. The memories remain idle when no memory operation is issued.

We can utilize this idle time to further optimize the memory access latency. As shown in Fig. 6, Module 2 remains idle as no memory operation is issued during the execution of *op*1. We can utilize this idle time, by issuing memory operations to the idle memory and reading or writing the data which can be used later for other operations. As shown in Fig. 7(a), during the execution of *op*2, Module 2 remains idle. We can issue the read operation (Read *C* [*i*+1]) that can be used by *op*4 as shown in Fig. 7(b). In this way, we can further reduce memory access latency from 46 to 44 clock cycles. Slack time management requires additional registers and can be restricted by dependencies among operations.

## V. Experimental Results

We tested our proposed approach with the set of random examples. Table IV shows the benchmarks and the conflict resolving ratio by using proposed partitioning algorithm with two, three, or four memory modules for allocation. Experimental results show that proposed approach resolves 75%, 95%, and 97% of conflicts when two, three, and four memory modules are used, respectively. After memory allocation and exploiting the page mode, we apply slack time management technique to further optimize the memory access latency. In Table V, we can see the exploiting the page mode optimizes the memory access by up to 16% (avg. 2% to 8%). After applying slack time management technique, the memory access latency is reduced by up to 32% (avg. 17% to 20%) with additional register requirement.



```
op1.    IDLE              Read X[i](NR)       IDLE              ; NR(5) //  5 clk
        Write D[i](NW)    Read X[i+1](PR)     IDLE              ; NW(8) // 13 clk

op2.    Read D[i](PR)     Read Y[i](NR)       IDLE              ; NR(5) // 18 clk
        Read D[i+1](PR)   Read Y[i+1](PR)     Write C[i+1](NW)  ; NW(8) // 26 clk

op3.    Read D[i](PR)     IDLE                IDLE              ; PR(2) // 28 clk
                          Write B[i](NW)      IDLE              ; NW(8) // 36 clk

op4.    Read D[i](PR)     IDLE                Read C[i+1](PR)   ; PR(2) // 38 clk
                                              Write C[i](PW)    ; PW(3) // 41 clk

op5.    Read D[i](PR)     IDLE                IDLE              ; PR(2) // 43 clk
        Write D[i](PW)    IDLE                IDLE              ; PW(3) // 46 clk
```

Total latency per iteration : 46 clock cycle
Total area(mm$^2$): 21.694 ( Using memory module M0, M2, M0 )

(a)

```
op1.    IDLE              Read X[i](NR)       IDLE              ; NR(5) //  5 clk
        Write D[i](NW)    Read X[i+1](PR)     IDLE              ; NW(8) // 13 clk

op2.    Read D[i](PR)     Read Y[i](NR)       IDLE              ; NR(5) // 18 clk
        Read D[i+1](PR)   Read Y[i+1](PR)     Write C[i+1](NW)  ; NW(8) // 26 clk

op3.    Read D[i](PR)     IDLE                Read C[i+1](PR)   ; PR(2) // 28 clk
                          Write B[i](NW)      IDLE              ; NW(8) // 36 clk

op4.    Read D[i](PR)     IDLE                Write C[i](PW)    ; PW(3) // 39 clk

op5.    Read D[i](PR)     IDLE                IDLE              ; PR(2) // 41 clk
        Write D[i](PW)    IDLE                IDLE              ; PW(3) // 44 clk
```

Total latency per iteration : 44 clock cycle

Fig. 7. (a) Exploiting the page mode. (b) Exploiting page mode with slack/idle time management.

TABLE IV: Summary of Benchmarks

| Bench marks | # of arrays | Total # of conflict | Using min-cut partitioning | | Proposed approach | | Conflicts resolved |
|---|---|---|---|---|---|---|---|
| | | | Conflict resolved | Cost | Conflict resolved | Cost | |
| IBM01 | 4 | 3 | 1 | 2 | 3 | 6 | 100% |
| IBM02 | 6 | 7 | 1 | 2 | 6 | 15 | 85% |
| IBM03 | 8 | 14 | 4 | 9 | 12 | 22 | 85% |

TABLE V: Access Information Table

| Bench marks | # of arrays | Memory allocation (cycle) | Exploiting Page Mode | | Slack Time Management | |
|---|---|---|---|---|---|---|
| | | | Clock cycle | Reduction | Clock cycle | Reduction |
| IBM01 | 4 | 42 | 31 | 26% | 19 | 38% |
| IBM02 | 6 | 139 | 115 | 14% | 100 | 13.04% |
| IBM03 | 8 | 215 | 188 | 12.5% | 169 | 10% |
| IBM15 | 15 | 2085 | 1855 | 11% | 1613 | 13% |
| Avg | | | | 12.5% | | 11.5% |

## VI. Conclusion

We have developed memory delay model for finding the cycle accurate latency for off-chip memories and optimizing the memory access latency. We have developed an effective technique for memory allocation by using modified conflict graph. Furthermore, we introduced slack time management technique to further optimize the memory access time. Experimental results show that memory access latency can be reduced by effective memory allocation techniques which maximize parallel access. Memory access latency can be further optimized by exploiting the page mode and by using slack time management techniques.

## REFERENCES

[1] T. Wada, S. Rajan, and S. A. Przbylski, *An Analytical Access Time Model for on-Chip Cache Memories*, vol. 27, pp. 1147-1156, 1992.

[2] S. J. E. Wilton and N. P. Jouppi, *CACTI: An Enhanced Cache Access and Cycle Time Model*, vol. 31, pp. 667-688, 1996.

[3] P. R. Panda, N. D. Dutt, and A. Nicolau, "Incorporating DRAM access modes into high-level synthesis," *IEEE Transaction on Computer-Aided Design*, vol. 17, pp. 96-106, Feb. 1998.

[4] P. R. Panda, N. D. Dutt, and A. Nicolau, "Exploiting off-chip memory access modes in high-level synthesis," *International Conference on Computer-Aided Design (ICCAD '97)*, pp. 333, 1997.

[5] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg, "Data and memory optimization techniques for embedded systems," *ACM Transaction Design Automation Eletron. Syst*, vol. 6, no. 2, pp. 149-206, Apr. 2001.

[6] J. I. Gomez, P. Marchal, S. Verdoorlaege, L. Pinuel, and F. Catthoor, "Optimizing the memory bandwidth with loop morphing," *15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'04)*, pp. 213-223, 2004

[7] P. Marchal, J. I. Gomez, and F. Catthoor, "Optimizing the memory bandwidth with loop fusion," *IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS'04)*, pp. 188-193, 2004.

[8] A. Fraboulet, G. Huard, and A. Mignotte, "Loop alignment for memory access optimization," *12th International Symposium on System Synthesis*, pp. 71, 1999.

[9] T. Kim and J. Kim, "Integration of code scheduling, memory allocation, and array binding for memory access optimization," *IEEE Trans. CAD*, vol. 26, no. 1, pp. 142-151, 2007.

[10] N. Kim and R. Peng, "A memory allocation and assignment method using multiway partitioning," *IEEE International SoC Conference*, 2004.

[11] H. Shin and C. Kim, "A simple yet effective technique for partitioning," *IEEE Transaction on Very Large Integration (VLSI) System*, pp. 380- 386, 1993.

**Sultan Daud Khan** received the BS degree in Computer Engineering from University of Engineering & Technology, Peshawar, in 2005 and MS degree in Electronics and Communication Engineering from Hanyang University, South Korea, in 2010.During his MS studies, his research was mainly focused on Off-Chip memory access optimization for MPSoCs. He was a member of Digital System Laboratory (DSL).Currently he is a Lecturer and Research Assistant in Department of Computer Engineering of Umm Al-Qura University, Saudi Arabia since Jan,2011 where he is working on Hajj Core (high funded project). His main responsibilities are developing computer vision's applications for detecting and estimating number of people in high density crowds. He worked as a Design Engineer in And Or Logics, a renowned multi-national company, pvt, Pakistan where he worked on military projects. He was also a part-time Design Engineer in North-West Research Company, Pakistan. He is a member of IACSIT.