# HyMacs: Hybrid Memory Access Optimization based on Custom-instruction Scheduling [*]

Kang Zhao, Jinian Bian, Sheqin Dong
EDA Lab, Dept. Computer Science & Technology
Tsinghua University, Beijing 100084, China
zhao-k04@mails.tsinghua.edu.cn
{bianjn,dongsq}@tsinghua.edu.cn

Yang Song and Satoshi Goto
School of Information, Production and Systems
Waseda University, Kitakyushu, 808-0135 Japan
syang@asagi.waseda.jp
goto@waseda.jp

## ABSTRACT

This paper presents an efficient hybrid memory access optimization system called HyMacs, which integrates the hardware and software optimization strategies in the embedded system design. First, HyMacs features a pre-configuration stage which is equipped with a memory configuration algorithm to satisfy area constraints. Then a custom instruction generation process is integrated in the system via a seed-growth algorithm under the intelligent guide functions. The custom instruction benefits to the reduction of the whole memory access latency and thus relieves the burden of system through hardware mode. Finally, a data-dependency-driven scheduling algorithm is also integrated to compress the whole latency through access mode conversion. We have tested the system on a set of commonly used benchmarks, and compared the results with the previous memory access system MACCESS-opt proposed in DAC'05. The experimental results indicate 20% enhancement obtained for the total memory access latency reduction compared with MACCESS-opt, where the custom instruction generation and scheduling contribute about 15% and 5% respectively.

## Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*methodologies*; C.3 [**Special purpose and application-based systems**]: Real-time and embedded systems

## General Terms

Algorithms, Design, Experimentation

## Keywords

CAD algorithm, hardware/software co-design, ASIP

[*]This work was supported in part by National Natural Science Foundation of China under grant NSFC-90207017, and NSFC-90067001; National Basic Research Program of China (973) under grant 2005CB321605.

## 1. INTRODUCTION

Memory issues often play a very important role in the embedded system design, which impact significantly the embedded system's performance, power, and the cost of implementation [1]. To accelerate the memory access, the use of efficient access mode is greatly encouraged to promote the access bandwidth [2]. However, speed requirement is not principal for all target application markets, instead, how to get an efficient area becomes the main task. So the area-speed tradeoff of the memory access must be explored.

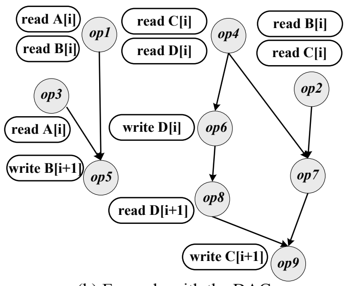
To address this issue and compress the memory access latency as much as possible, a system named as HyMacs is proposed in this paper, which uses a hybrid memory access optimization to accelerate the memory access. This strategy is similar to the hardware/software co-design strategy. Recently, many works have been done to optimize the memory access, and they can be classified to two types: software optimization and hardware optimization. Here we will give a brief introduction for compare.

In literature, much work has been done utilizing the software optimization mode. Kim presented a typical algorithm called MACCESS-opt in [3]. MACCESS-opt utilized the efficient page access, and considered three techniques simultaneously: determination of memories, array mapping to memories, and scheduling of memory access operations, so that the use of DRAM page access mode was maximized. The details of the page access will be presented in Section 2. Kim improved this algorithm with a rescheduling technique in the successive work [4]. The advantage of MACCESS-opt is that it can achieve a maximum speedup by scheduling the code under area/cost constraints. However, because software optimization must be limited under the hardware constraints, this algorithm can only get about 15-25% reduction on the whole latency compared to the random generation. In this paper we will use it as the baseline algorithm for comparison purpose, and utilize a hardware/software co-design strategy to settle this issue. Besides, Choi studied the problem of DRAM memory layout for storing non-array variables to maximum the use of memory bandwidth [5]. To reduce the access delay, Choi proposed an efficient memory layout algorithm for the page access mode. Similarly, our focus in this paper is on DRAM memory, which is the same with [5]; however, our emphasis is on the array variables, which is different from Choi's work.

The memory access optimization can also be achieved by hardware mode. The strategy is adopting custom instructions to minimize the memory access latency based on the application specific instruction-set processor (ASIP). ASIP

| Memory modules | Modules size (bits * words) | Area (mm$^2$) |
|---|---|---|
| M1 | 16*1024 | 5.4154 |
| M2 | 32*1024 | 10.8309 |
| M3 | 16*2048 | 7.6586 |
| M4 | 32*2048 | 15.3171 |
| M5 | 16*4096 | 10.8308 |
| M6 | 32*4096 | 21.6617 |
| M7 | 16*8192 | 15.3171 |

(a) Memory module library

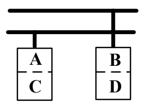


(b) Example with the DAG (directed acyclic graph) format

```
op1:  e = A[i] + B[i]        ; read A[i], read B[i]
op2:  f = B[i] - C[i]        ; read B[i], read C[i]
op3:  g = A[i] − 3           ; read A[i]
op4:  h = C[i] + D[i]        ; read C[i], read D[i]
op5:  B[i+1] = e − g         ; write B[i+1]
op6:  D[i] = h               ; write D[i]
op7:  i = f − h
op8:  t = D[i+1] + 4         ; read D[i+1]
op9:  C[i+1] = i + t         ; write C[i+1]
```

(c) The source code for the DAG example in (b)

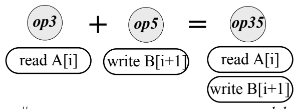
**Memory configuration:**



```
op#                                         delay
1. read A[i]   NR     read B[i]   NR        :NR
2. read C[i]   NR     read B[i]   PR        :NR
3. read A[i]   NR                           :NR
4. read C[i]   NR     read D[i]   NR        :NR
5.                    write B[i+1] NW       :NW
6.                    write D[i]   NW       :NW
8.                    read D[i+1]   PR      :PR
9. write C[i+1]   PW                        :PW
            Area: 18.4894 mm^2
          Total latency: 41 cycles
```

(d) Memory configuration for the operation schedule in (c)

**Custom instruction generation:**

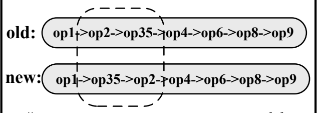


```
op#                                            delay
1.   read A[i]   NR      read B[i]   NR        :NR
2.   read C[i]   NR      read B[i]   PR        :NR
35.  read A[i]   NR                            :NR
                        write B[i+1] PW        :PW
4.   read C[i]   PR      read D[i]   NR        :NR
6.                       write D[i]   PW       :PW
8.                       read D[i+1]  PR       :PR
9.   write C[i+1]   PW                         :PW

          Total latency: 31 cycles
```

(e) Custom instruction generation under the memory configuration in (e)

**Scheduling:**


old:
op1->op2->op35->op4->op6->op8->op9
new:
op1->op35->op2->op4->op6->op8->op9

```
op#                                            delay
1.   read A[i]   NR      read B[i]   NR        :NR
35.  read A[i]   PR                            :PR
                        write B[i+1] PW        :PW
2.   read C[i]   NR      read B[i]   PR        :NR
4.   read C[i]   PR      read D[i]   NR        :NR
6.                       write D[i]   PW       :PW
8.                       read D[i+1]   PR      :PR
9.   write C[i+1]   PW                         :PW

          Total latency: 28 cycles
```

(f) Scheduling results based on the custom instruction in (e)

**Figure 1: Example to illustrate the motivation of memory configuration, custom instruction generation and scheduling. It is assumed that the page read (PR), normal read (NR), page write (PW) and normal write (NW) will use 2, 5, 3 and 8 clock cycles respectively. Those data are spurious only for analysis and compare.**

is the processor designed for particular applications, which provides a tradeoff between efficiency and flexibility, and custom instruction in ASIP can be viewed as hardware for special purposes to accelerate the computation [6]. In this domain, Biswas described the problems resulted from local memory operations after the instruction set extension (ISE) in [7]. Since memory operations pose the challenge for ISE approaches by limiting the size of resulting instructions, this paper presented two kinds of memory elements into custom units, and proposed a genetic algorithm to exploit the opportunities of introducing memory elements during ISE generation. Then Dimond drew a conclusion and proposed the methodology to optimize the memory access through the specific instruction set customization [8]. Furthermore, some commercial tools on ASIP have been presented which can settle the issues related with memory access, such as the Xtensa-core-based toolset from Tensilica Inc. [9].

Given an objective application, it is concluded that design methodologies provide two ways of either scheduling the code or synthesize instructions to satisfy the requirements of time/area tradeoff. However, each way has its own intrinsic limitation and cannot obtain the maximum potential of the latency reduction. For example, the software optimization must be implemented under the hardware constraints; and the precondition of the hardware optimization is the memory allocation, which is implemented by software optimization.

The new system called HyMacs which is proposed in this paper can settle this issue effectively, and it contains three steps: memory configuration, custom instruction generation and instruction scheduling. The experimental results prove that considering a hybrid design strategy of software and hardware can obtain a better improvement than considering only one of them.

The rest of the paper is organized as follows. In Section 2, the motivation of memory access optimization is given. And in Section 3, the problem formulation is described. Then in Section 4, the details of the proposed algorithms are given. To verify the new system, the experimental result is shown in Section 5. Finally, a conclusion is drawn in Section 6.

## 2. MOTIVATION

Firstly, this section presents the definitions of normal mode access and page mode access in DRAMs, which is similar to [4]. In normal mode, a row decoding stage is first used to copy the entire row of words to the row buffer, and then a column decoding stage is adopted similarly. Finally, a precharging stage is performed to prepare the execution for the next memory access operation. For page mode, however, if the word to be accessed in the next operation has already been in the same page that was retrieved just before, the execution of row decoding is not needed. So the latency of page mode is much shorter than the normal mode, and their difference focuses on whether the utilizing array variables are the same or not between neighbor operations. This paper will use the memory module library shown in Fig. 1(a).

Therefore, under the memory area constraints, convert as many as possible normal access mode to page access mode is the key to reduce the whole access latency. In this paper, we use three techniques to achieve this objective: memory configuration, custom instruction generation and scheduling. To explain it clearly, we will use the DAG example which is illustrated in Fig. 1(b).

Table 1: Definitions for notations.

| Notations | Meaning |
|---|---|
| $Arrays$ | Set of array names in the program |
| $M$ | Set of memory module names |
| $f$ | Mapping from $Arrays$ to $M$ |
| $S$ | Scheduling of the instructions |
| $weight_{max}$ | The maximum weight of subgraphs |
| $hier/hier_{max}$ | $hier$ represents the current level in the hierarchical DAG; $hier_{max}$ is the maximum level |
| $area(f), Area_{max}$ | $area(f)$ means the area under the memory allocation $f$; $Area_{max}$ is the area constraint |
| $Weight(G')$ | Graph weight of $G'$ |
| $delay(v)$ | Latency for the node $v$ |

1) Different memory allocation can result in different area cost and different latencies; furthermore, memory allocation is the essential conditions for other processes. So firstly the memory configuration should be confirmed. As illustrated in Fig. 1(d), array variables $A$ and $C$ are assigned into the same memory module. So if $A$ and $C$ appear in the same operation, they cannot be accessed in parallel. If we change the results in Fig. 1(d) and put variables $A$ and $B$ into the same module, both *op*1 and *op*4 will be accessed through two NRs, and the total access latency will be longer. This is just the motivation of finding the best memory configuration.

2) Custom instruction is the combination of multiple basic operations, and it can be viewed as the hardware for special purposes. How can it reduce the memory access latency? For example shown in Fig. 1(e), when *op*3 is combined with *op*5, the access mode for *op*5 will be converted from normal write (NW) to page write (PW) compared with Fig. 1(d). Besides, since *op*5 disappears and *op*6 will run after *op*4 directly, so the access mode of *op*6 should be changed from NW to PW. Additionally, suppose that we combine *op*1 and *op*3 together instead, and then the normal read operation for *op*3 can be omitted. Therefore, we can see that the custom instruction generation not only can change the memory access mode, but also can reduce the access count.

3) The reason to utilize the normal access mode is that the previous operation uses different arrays. So we can make the same arrays exist in the previous operation via scheduling. For example illustrated by Fig. 1(f), when the position of *op*2 and *op*35 is exchanged, the page read (PR) mode will be changed to normal read because of the variable $A[i]$ in *op*35, and then the total latency is reduced. Therefore, it is feasible to reduce the whole latency through scheduling based on the custom instructions.

## 3. PROBLEM FORMULATION

To explain the formulation clearly, we first present several preliminary definitions:

Definition 1. *$G(V, E)$ is a directed acyclic graph (DAG), where $V$ and $E$ are the sets of nodes and edges. The weight of $G(V, E)$ is defined as the length of its longest path.*

Since the custom instruction is synthesized by combining basic operations, so if each basic operation is mapped to a certain node in the DAG, then each custom instruction will be mapped to a feasible sub-graph in the DAG. The graph weights of sub-graphs should be limited to a value $weight_{max}$, because when the graph weight is larger, the complexity of the customization algorithm will be higher.

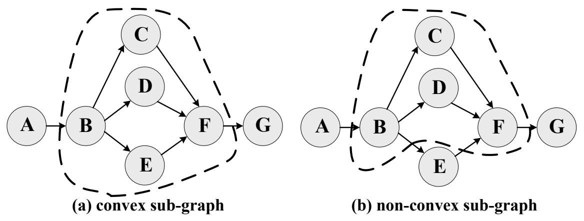


Figure 2: Convex and non-convex sub-graphs.

Definition 2. *$G'(V', E')$ is a sub-graph of $G(V, E)$. For $\forall v_1, v_2 \in V'$, if all the nodes on the paths between $v_1$ and $v_2$ are also contained in $V'$, $G'$ will be convex, as shown in Fig. 2; Otherwise, $G'$ will be non-convex.*

C programs must be implemented with a sequential order. If a sub-graph in DAG is mapped to a custom instruction, it must be convex. Because non-convex graph can result in the non-automated execution for the instructions, and this is not permitted in the program implementation.

Definition 3. *Each custom instruction can be mapped to a feasible sub-graph $G'$, which satisfies two conditions: 1) the number of incoming and outgoing arcs are limited, because the ports of the register file is also limited. 2) $G'$ must be a convex sub-graph.*

Following the three definitions above and the notations defined in Table 1, the problem can be described as:

**Problem:** Given a DAG $G(V, E)$ of high-level source code with array access $Arrays$, DRAM module library $M$ and memory area constraint $Area_{max}$, then generate a scheme of memory allocation $f$, custom instruction set and the scheduling $S$, so that the total memory access latency $delay(G)$ is minimum not exceeding $Area_{max}$.

Finally, the formulation description is given as follows:

$$\begin{aligned} Min: \quad & \sum_{i=1}^{|V|_{hier=hier_{max}}} delay({}^f_S G'_i) \\ s.t. \quad & area(f : Arrays \rightarrow M) \leq Area_{max} \\ & Weight({}^f_S G'_i) \leq weight_{max} \\ & 1 \leq hier \leq hier_{max} \end{aligned}$$

Here $\{{}^f_S G'_i\}$ means the set of sub-graphs under the memory configuration and scheduling results. $G'_i$ is the $i^{th}$ sub-graph.

## 4. METHODOLOGY

To reduce the memory access latency while satisfying the area constraints, we utilize the hardware/software (HW/SW) co-design strategy and propose a new system called HyMacs (hybrid memory access through custom-instruction-based scheduling). This system includes three techniques: memory configuration, custom instruction generation and instruction scheduling.

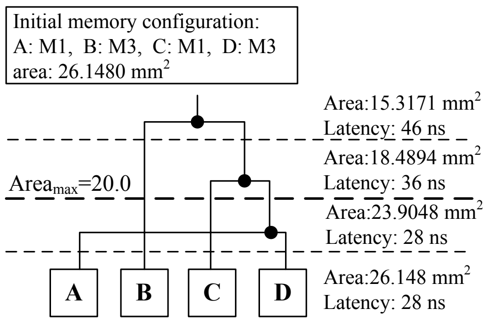


**Figure 3: Example for the memory configuration.**

## 4.1 Memory Configuration

Because the focus of custom instruction and scheduling is to reduce the total latency of memory access, and the memory allocation result is just the precondition of latency calculation, so the motivation of this step is to satisfy the memory area constraint first. The configuration algorithm is the same with [3], so we only give a brief explanation for it. Because the area for each memory module has been defined in the Fig. 1(a), we can consider to combine each two arrays together from low latency to high latency, as shown in Fig. 3. Since the area cannot exceed the maximum value $Area_{max}$, so the result near to the deadline will be selected, and the final result indicates that variables $A$, $C$ and $D$ are assigned to the same memory module.

## 4.2 Custom Instruction Generation

The motivation of this step is to reduce the total latency via instruction customization, because custom instruction has the effect to change the access mode and then reduce the memory access latency. The basic strategy of the instruction customization is to combine basic operations in the DAG, i.e. sub-graph selection. However, for a DAG $G$ with $n$ nodes, there will be $2^n$ candidate sub-graphs. To reduce this exponential search space, we will present the definitions of seed node and two guide functions $F_1$ and $F_2$.

To avoid searching the useless candidates and reduce the huge search space, we first select some seed nodes under the guide function $F_1(distance, priority)$, and then grow from them under the best direction which is decided by the guide function $F_2(distance, reduction)$.

1) The variable *distance* denotes the space between the current node and the longest path of the DAG. Because combining multiple nodes on the longest path will compress the access latency and improve the performance directly, if the value of *distance* is smaller, the current node will hold higher potential to be contained in the candidate sub-graph.

2) The parameter *priority* is defined as follows:

$$priority = in + out + in \times out \tag{1}$$

Where *in* and *out* represent the numbers of incoming and outgoing arcs. In Eq.1, the third item is used to deal with the situation that the value of either *in* or *out* is zero. When the variable *priority* is bigger, the potential of selecting the current node as the seed node is higher. The reason to select the node with highest value of *priority* as seed node is that it can reduce the total number of seed nodes, which is helpful to reduce the complexity of the algorithm.

```
Custom_Instruction($G$, $f$, $S$)
/* $G$: graph, $f$: memory allocation, $S$: schedule order */
 1. while(all nodes in $G$ are visited)
 2.    Select node $i$ with the maximum $priority$ on the
         longest path $p$;
 3.    set $i$ as the current seed node;
 4.    $Q$ = set of nodes connected with $i$;
 5.    for(each node $j$ in $Q$){
 6.       combine node $i$ and $j$ as $t$;
 7.       if($t$ is convex and $Weight(t) \leq weight_{max}$){
 8.          $G' = G - \{i, j\} + \{t\}$;
 9.          $\triangle L = delay(G, f, S) - delay(G', f, S)$;
10.       }endif
11.    }endfor
12.    select $j$ in $Q$ with the largest value of $\triangle L$;
13.    Combine node $i$ and $j$ and update the graph $G$;
14. }endwhile
15. return $G$;
```

**Figure 4: Custom instruction generation algorithm.**

3) Suppose that variable *reduction* represents the quantity of the latency reduction after the current node is combined with the seed node. When the value of *reduction* is larger, the potential of reducing the total latency will be more.

Following the discussion above, the two guide functions $F_1$ and $F_2$ are defined in Eq.2. $F_1$ is used to select the seed node which is nearer the longest path and hold a high priority; $F_2$ is adopted to choose the node which should be combined with the current seed node, and this node is near to the longest path and can obtain a higher reduction for the access latency. In Eq.2, the item of "*distance*+1" is to prevent the situation that the value of *distance* is zero.

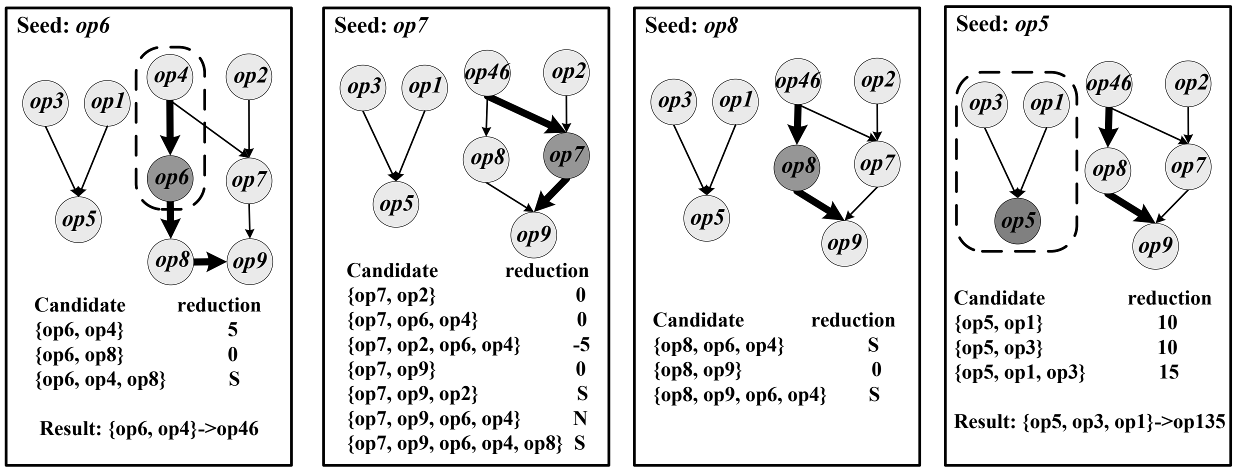
$$F_1 = \frac{distance + 1}{priority} \qquad F_2 = \frac{reduction}{distance + 1} \tag{2}$$

The details of our algorithm are shown in Fig. 4. Its main idea is to determine the seed node with the minimum $F_1$, and then select the node connected with the current seed node with maximum $F_2$. Finally the whole graph is updated under the generation of new custom instructions.

For example, Fig. 5 shows the custom instruction process in detail for the DAG in Fig. 1(b), which is under the results of memory allocation in Fig. 1(d). From the leftmost to the right, we can find that the longest path will be changed once a new instruction appears, and the values of guide functions $F_1$ and $F_2$ for each node are also changed synchronously. In this process, we can notice that several candidates are not feasible sub-graphs. As the example shown in Fig. 5(a), when merging *op*6, *op*4 and *op*8, it will exceed the maximum value of sub-graph weight which is assumed to be 2 in this example; and in Fig. 5(b), when merging *op*7, *op*9 and *op*46, a non-convex sub-graph will be generated and it is unfeasible. Therefore, the constraint of feasible sub-graphs is one of the conditions to stop the iteration in the algorithm.

## 4.3 Instruction Scheduling

After custom instruction generation is finished, we will schedule the order based on the specific instructions so that maximum access modes can be changed from normal mode to page mode. However, not all the positions of operations can be exchanged because of the data dependency con-


Seed: op6
op3 op1 op4 op2 op6 op7 op5 op8 op9
Seed: op7
op3 op1 op46 op2 op8 op7 op5 op9
Seed: op8
op3 op1 op46 op2 op8 op7 op5 op9
Seed: op5
op3 op1 op46 op2 op8 op7 op5 op9

| Candidate | reduction |
|---|---|
| {op6, op4} | 5 |
| {op6, op8} | 0 |
| {op6, op4, op8} | S |

Result: {op6, op4}->op46

(a) seed=*op6*

| Candidate | reduction |
|---|---|
| {op7, op2} | 0 |
| {op7, op6, op4} | 0 |
| {op7, op2, op6, op4} | -5 |
| {op7, op9} | 0 |
| {op7, op9, op2} | S |
| {op7, op9, op6, op4} | N |
| {op7, op9, op6, op4, op8} | S |

(b) seed=*op7*

| Candidate | reduction |
|---|---|
| {op8, op6, op4} | S |
| {op8, op9} | 0 |
| {op8, op9, op6, op4} | S |

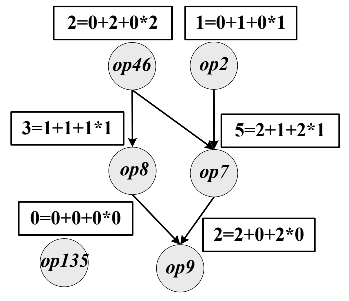
(c) seed=*op8*

| Candidate | reduction |
|---|---|
| {op5, op1} | 10 |
| {op5, op3} | 10 |
| {op5, op1, op3} | 15 |

Result: {op5, op3, op1}->op135

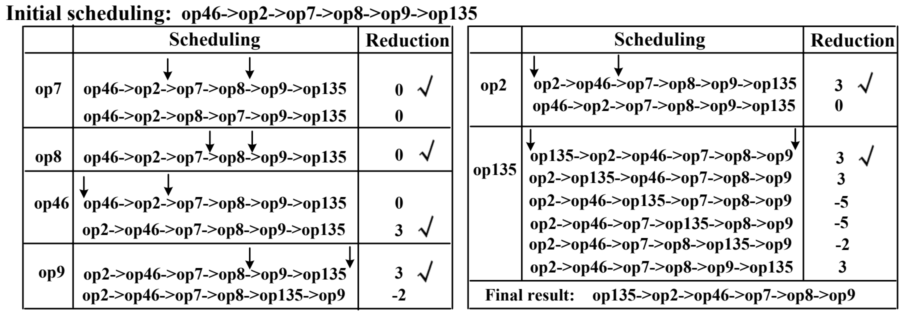
(d) seed=*op5*

**Figure 5: Example explaining for the seed-growth algorithm of custom instruction generation. Here uses the DAG in Fig. 1(b). The thick lines denote the longest path, and the node filled with shadow is the seed node.** $S$**: the sub-graph exceeding the maximum weight, which is 2 in this example;** $N$**: non-convex sub-graph.**



**Priority**: *op7>op8>op46=op9>op2>op135*

(a) Priority enumeration for each operation



Initial scheduling: op46->op2->op7->op8->op9->op135

| | Scheduling | Reduction |
|---|---|---|
| op7 | op46->op2->op7->op8->op9->op135 | 0 √ |
| | op46->op2->op8->op7->op9->op135 | 0 |
| op8 | op46->op2->op7->op8->op9->op135 | 0 √ |
| op46 | op46->op2->op7->op8->op9->op135 | 0 |
| | op2->op46->op7->op8->op9->op135 | 3 √ |
| op9 | op2->op46->op7->op8->op9->op135 | 3 √ |
| | op2->op46->op7->op8->op135->op9 | -2 |

| | Scheduling | Reduction |
|---|---|---|
| op2 | op2->op46->op7->op8->op9->op135 | 3 √ |
| | op46->op2->op7->op8->op9->op135 | 0 |
| op135 | op135->op2->op46->op7->op8->op9 | 3 √ |
| | op2->op135->op46->op7->op8->op9 | 3 |
| | op2->op46->op135->op7->op8->op9 | -5 |
| | op2->op46->op7->op135->op8->op9 | -5 |
| | op2->op46->op7->op8->op135->op9 | -2 |
| | op2->op46->op7->op8->op9->op135 | 3 |

Final result: op135->op2->op46->op7->op8->op9

(b) Scheduling results following the priority order in (a). Each pair of arrows represent the begin and end positions of the shift range for the current scheduling.

**Figure 6: Example explaining for the scheduling algorithm based on the custom instructions. Figure (a) is the illustration for the priority computation, and (b) shows the scheduling process in detail based on the priority order in (a). In (b) each pair of arrows represents the positions of the movement range, i.e. [*begin*, *end*]. The hooks shown in (b) mean that the current scheduling will be selected.**

straint. For example in Fig. 1(b), the nodes $op1$ and $op5$ cannot be exchanged because the variable e must be written in $op1$ first, and then it can be read in $op5$.

The strategy for this scheduling process is to sort the instruction order first from high priority to low priority by the rule with Eq.1, as shown in Fig. 6(a). Then each operation $v$ will be selected under this priority order and is moved during the current scheduling to find the maximum latency reduction. In this movement, the iteration is used to confirm the position of the operation $v$. However, the move range of $v$ is limited in [*begin*, *end*], where *begin* stands for the nearest operation which has an outgoing arc to $v$, and *end* is the nearest operation which has an incoming arc from $v$. For example as shown in Fig. 6(b), when the current node is $op7$, then its move range is [$op7$, $op8$]. After the positions of all instructions have been confirmed, this process will stop.

The details of the scheduling algorithm are shown in Fig. 7. Compared to the scheduling algorithms in [3], our algorithm compresses the exploration space through two techniques: 1) Instead of the random order, we utilized the order related with priority under the rule of Eq.1. When *priority* is higher, the node will be on the middle part of one path, and if the position of this node is confirmed first, the movement range for other nodes on the same path will be reduced; 2) Instead of moving operations during the whole scheduling, we limit the range between *begin* and *end*, which considers the data dependency constraint adequately.

## 5. EXPERIMENTAL RESULTS

We have implemented the HyMacs algorithms in C++ language, and have tested them by a set of benchmarks in numerical recipes [10], which are the same with the benchmarks in [3] only for compare. The basis processor core is obtained from Tensilica [9], its speed is 319 MHz and it is equipped with a RAM of 1G size. In the custom instruction generation, fusion type of instructions is mainly used based on Xtensa core. Table 2 illustrates the experimental results on the test benchmarks, where the first column shows the names of the benchmarks, the second one denotes

**Table 2: Performance improvement results for the memory access latency (page size: 16)**

| Benchmark | array/area | MACCESS-opt | HyMacs ($weight_{max}$=2) | | | | HyMacs ($weight_{max}$=3) | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | #instr. | #imp. | #sched. | #imp. | #instr. | #imp. | #sched. | #imp. |
| FOURFS | 6/21.5 | 150 | 122 | 18.67% | 122 | 18.67% | 90 | 40.00% | 84 | 40.00% |
| SPLINE | 4/17.5 | 69451 | 61902 | 10.87% | 54353 | 21.74% | 49824 | 28.26% | 49824 | 28.26% |
| STOERM | 4/17.5 | 44642 | 39198 | 12.19% | 39198 | 12.19% | 37020 | 17.07% | 37020 | 17.07% |
| PZEXTR | 6/28.5 | 143172 | 135972 | 5.03% | 128412 | 10.31% | 129888 | 9.28% | 122508 | 14.43% |
| RATINT | 4/4.25 | 284308 | 261563 | 8.02% | 216074 | 24.00% | 170585 | 39.99% | 170585 | 39.99% |
| Average improvement | | ⁄ | 10.96% | | 17.38% | | 26.92% | | 28.75% | |

```
Scheduling($G$, $f$, $S$)
/* $G$: graph, $f$: memory allocation, $S$: schedule order */
 1. for(each node $i$ in $G$){
 2.     $priority_i = in_i + out_i + in_i \times out_i$;
 3. }endfor
 4. $P$={operations in $G$ from high priority to low};
 5. for(each node $j$ in the set $P$){
 6.     $begin$=position of node $j$'s predecessor in $S$ and $G$;
 7.     $end$ = position of node $j$'s successor in $S$ and $G$;
 8.     move $j$ in the range [$begin$, $end$];
 9.     $S_1$={Speedup when $j$ is moved in [$begin$, $end$]};
10.     Select the largest element in the set $S_1$;
11.     $S \Leftarrow S_1$;
12. }endfor
13. return $S$;
```

**Figure 7: Scheduling algorithm on the instructions.**

the array number and area results under the memory configuration, and the third column presents the latency results utilizing the previous system MACCESS-opt in [3] for compare. Then the access latency results with the proposed system HyMacs are presented under different constraints. The page size is supposed to be 16. Column under "#instr" gives the latency results after the custom instruction generation, and "#sched" gives the final results after scheduling process based on custom instructions. Finally, "#imp" represents the improvements on the reduction of memory access latency compared with the previous system MACCESS-opt.

From Table 2 we can find that when the maximum value of subgraph weight increases, the improvements rise because the number of custom instructions is raised. In conclusion, HyMacs system can achieve about 20% improvements than the system MACCESS-opt in [3], where custom instructions and scheduling contribute about 15% and 5% respectively. This can also be validated by the Fig. 8, which illustrates the average improvements on the latency for each benchmark. Here the first figure shows the case that $weight_{max} = 2$, and the second figure shows the case that $weight_{max} = 3$.

## 6. CONCLUSION

In this paper, we propose a hybrid memory access system to reduce the whole memory access latency which integrates the custom instruction generation and scheduling algorithms. By applying a hardware/software co-design strategy, our hybrid system has obtained a significant improvement on the access latency reduction than the previous system which only considers the software optimization.

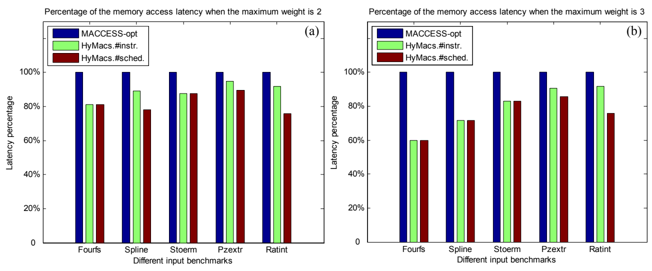


**Figure 8: Average speedup on the access latencies.**

## 7. REFERENCES

[1] P. R. Panda, et al. "Data and memory optimization techniques for embedded systems," *ACM Trans. Des. Autom. Electr. Syst.*, vol. 6, no. 2, pp. 149-206, 2001.

[2] Betty Prince. "High Performance Memories, New Architecture DRAMs and SRAMs Evolution and Function," West Sussex, U.K.: Wiley, 1996.

[3] J. Kim, T. Kim. "Memory access optimization through combined code scheduling, memory allocation, and array binding in embedded system design," in *Proc. DAC'05*, Anaheim, pp.105-110, 2005.

[4] T. Kim, 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, Jan. 2007.

[5] Y. Choi, et al. "Memory layout techniques for variables utilizing efficient DRAM access modes," *IEEE Trans. CAD*, vol. 24, no. 2, pp. 278-287, Feb. 2005.

[6] M. K. Jain, M. Balakrishnan and A. Kumar. "ASIP Design Methodologies: Survey and Issues," in *Proc. VLSI Design*, Bangalore, pp. 76-81, Jan. 2001.

[7] P. Biswas, V. Choudhary, et al. "Introduction of local memory elements in instruction set extensions," in *Proc. DAC'04*, pp.729-734, 2004.

[8] R. Dimond, O. Mencer, et al. "Automating Processor Customization: Optimised Memory Access and Resource Sharing," in *Proc. DATE'06*, pp.1-6, 2006.

[9] Tensilica Inc.: http://www.tensilica.com/

[10] W. H. Press, et al. "Numerical Recipes in C." U.K: Cambridge University press, 1992.