Hybrid Memory Access Optimization based on Custom-instruction Scheduling

Kang Zhao, Jinian Bian, Sheqin Dong,

Yang Song and Satoshi Goto

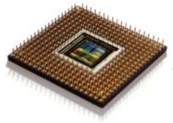
EDA Lab, Dept. Computer Science & Technology

Tsinghua University, Beijing 100084, China

May 5, 2008

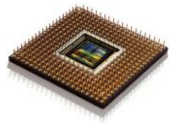
Contents

- Introduction
- Motivation
- Problem Formulation
- Methodology
- Conclusion



Contents

- Introduction
- Motivation
- Problem Formulation
- Methodology
- Conclusion



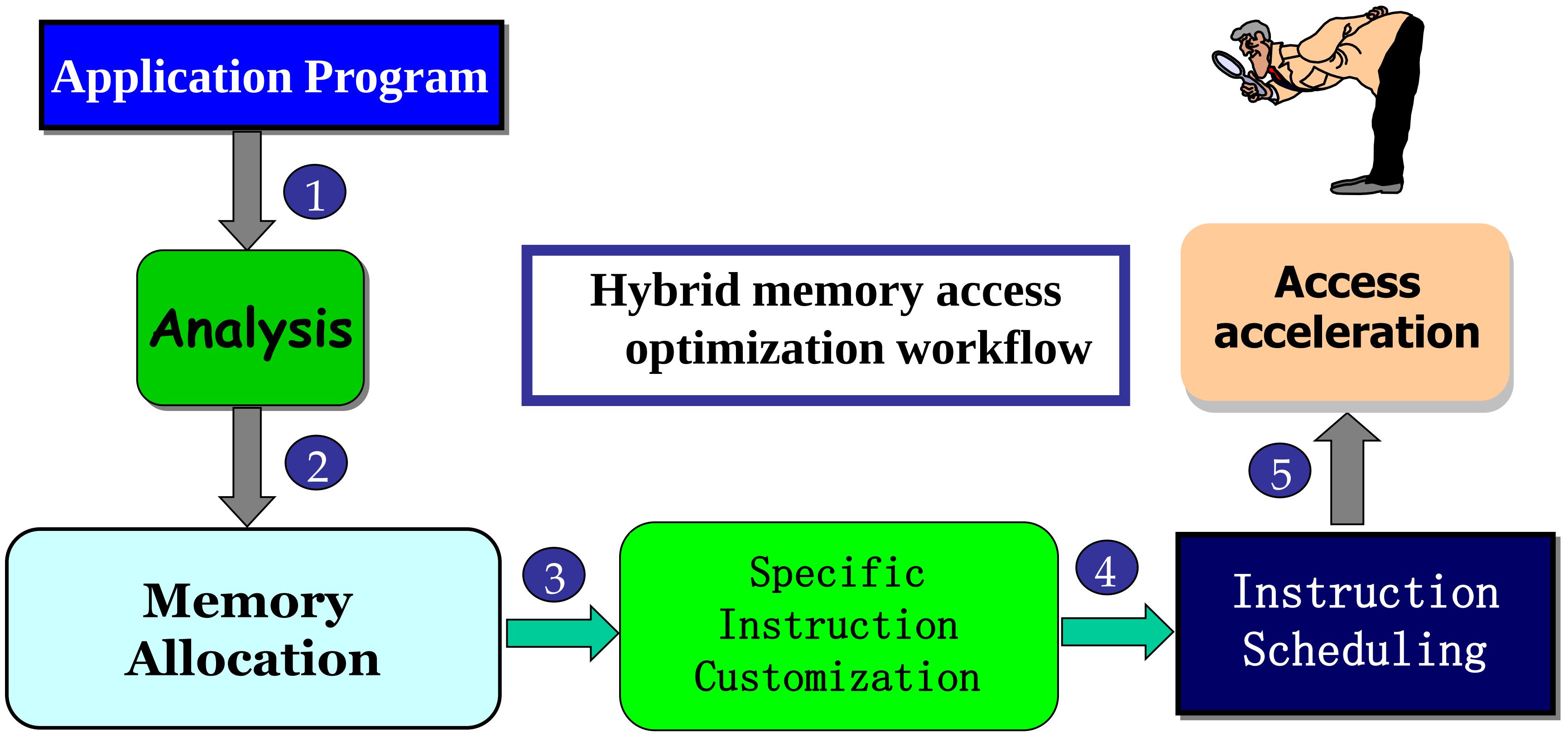
Introduction

- Memory access
  - 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
  - To accelerate the memory access, the use of efficient access is greatly encouraged to promote the access bandwidth
  - However, speed requirement is not principal for all applications, instead, how to get an efficient area becomes the main task. So the area-speed tradeoff of the memory access must be explored
- Focus
  - Page access, which is one of the most efficiently used DRAM accesses

Literature

- Software optimization mode
  - [3] proposed an algorithm called MACCESS-opt. It used the efficient page access, and considered three techniques: determination of memories, array mapping to memories, and scheduling of memory access operations
  - The advantage of MACCESS-opt is that it can achieve a maximum speedup by scheduling the code under area/cost constraints
- Hardware optimization mode
  - The strategy is adopting custom instructions to minimize the total memory access latency based on the ASIP
  - ASIP = General core + application specific instructions
  - Custom instructions in ASIP can be viewed as hardware for special purpose to accelerate the processor
- Limitations
  - 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

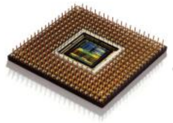
Hybrid Optimization Workflow



An illustration for the proposed hybrid method based on HW/SW co-optimization

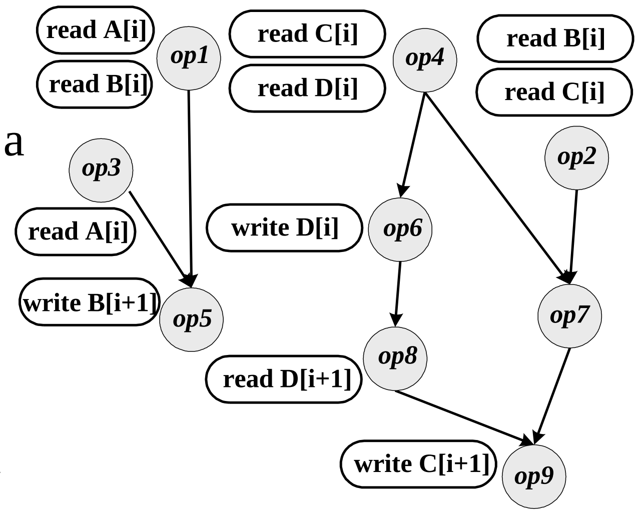
Contents

- Introduction
- Motivation
- Problem Formulation
- Methodology
- Theoretical Analysis
- Conclusion



Normal mode & Page mode

- Normal access 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 used
  - Finally, a precharging stage is performed to prepare the execution for the next memory access operation
- Page access mode
  - If the word to be accessed in the next operation has already been in the same page that was retrieved just before, then the execution of row decoding is not needed
  - 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
- Key: convert normal mode to page mode as many as possible



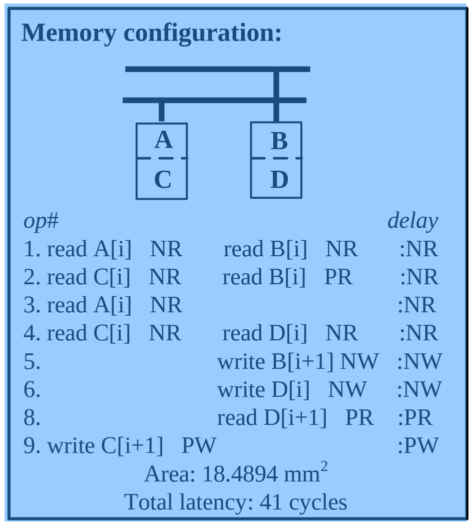
(b) Example with the DAG

```
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 example (b)

Motivational Example

- Memory Allocation
  - Variables *A* and *C* are assigned into the same memory module. If *A* and *C* appear in the same operation, they cannot be accessed in parallel
  - If we change the results in Fig.(d) and put *A* and *B* into the same module, both *op*1 and *op*4 will be accessed through two NRs, and the latency will be longer
  - page read (PR), normal read (NR), page write (PW), normal write (NW)

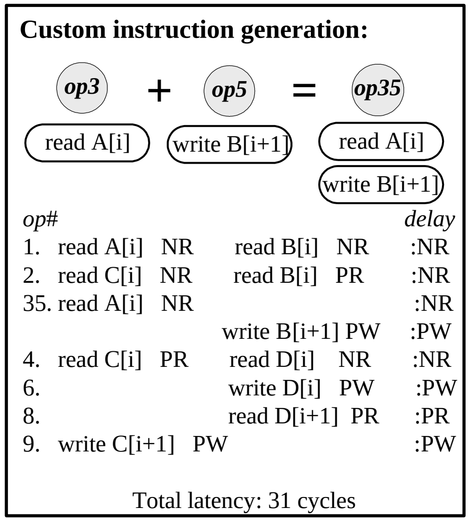


op#			delay
1.	read A[i] NR	read B[i] NR	:NR
2.	read C[i] NR	read B[i] PR	:NR
3.	read A[i] NR		:NR
4.	read C[i] NR	read D[i] NR	:NR
5.		write B[i+1] NW	:NW
6.		write D[i] NW	:NW
8.		read D[i+1] PR	:PR
9.	write C[i+1] PW		:PW

Area: 18.4894 mm^2

Total latency: 41 cycles

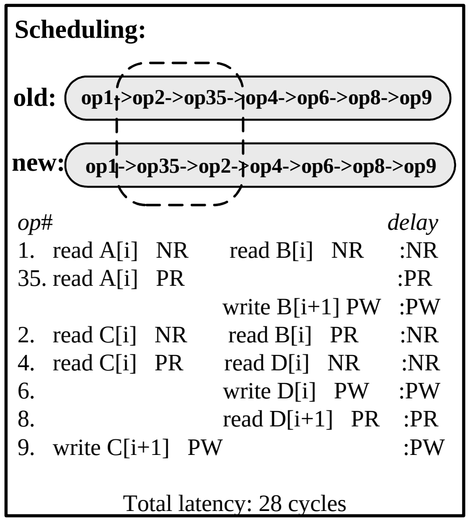
(d) Memory configuration for the operation schedule in (c)



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 (d)


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

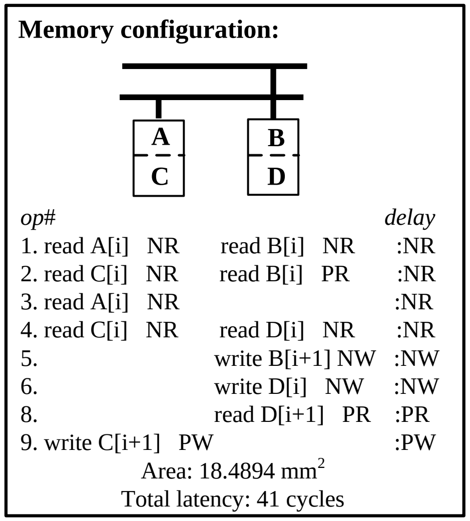
op#			delay
1.	read A[i] NR	read B[i] NR	:NR
35.	read A[i] PR		:PR
		write B[i+1] PW	:PW
2.	read C[i] NR	read B[i] PR	:NR
4.	read C[i] PR	read D[i] NR	:NR
6.		write D[i] PW	:PW
8.		read D[i+1] PR	:PR
9.	write C[i+1] PW		:PW

Total latency: 28 cycles

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

Motivational Example

- Custom instruction generation
  - If *op*3 is combined with *op*5, the mode for *op*5 will change from NW to PW
  - 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
  - Suppose that we combine *op*1 and *op*3 together instead, and then the NR latency for *op*3 will be omitted

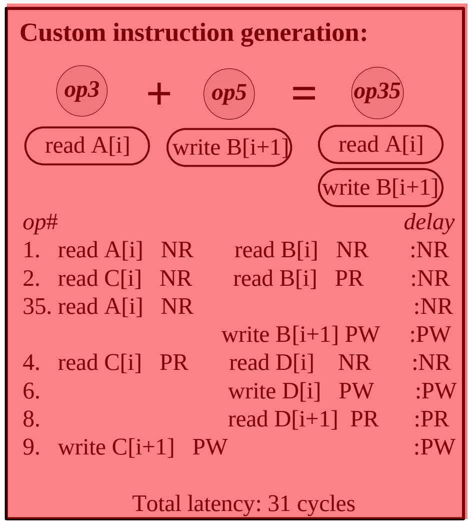


op#			*delay*
1. read A[i] NR	read B[i] NR		:NR
2. read C[i] NR	read B[i] PR		:NR
3. read A[i] NR			:NR
4. read C[i] NR	read D[i] NR		:NR
5.	write B[i+1] NW		:NW
6.	write D[i] NW		:NW
8.	read D[i+1] PR		:PR
9. write C[i+1] PW			:PW

Area: 18.4894 mm^2

Total latency: 41 cycles

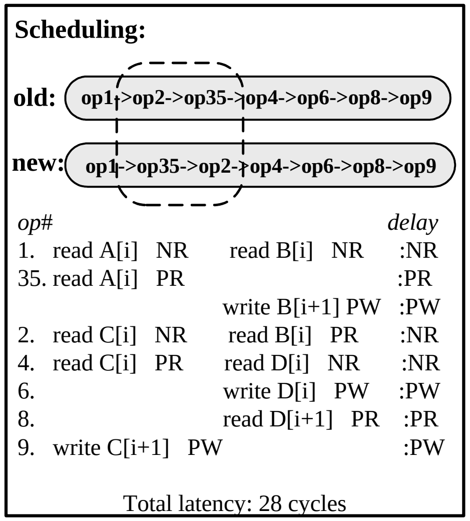
(d) Memory configuration for the operation schedule in (c)



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 (d)


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

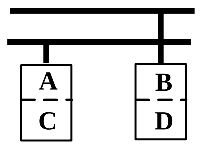
Total latency: 28 cycles

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

Motivational Example

- Scheduling
  - The reason to use the normal mode is that the previous operation uses different arrays, so we can make the same arrays exist in the previous operation
  - For example, when the position of *op*2 and *op*35 is exchanged, the NR will be changed to PR, and then the total latency is reduced
  - It is feasible to reduce the total latencies based on scheduling

Memory configuration:



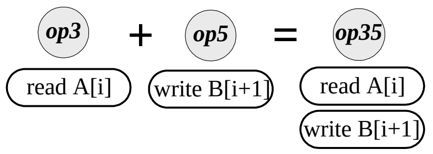
op#			*delay*
1. read A[i]	NR	read B[i] NR	:NR
2. read C[i]	NR	read B[i] PR	:NR
3. read A[i]	NR		:NR
4. read C[i]	NR	read D[i] NR	:NR
5.		write B[i+1] NW	:NW
6.		write D[i] NW	:NW
8.		read D[i+1] PR	:PR
9. write C[i+1]	PW		:PW

Area: 18.4894 mm^2

Total latency: 41 cycles

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

Custom instruction generation:

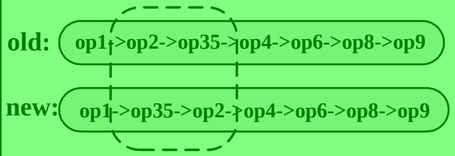


op#			*delay*
1. read A[i]	NR	read B[i] NR	:NR
2. read C[i]	NR	read B[i] PR	:NR
35. read A[i]	NR		:NR
		write B[i+1] PW	:PW
4. read C[i]	PR	read D[i] NR	:NR
6.		write D[i] PW	:PW
8.		read D[i+1] PR	:PR
9. write C[i+1]	PW		:PW

Total latency: 31 cycles

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

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

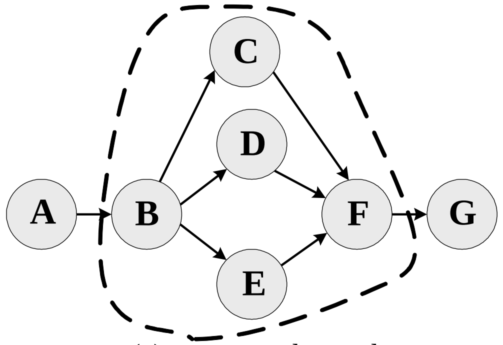
(f) Scheduling results based on the custom instruction in (e)

Contents

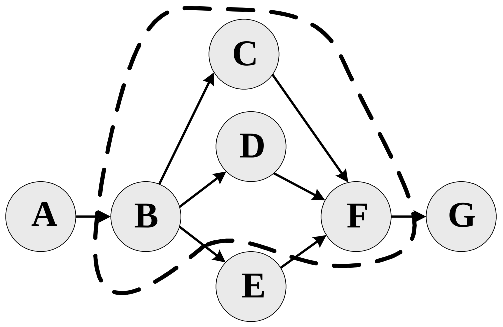
- Introduction
- Motivation
- Problem Formulation
- Methodology
- Conclusion

Definitions

- *G* (*V*, *E*) is a date flow graph (DFG), where *V* and *E* are the sets of nodes and edges. Then the weight of *G* (*V*, *E*) is 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 DFG, then each custom instruction will be mapped to a feasible sub-graph
- *G′* (*V′*, *E′*) be a sub-graph of *G*(*V*, *E*). For ∀*v*1, *v*2∈*V′*, if all the nodes on the paths between *v*1 and *v*2 are contained in *V′*, *G′* is a convex graph; Otherwise, G′ is non-convex
  - If a sub-graph is mapped to a custom instruction, it must be convex. Because non-convex graph can result in non-automated execution for the instructions



(a) convex sub-graph



(b) non-convex sub-graph

Problem

- **Definition:** each custom instruction is corresponding to a feasible sub-graph *G'*, which satisfies two conditions:
  - the number of incoming and outgoing arcs are limited, because each sub-graph is mapped to an instruction, and the operand number is also limited
  - *G*' must be a convex sub-graph
- **Problem:** Given a DFG *G* of high-level code with array access *Arrays*, module library *M* and memory area constraint $Area_{max}$, then generate a scheme of memory allocation *f*, custom instruction set and scheduling *S*, so that the access latency *delay*(*G*) is minimum not exceeding $Area_{max}$

$$Min: \quad \sum_{i=1}^{|V|_{hier=hier_{max}}} delay({}_S^f G'_i)$$

$$s.t. \quad area(f : Arrays \rightarrow M) \leq Area_{max}$$

$$Weight({}_S^f G'_i) \leq weight_{max}$$

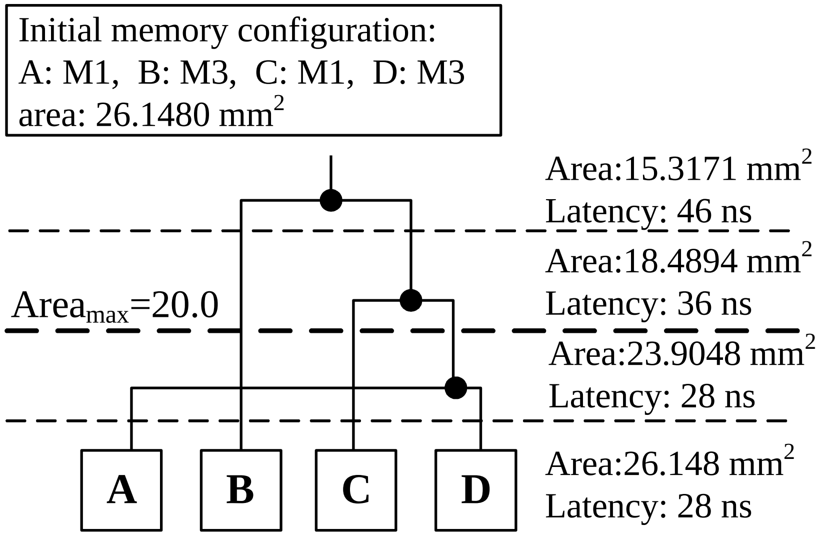
$$1 \leq hier \leq hier_{max}$$

Contents

- Introduction
- Motivation
- Problem Formulation
- Methodology
- Conclusion

(1) Memory Configuration

- Motivation
  - 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, the motivation of this step is to satisfy the memory area constraint first
- Method
  - We can consider to combine each two arrays together from low latency to high latency. Since the area cannot exceed $Area_{max}$, so the result near to the deadline will be selected, and the final result indicates that *A*, *C* and *D* are assigned to the same memory module



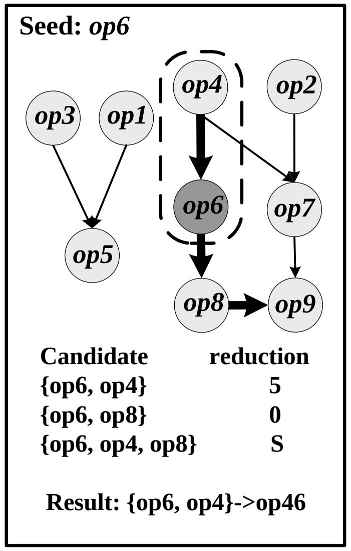
Memory configuration example

(2) Custom Instruction Generation

- Huge search space
  - The basic strategy of the instruction customization is to combine basic operations in the DFG, i.e. sub-graph selection
  - For a DFG *G* with *n* nodes, there will be 2^n candidate sub-graphs
- Customization strategy
  - To avoid searching the useless candidates, we first select some seed nodes under the 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*)
    - *distance* denotes the space between the current node and the longest path. If *distance* is smaller, the current node will hold higher potential to be contained in the candidate sub-graph
    - *priority* is defined as: $priority = in + out + in \times out$, where *in* and *out* mean the numbers of incoming and outgoing arcs. When *priority* is bigger, the potential of selecting the current node as the seed node is higher
    - Suppose that *reduction* represents the quantity of the latency reduction after the current node is combined with the seed node

Seed-growth Algorithm

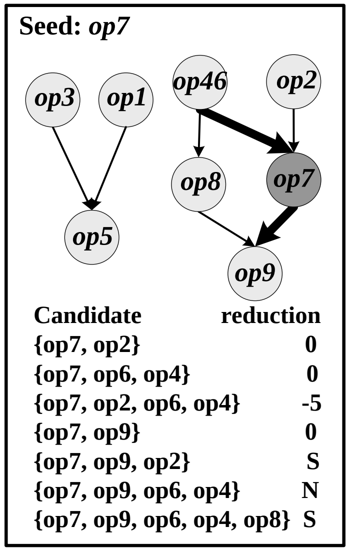
- Guide functions
  - $F_1 = \dfrac{distance+1}{priority}$ $\quad F_2 = \dfrac{reduction}{distance+1}$
  - F_1 is used to select the seed node which is nearer the longest path and hold a higher priority; F_2 is adopted to choose the node which should be combined with the current seed node



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

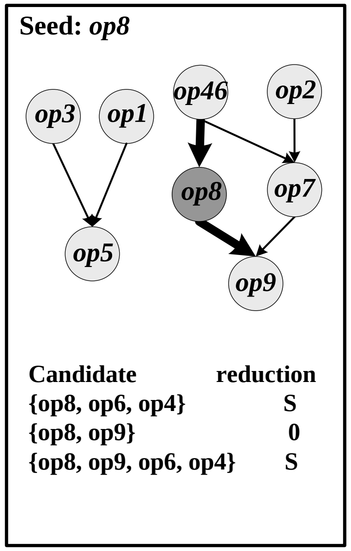
Result: {op6, op4}->op46

(a) seed=*op6*



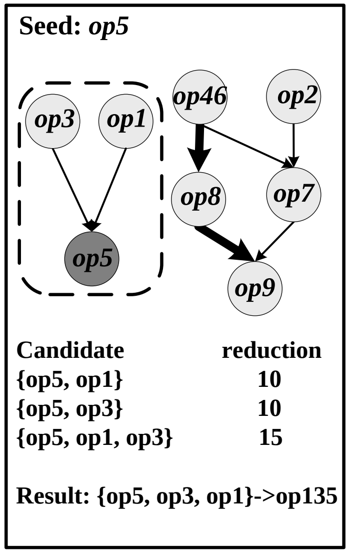
Candidate	reduction
{op7, op2}	0
{op7, op6, op4}	0
{op7, op2, op6, op4}	-5
{op7, op9}	0
{op7, op9, op2}	S
{op7, op9, op6, op4}	N
{op7, op9, op6, op4, op8}	S

(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

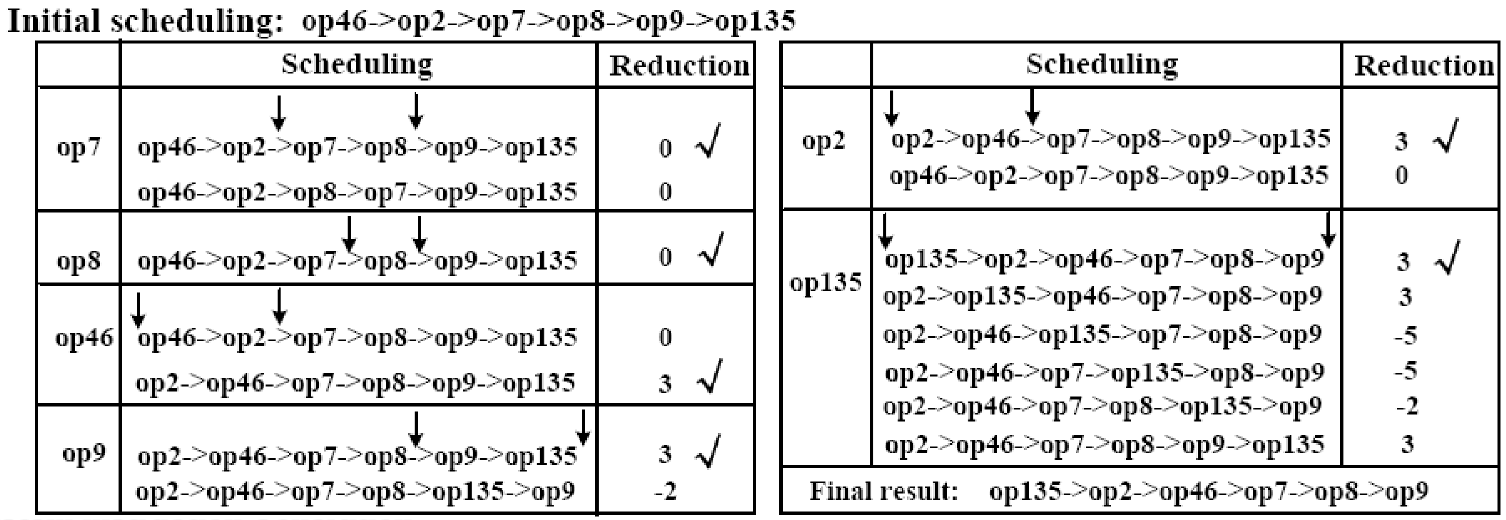
(d) seed=*op5*

(3) Instruction Scheduling

- Constraints
  - Not all the positions can be changed because of the data dependency
- Scheduling strategy
  - Sort the order first from high to low priority (*priority*=*in*+*out*+*in*×*out*)
  - Then each operation *v* will be selected under this priority order and will be moved to find the maximum latency reduction
  - 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*

2=0+2+0*2

1=0+1+0*1



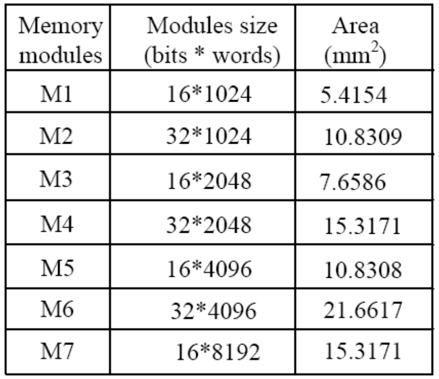
Initial scheduling: op46->op2->op7->op8->op9->op135

	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	

Experiments

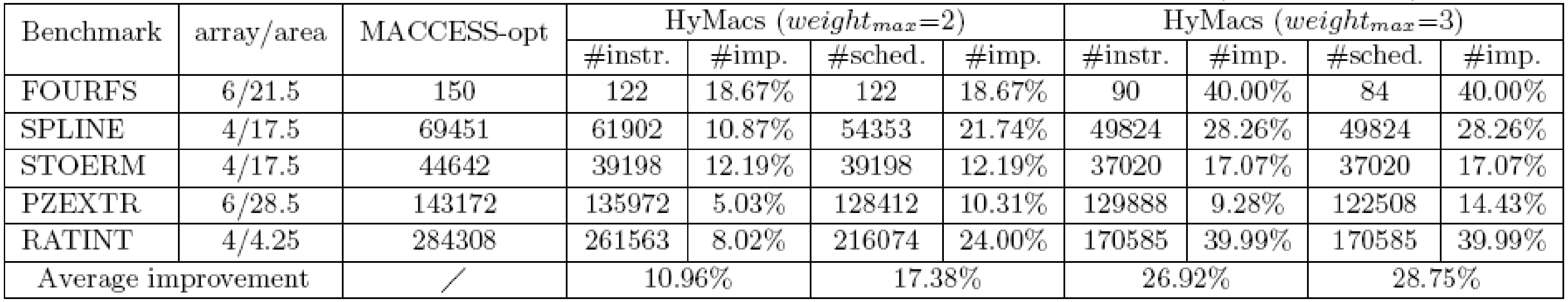
- Focus: memory access latency
  - Input: a set of benchmarks in numerical recipes
  - Output: access cycles based on the library in [3]



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

Memory module library in [3]

- Experimental results
  - The third column shows the latency results using the previous system MACCESS-opt in [3] for compare
  - HyMacs can achieve about 20% improvements than the system MACCESS-opt in [3], where custom instructions and scheduling contribute about 15% and 5% respectively



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%	

Contents

- Introduction
- Motivation
- Problem Formulation
- Methodology
- Conclusion

Conclusion

- Our contributions
  - We propose a hybrid memory access system to reduce the whole memory access latency which integrates the custom instruction generation and scheduling algorithm
  - By applying a hardware/software co-design strategy, the hybrid system can obtain an improvement on the access latency reduction than the previous method which only considers the software optimization

Thanks