LOOP SCHEDULING OPTIMIZATION FOR CHIP-MULTIPROCESSORS WITH NON-VOLATILE MAIN MEMORY

Yan Wang[2], Jiayi Du[2], Jingtong Hu[3], Qingfeng Zhuge[1], and Edwin H.-M. Sha[1,3]

[1]College of Computer Science, Chongqing University, Chongqing, China
[2]College of Information Science and Engineering, Hunan University, Hunan, China
[3]Department of Computer Science, University of Texas at Dallas, TX, 75082, USA

ABSTRACT

Non-Volatile Memories (NVMs) have many advantages over traditional DRAM. It is desirable to apply NVM as main memory in embedded Chip Multi-Processor (CMP) systems. However, NVMs have drawbacks that need to be overcome. That is, a write to the NVMs is expensive. Loops are the most critical and time-consuming part in digital signal processing (DSP) applications. However, loops are difficult to parallelize on multi-processor systems due to the inter-iteration dependencies. This paper targets on embedded CMP systems and proposes techniques to improve loop parallelism while considering reducing the write activities to the NVMs when they are used as main memory. The experimental results show that the proposed algorithm can reduce the number of write activities on NVM by 21.1% on average. In other words, the average lifetime of NVM can be extended to at least 2 times longer than before and the total schedule length is reduced by 19.6% on average.

1. INTRODUCTION

Chip multi-processors (CMP) have been the *de facto* design for modern high-performance DSP processors. Non-Volatile Memories (NVM), including Phase Change Memory (PCM), Flash Memory, and Magnetic RAM (MRAM) have many advantages over DRAM while they are applied as main memory for embedded systems due to their attractive characteristics such as power-economy, low-cost, high density, non-volatility, and shock-resistivity.

However, the NVMs have two drawbacks which need to be overcome when used as main memory. 1) write activities take longer time than read activities on NVM. 2) NVMs have a maximum number of write operations that can be performed. In this paper, we propose a loop scheduling algorithm to improve the parallelism and reduce the schedule length while considering reducing the number of write activities on non-volatile main memory.

In DSP applications, "loops" take the longest computation time and consume the most energy. Many loop scheduling techniques were proposed to reduce the schedule length of loops on multi-core processors, such as retiming, rotation scheduling, etc. [2, 3, 6]. Retiming technique decreases the scheduling length for loops by evenly distributing the delays [2]. Rotation technique is a iterative pipelining technique based on retiming to obtain a compact loop scheduling under resource constraint. The above techniques work well when DRAM is used as main memory. However, when NVM is used as main memory in a multi-core system, traditional rotation algorithm may cause degraded performance. This is because that in each iteration of the rotation technique, the write activities in the main memory may increase. Even though a short scheduling length can be obtained through rotation, the total running time might degrade due to the increased write activities to the non-volatile main memory. Therefore, it is crucial to develop novel loop scheduling techniques that can minimize loop schedule length as well as reduce the number of write activities on NVM.

Many researchers have been addressing the problem of expensive write activities in order to extend the lifetime of NVM when they are applied as main memory. Hu et al. [4, 5] introduce data migration and recomputation techniques to reduce write activities to NVM. These two techniques can greatly reduce the write activities from software part. However, in their paper, they only consider the applications without loops. Furthermore, they did not propose any scheduling algorithm. In this paper, we propose scheduling algorithms for loops while reducing the write activities to the non-volatile memory. The data migration and recomputation techniques can also be combined with the scheduling algorithm proposed in this paper.

In this paper, we target CMPs with a Scratch Pad Memory (SPM) as its on-chip memory and NVM as its main memory. We propose the Rotation using Maximum Bipartite Match (RMBM) algorithm that improves the loop parallelism, decreases the schedule length, and reduces the number of write activities on non-volatile main memory. In traditional rotation scheduling algorithm, in each rotation phase, multiple computation tasks will be reassigned to new cores. During reassignment of computation tasks, the traditional rotation scheduling assigns cores to tasks randomly. In RMBM, maximum bipartite matching method is proposed to obtain the best reassignment with a minimum number of write activities while minimizing schedule length of loops. Furthermore, data migration and data recomputation are combined into RMBM in order to minimize the write activities.

The main contributions of our work include:

- We model tasks assignment problem in each rotation phase as a maximum bipartite matching problem to reduce the write activities.
- We combine data migration and data recomputation into rotation technique to minimize the scheduling length of loops and to reduce the number of write activities on NVM when it is applied as main memory.

The remainder of this paper is organized as follows. Section 2

This work is partially supported by NSF CNS-1015802, Texas NHARP 009741-0020-2009, NSFC 61173014, NSFC 61133005, China Thousand-Talent Program.

Table 1. Initial schedule.

1. Initial Schedule. (4520 clock cycles and 7 write activities)																
Steps		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Core 1	OP	Load IW1	Load AR1	Task A	**WRITE AW1**	Load AW1	Task C	**WRITE CW1**								
	SPM1	IW1	IW1	AW1		AW1	CW1									
	SPM2		AR1	AR1	AR1	AR1	AR1	AR1								
Core 2	OP				Load AW1	Load DR1	Task D		Load EW1	TASK G	LOAD CW1	TASK I	LOAD HW1	TASK J	WRITE IW1	WRITE JW1
	SPM1				AW1	AW1	DW1	DW1	EW1	GW1	CW1	IW1	IW1	IW1		
	SPM2					DR1	DR1	DR1	EW1	EW1	EW1	CW1	CW1	HW1	JW1	
Core 3	OP	Load JW1	LOAD BR1	TASK B	WRITE BW1	LOAD BW1	LOAD ER1	TASK E	WRITE EW1	LOAD FW1	TASK H	WRITE HW1				
	SPM1	JW1	JW1	BW1		BW1	BW1	EW1		FW1	HW1					
	SPM2		BR1				ER1	EW1	EW2	EW2	EW2	EW2 EW2				
Core 4	OP					LOAD BW1		TASK F	WRITE FW1							
	SPM1					BW1	BW1	FW1								
	SPM2															

presents the hardware and software model. A motivational example is also presented. Section 3 proposes the RMBM algorithm. The experimental results are shown in Section 4 and finally this paper is concluded in Section 5.

2. MODELS AND EXAMPLE

In this section, we will first introduce the architectural model and software model. Then, a motivational example is presented to illustrate the motivation.

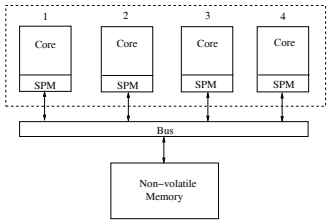

Fig. 1. Example system with four cores.

We target embedded Chip Multiprocessors with Scratch Pad Memory (SPM) as its on-chip memory and NVM as its main memory as shown in Figure 1. In this architecture, each core is equipped with a SPM. As shown in Figure 1, all cores are connected to NVM via a bus. SPM is small on-chip memory component that is managed by software, either by application program or through automated compiler support. Nowadays, many DSP processors are employing SPM rather than cache as their on-chip memories. Examples of CMPs employing SPM include TI OMAP4470 [1].

In this paper, we model loops with Data Flow Graphs (DFGs). Formally, the input we consider in this paper is a DFG $G = \langle V, E, P, R, W, t, d \rangle$. $V = \{v_1, v_2, v_3, \ldots, v_n\}$ is the set of n tasks. $E \subseteq V \times V$ is the set of edges where $(u, v) \in E$ means that task u must be scheduled before task v. $P = \{p_1, p_2, p_3, \ldots, p_m\}$ is the set of m pages that are accessed by the tasks. $R : V \rightarrow P^*$ is the function where $R(v)$ is the set of pages that task v reads from. $W : V \rightarrow P^*$ is the function where $W(v)$ is the set of pages that task v writes to. $t(v)$ represents the computation time of task v while all the required data is in the cache. d is a function from E to the nonnegative integers, and $d(e)$ of an edge e equals the number of delays (delay count) on edge e. The number of delays stands for the number of previous iterations task v depends on u. The output is a schedule of tasks, SPM data block replacement, and non-volatile memory read and write operations.

After presenting the hardware and software model, an example will be presented to illustrate that different task scheduling has different write activities and execution time. Assume there are four cores in the example system as shown in Figure 1. Each core is equipped with a SPM, and each SPM can hold two data. In the motivational example, we assume that reading a data from NVM takes 80 cycles; writing a data to NVM takes 800 cycles; a core accessing its own SPM takes 2 cycles; and a core accessing data from other core's SPM takes 5 cycles. We assume each task takes 10 cycles to finish in this example. The input of this example is a DFG as shown in Figure 2. There are 10 tasks in the input graph. Each task has a read set and a write set.

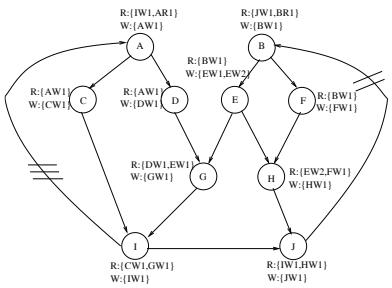

Fig. 2. Example input graph.

In the initial schedule, we schedule the example DFG to run on the example system with list scheduling. Under list scheduling, task A and C are assigned in core 1. Task D, H, I, and J are assigned in core 2. Task B and E are assigned in core 3. Task F is assigned in core 4. A complete view of the execution of input task graphs with memory accesses is presented in Table 1. In Table 1, the second row shows step number. In each core, the first row shows the instructions that are executed. The second row shows the content of the first SPM block at each step and the third row shows the content of the second block at each step. In the initial schedule, to finish these tasks in a iteration needs 4520 clock cycles, and there are 7 write activities to the NVM.

Rotation can be used to reduce the task scheduling length. We notice that task C and D in current iteration are independent of task A in the next iteration. Task E and F in current iteration are independent of task B in the next iteration. Therefore, we can execute task A and B in the first iteration by themselves. Then, for each of the following iteration, we can move task A and B to previous iteration to form a new iteration body. Task A and B are both independent of all other tasks in this new iteration body. In this new iteration body, task A and B can be scheduled at any step as long as the processor core is available. By doing so, the cores can be utilized more effectively and task scheduling length can be reduced. There are many available positions that A and B can be assigned to. Different assignments will generate different number of write activities to the NVM

Table 2. Rotation scheduling with the random reassignment.

2. Schedule after rotation using the random reassignment. (2000 clock cycles and 3 write activities)													
Steps		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Core 1		Load AW1	Task C	**write CW1**	Load IW1$_r$	Load AR1$_r$	Task A$_r$						
	SPM1	AW1	AW1	AW1	IW1$_r$	IW1$_r$	AW1$_r$						
	SPM2		CW1		AR1$_r$	AR1$_r$							
Core 2		Load AW1	Load DR1	Task D	**Load EW1**	Task G	**Load AW1**	Task C	Task I	**Load HW1**	Task J	**Write I**	**Write J**
	SPM1	AW1	AW1	DW1	DW1	GW1	GW1	GW1	IW1	IW1	IW1		
	SPM2		DR1	DR1	EW1	EW1	AW1	EW1	CW1	HW1	JW1	JW1	
Core 3		Load BW1	Load ER1	Task E	**Migrate EW1**	**Load BW1**	Task F	Task H		**Migrate HW1**			
	SPM1	BW1	BW1	EW1		BW1	FW1	HW1	HW1	HW1			
	SPM2		ER1	EW2	EW2	EW2	EW2	EW2	EW2	EW2			
Core 4		Load BW1	Load FR1	Task F	**Discard FW1**	Load JW1$_r$	Load BR1$_r$	Task B$_r$					
	SPM1	BW1	BW1	FW1		JW1$_r$	JW1$_r$	BW1$_r$					
	SPM2						BR1$_r$	BR1$_r$					

Table 3. Rotation scheduling with bipartite matching algorithm.

3. Schedule after first rotation using the bipartite matching algorithm. (1815 clock cycles and 2 write activities)												
Steps		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Core 1		Load AW1	Task C					Load IW1$_r$	Load AR1$_r$	Task A$_r$		
	SPM1	AW1	CW1	CW1	CW1	CW1	CW1	IW1$_r$	IW1$_r$	AW1$_r$		
	SPM2				FW1	FW1	FW1	FW1	AR1$_r$	AR1$_r$		
Core 2		Load AW1	Load DR1	Task D	**Load EW1**	Task G	**Load CW1**	Task I	**Load HW1**	Task J	**Write I**	**Write J**
	SPM1	AW1	AW1	DW1	DW1	GW1	GW1	IW1	IW1	IW1		
	SPM2		DR1	DR1	EW1	EW1	CW1	HW1	HW1	JW1	JW1	
Core 3		Load BW1	Load ER1	Task E	**Migrate EW1**	**Load FW1**	Task H		**Migrate HW1**			
	SPM1	BW1	BW1	EW1		FW1	HW1	HW1	HW1			
	SPM2		ER1	EW2	EW2	EW2	EW2	EW2	EW2			
Core 4		Load BW1	Load FR1	Task F	Load JW1$_r$	Load BR1$_r$	Task B$_r$					
	SPM1	BW1	BW1	FW1		JW1$_r$	JW1$_r$	BW1$_r$				
	SPM2						BR1$_r$	BR1$_r$				

and therefore affect the total execution time for each iteration.

Traditional rotation will assign task A and B randomly. One possible assignment for A and B is that scheduling task A on core 1 at step 6 and scheduling task B on core 4 at control step 7. The full schedule with all memory access information is shown in Table 2. In this schedule, executing a loop iteration needs 3 write activities and takes 2000 cycles.

Another better schedule can be obtained by assigning task A on core 1 at step 9 and assigning task B on core 4 at step 6 as shown in Table 3. The full schedule with all memory access information is shown in Table 3. In this schedule, finishing a loop iteration only needs 2 write activities and takes 1815 cycles. Compared to the traditional rotation scheduling, 1 write activity is eliminated and the execution time of finishing a loop is reduced by 10%. Compared with list scheduling, 5 write activities to NVM are eliminated compared with the list scheduling and the execution time is reduced by 59.84%.

3. ALGORITHMS

In this section, we present the details of the proposed Rotation using Maximum Bipartite Match (RMBM) algorithm. The RMBM algorithm is based on the retiming technique [6] and rotation scheduling [3].

3.1. The Rotation using Maximum Bipartite Match Algorithms

In this subsection, we will present the details of the Rotation using Maximum Bipartite Match (RMBM) algorithm. The problem with the traditional rotation scheduling is that during each phase of rotation, the rotated tasks are reassigned to processor cores randomly. It only considers the task scheduling length. However, in actual execution, other aspects like the off-chip memory accesses are equally important in reducing the total execution time. In the RMBM algorithm we devised a novel strategy to reassign the rotated tasks to the cores.

Algorithm 3.1 Rotation using Maximum Bipartite Matching Algorithm (RMBM)

Input: A DFG graph of tasks and number of cores.
Output: New schedule with fewer cycles, and fewer write activities on non-volatile memory.

```
repeat
    Find X ← rotatable tasks.
    Retiming on the previous DFG graph
    Change the control step (core_ij)
    L ← the location that could be assigned to a new task
    for Each(task ∈ X, core_ij ∈ L) do
        w ← rotation_benifit(task, core_ij)
    end for
    Use maximum bipartite match algorithm to obtain a new assign, in which ∑ w is maximum.
    for Each node do
        Find the migration path and compute cycles of in the migration path
        Compute cycles of using recomputation
        Compare cycles of migration, recomputation and write
        Choose the method that produces a shorter cycles schedule
    end for
until The schedule length is minimum.
```

Before presenting the details of the RMBM algorithm, we define a benefit function $rotation\_benefit$. $rotation\_benefit$ is a function of the tasks and the processor cores. For example, when the task A is assigned to core 1 at step 1, we use $rotation\_benefit(A, core\_11)$ to denote the total benefits of this assignment. $rotation\_benefit$ equals to $earn$ subtracting $consum$. When node A can use the previous node's data in the assigned core,

Table 4. Experimental results.

Bench.	task number	control step		write number			read number			cycle		
		List	after rotation	ASAP algorithm	RMBM algorithms	%W-L	ASAP algorithm	RMBM algorithms	%W-R	ASAP algorithm	RMBM algorithms	%W-C
IIR	8	4	3	3	2	33.3	7	6	14.3	1855	1050	43.3
2IIR	16	6	4	11	9	18.2	21	17	19.2	4345	3763	13.4
8_lattice	42	21	11	24	19	20.9	25	19	24	8560	7879	8.1
diff2	10	6	4	2	1	50	2	1	50	1710	835	51.2
deq	11	5	3	3	2	33.3	10	10	0	2625	1865	28.9
allpole	15	12	6	3	2	33.3	6	4	33.3	1965	1670	15.1
4_lattic	26	11	7	17	15	11.7	29	24	17.2	6330	5450	14.0
voltera	27	10	8	14	13	7.1	28	25	10.7	7790	6980	10.4
ellip	34	14	10	15	14	6.6	25	19	24	8260	7479	9.5
4latirr	52	15	13	29	26	10.3	46	45	2.0	9225	8425	8.9
2-4latirr	52	15	13	27	25	7.4	28	20	28.6	9005	7890	12.4
Average Improvement				write number		21.1	read number		20.3	cycle		19.6

$earn$ is the benefit of the cycles we can save compared to the cost before rotation. $consume$ includes two cases. The first case is the cost of obtaining data from other cores while originally it can use the data from its own SPM. The second case is the additional cost of delivering the data to its children nodes while originally the children nodes can use the data directly from the SPM.

The Rotation Using Maximum Bipartite Match (RMBM) Algorithm is shown in Algorithm 3.1. The main idea of RMBM Algorithm is that during each phase of rotation, rather than assigning rotated tasks into random cores, we construct a bipartite graph with the rotated tasks and cores. Then we compute $rotation\_benefit$ between each pair of task and core. Then, using maximum bipartite algorithm to reassign these tasks to obtain an schedule that total $rotation\_benefit$ is maximum.

At the end of each phase of rotation, we construct a bipartite graph $M =< V_m, E_m, w >$ as follows. Let X be the set of cores and L be the set of rotated tasks. For each element in X and L, we add a vertex into V_m. For each pair of elements from X and L, we add an edge into E_m. For each edge, we compute the $rotation\_benefit$ associated with it. w is the set of $rotation_benefit$s. After the construction of bipartite graph M, we find a maximum bipartite matching in M. According to the maximum bipartite matching, we will assign the tasks to the corresponding cores in the matching. By doing this, we can reduce the total time needed. Since write activities to the non-volatile memory is the most time consuming part in this architecture, during the minimization of total execution time, the number of write activities to the non-volatile main memory is also reduced.

In this paper, we use Ford-Fulkerson method to find a maximum matching in M. The time complexity of Ford-Fulkerson method is $O(XL)$. retiming one time for the set X takes $O(X)$ time. Then, the time complexity of rotation a time using maximum bipartite basic retiming is $O(XL + X)$. Therefore, the time complexity of RMBM Algorithm is $O(VE + V)$.

4. EXPERIMENTS

In this section, we present the experimental results. The effectiveness of the RMBM algorithm is evaluated by running the DSPStone benchmarks [8]. The DFGs are all extracted from gcc compiler and then the DFGs are fed into a custom simulator.

In the set of experiments, there are 4 cores in the system and each core has a SPM with capacity of 16KB. Table 4 shows the results of the RMBM algorithm compared with the ASAP algorithm [7], which is widely applied on loop scheduling to minimize the schedule length of loops. The first column shows the benchmarks' names. The second column shows number of tasks of each benchmark. The third and fourth columns show number of control steps of initial schedule and rotation scheduling, respectively. The fifth to seventh columns show results of number of write activities to NVM of RMBM algorithm compared with ASAP algorithm. The eighth to tenth columns and the eleventh to thirteenth columns show results of number of read activities from NVM and execution time to finish one loop iteration of RMBM algorithm compared with ASAP algorithm, respectively. We can see from Table 4 that on average, the RMBM algorithm can reduce the execution time to finish a loop interation by 19.6%. Compared with ASAP algorithm, the RMBM algorithm reduces the number of read activities from NVM of each loop by 21.1% on average.

5. CONCLUSION

In this paper, we propose the RMBM algorithm, which can significantly reduce the programs's schedule time and extend the lifetime of NVM at the same time. The experimental results show that the proposed algorithm can reduce number of write activities to NVM by 21.1% on average. At the same time, the execution time is reduced by 19.6% on average.

6. REFERENCES

[1] http://www.engadget.com/2011/06/02/texas-instruments-announces-multi-core-1-8ghz-omap4470-arm-proc/,2011

[2] Qingfeng Zhuge, Chun Xue, Zili Shao, Meilin Liu, Meikang Qiu and Edwin H.-M. Sha. Design Optimization and Space Minimization Considering Timing and Code Size via Retiming and Unfolding. In Microprocessors and Microsystems, 30(4), pp. 173-183, 2006.

[3] Liang-Fang Chao and A.S. Lapaugh and Edwin H.-M. Sha. Rotation scheduling: a loop pipelining algorithm. IEEE TCAD. 16(3), pp. 229-239, 1997.

[4] J. Hu, C. J. Xue and W. C. Tseng et al. Reducing Write Activities on Non-volatile Memories in Embedded CMPs via Data Migration and Recomputation. DAC'10, pp.350 - 355, 2010.

[5] J. Hu, C. J. Xue and W. C. Tseng et al. Minimizing Write Activities to Non-volatile Memory via Scheduling and Recomputation. SASP'10, pp.101 - 106, 2010.

[6] Qingfeng Zhuge, Bin Xiao, and Edwin H-M. Sha. Code Size Reduction Technique and Implementation for Software-Pipelined DSP Application. ACM TECS, Vol.2, No.4, pp. 590-613. 2003

[7] Cathy Qun Xu, Chun Jason Xue, Jingtong Hu, and Edwin H.-M. Sha. Optimizing Scheduling and Intercluster Connection for Application-Specific DSP Processors. In IEEE TSP, pp.4538 - 4547, 2009

[8] V. Zivojnovic. J. Martinez, C. Schlager, and H.Meyr, Dspstone: A D-sporiented Benchmarking Methodology. In Proceedings of the International Conference on Signal Processing Applications and Technology. Dallas. Texas. USA, 1994