Minimizing Leakage Energy with Modulo Scheduling for VLIW DSP Processors

Meng Wang, Zili Shao, Hui Liu, and Chun Jason Xue

Abstract As technology scaling approaches to the nanometer, leakage power has become a significant component of the total power consumption. In this paper, we develop a novel leakage-aware modulo scheduling algorithm to achieve leakage energy savings for DSP applications with loops on VLIW architecture. The proposed algorithm is designed to maximize the idleness of function units integrating with leakage management scheme [9], and reduce the number of transitions between active and sleep modes. We have implemented our technique into the Trimaran compiler [1] and conducted experiments using a set of benchmarks from DSPstone [11] and Mibench [7] on the VLIW simulator of Trimaran. The results show that our algorithm achieves significant leakage energy savings compared with the leakage-aware scheduling algorithm [8].

1 Introduction

As technology feature size continues to shrink, leakage power is becoming comparable to dynamic power in the current generation of technology [3, 6, 10], and it will further dominate the overall energy consumption in future technologies [4]. High performance DSP (Digital Signal Processing) needs to be performed not only with high data throughput but also with low power consumption in embedded sys-

Meng Wang · Zili Shao
Department of Computing, The Hong Kong Polytechnic University, Hong Kong
e-mail: csmewang, cszlshao@comp.polyu.edu.hk

Hui Liu
Software Engineering Institute, Xidian University, Xi'an, China
e-mail: liuhui@xidian.edu.cn

Chun Jason Xue
Department of Computer Science, City University of Hong Kong, Hong Kong
e-mail: jasonxue@cityu.edu.hk

Please use the following format when citing this chapter:

Wang, M., et al., 2008, in IFIP International Federation for Information Processing, Volume 271; *Distributed Embedded Systems: Design, Middleware and Resources*; Bernd Kleinjohann, Lisa Kleinjohann, Wayne Wolf; (Boston: Springer), pp. 111–120.

tems. VLIW (Very Long Instruction Word) architecture that has multiple functional units (FUs) and can process several instructions simultaneously is widely adopted in high-end DSP. While this multiple-FUs architecture can be exploited to increase instruction-level parallelism and improve time performance, it causes more leakage power consumption. Therefore, it becomes an important problem to reduce the leakage energy of a DSP application on VLIW architecture. Since loops are usually the most critical parts in a DSP application, we develop a loop scheduling technique to reduce the leakage energy of an application on VLIW architecture.

A lot of research efforts have been put to characterize cost models for analyzing static power [3] and evaluate techniques for leakage power reduction [9]. The architecture level model in [3] confirms that the functional units contribute to a noticeable fraction of leakage power despite having relatively fewer transistors compared to caches. A hardware based leakage energy management scheme is proposed in [9] for short idle periods. In their scheme, the dual-threshold domino logic with sleep mode that can transit between active mode and sleep mode is utilized.

Many techniques [8, 5] are proposed to reduce leakage energy consumption of functional units for VLIW architecture. Nagpal et al. [8] proposed a leakage-aware instruction scheduling algorithm for VLIW and clustered VLIW architectures to reduce leakage energy by exploiting the scheduling slacks of instructions. In most of the above work, the instruction scheduling for reducing leakage power is based on DAG (Directed Acyclic Graph) scheduling in which only intra-iteration dependencies are considered. In this paper, we show that we can significantly improve the leakage energy by carefully exploiting inter-iteration dependencies.

In this paper, we propose a leakage-aware modulo scheduling algorithm that assists the hardware based leakage energy management scheme [9] to achieve leakage savings for DSP applications in the context of VLIW architecture. Our basic idea is to schedule nodes into better locations in order to maximize the idleness of function units integrating with leakage reduction control, and reduce the number of transitions between active and sleep modes. We implement our technique into the Trimaran compiler [1] and conduct experiments on a set of benchmarks from DSPstone [11] and Mibench [7] based on the power model in [9]. The results show that our algorithm achieves significant leakage energy savings compared with the leakage-aware scheduling algorithm [8]. On average, our technique contributes to 14.73% reduction in the leakage energy consumption with only 1.74% decrement in the performance.

The rest of the paper is organized as follows. Motivational examples are shown in Section 2. The basic concepts are introduced in Section 3. The scheduling algorithm is proposed in Section 4. The experimental results and analysis are provided in Section 5, and the conclusion is given in Section 6.

2 Motivational Examples

In order to show how our approach works, we present an example in this section. We use Trimaran compiler [1] to generate the Data Flow Graph for this example and perform modulo scheduling on it. We compare the scheduling generated by the traditional modulo scheduling and our technique. The energy model to calculate the leakage energy is introduced in Section 3.2.

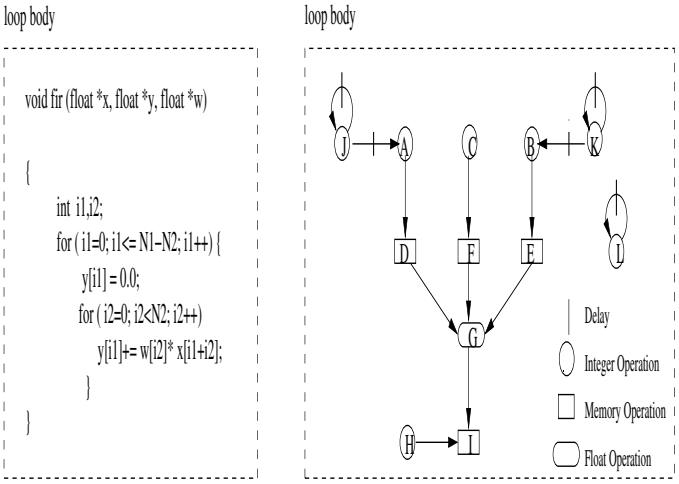

Fig. 1 The FIR program and its corresponding data flow graph.

A real DSP application, the FIR program and the Data Flow Graph of the innermost loop is shown in Figure 1. In the graph, each node denotes a computational task in the loop, and there are 7 integer ALU operations, 4 memory operations, 1 floating ALU operation and 1 branch for this particular example. The edge without delay represents the intra-iteration data dependency (e.g. $A \rightarrow D$), and the edge with delays represents the inter-iteration data dependency (e.g. $J \rightarrow A$ has an edge with one delay which is denoted by one bar), in which the number of delays represents the number of iterations involved.

Assume that we want to schedule the graph in Figure 1 to the VLIW architecture with 7 FUs which are fully pipelined. And let $FU1$ and $FU2$ be integer ALUs, $FU3$ and $FU4$ be floating-point ALUs, $FU5$ and $FU6$ be memory units and $FU7$ be the branch unit. Note that we assume the integer operations A, B, C, H, J, K, L take 1 cycle for execution, the load operations D, E, F take 2 cycles, the store operation I takes 1 cycle, and the floating multiplication node G takes 2 cycles to finish execution in this example.

The schedule generated by the traditional modulo scheduling is shown in Figure 2. Based on the power model in [9], for integer ALUs which are most heavily utilized, the dual-threshold domino logic with sleep mode can transit between ac-

Time	IALU		FALU		Memory Units		BRANCH
	FU1	FU2	FU3	FU4	FU5	FU6	FU7
0	A	B			I		
1	C	H			D	E	
2	J	K			F		
3	L						
4			G				
5							Branch

Fig. 2 The schedule generated by performance-oriented modulo scheduling.

Time	IALU		FALU		Memory Units		BRANCH
	FU1	FU2	FU3	FU4	FU5	FU6	FU7
0	A				I		
1	B						
2	C				D	E	
3	H				F		
4	J						
5	K		G				
6	L						Branch

Fig. 3 The schedule generated by our technique.

tive mode and sleep mode after one cycle of idleness. The circuit expends very little leakage energy in the sleep mode. However, the energy savings of this schedule are severely affected by frequent transitions from active mode to sleep mode and vice-versa because of many short idle periods.

The schedule generated by our technique is shown in Figure 3. For this example, the schedule generated by our technique has little performance loss than the traditional modulo scheduling algorithm. In this schedule, $FU2$ is totally unused, so we can put $FU2$ into the sleep mode before entering the loop body. Thus, our technique achieves big leakage savings compared with the performance-oriented modulo scheduling.

3 Basic Concepts

3.1 Modulo Scheduling Overview

The objective of modulo scheduling [2] is to engineer a schedule for one iteration of the loop such that when this same schedule is repeated at regular intervals, no intra- or inter-iteration dependence is violated, and no resource usage conflict arises between operations of either the same or distinct iterations. This constant interval between the start of successive iterations is termed the initiation interval (II). The repetitive portion can be re-rolled to yield a new loop which is termed the kernel.

The prologue is the code that precedes the repetitive part and the epilogue is the code following the repetitive part. The minimum initiation interval (MII) is a lower bound on the smallest possible value of II for which a modulo schedule exists. The MII must be equal to or greater than both the resource-constrained MII (ResMII) and the recurrence-constrained MII (RecMII). The candidate II is initially set to the MII and increased until a legal modulo schedule is found.

3.2 Energy Model

The energy model used in this paper is based on [9]. The total energy in a functional unit in this model is determined as follows:

$E_{total} = Dyn\_Energy + Leak\_Energy + Trans\_Energy + Sleep\_Energy$

$E_{total} = n_A\ (\alpha E_A + (1\text{-}D) E_{S_1}) + (n_A D + n_{UI})(\alpha E_{S_0} + (1\text{-}\alpha) E_{S_1}) + M_Z((1\text{-}\alpha) E_A + E_{Sleep}) + n_Z E_{S_0}$

Here, n_A is the number of active cycles, n_{UI} is the number of uncontrolled idle cycles, n_Z is the number of sleep mode cycles and M_Z is the number of transitions between different modes. E_{S_0} and E_{S_1} are low and high leakage energy consumption, respectively.

4 Leakage-Aware Modulo Scheduling Algorithm

In this section, we first propose the leakage-aware modulo scheduling algorithm in Sections 4.1. Then we discuss its key functions in Section 4.2.

4.1 The Proposed Algorithm

In the proposed algorithm, our basic idea is to schedule nodes of a loop to better locations in order to enlarge the idleness in FUs which can be exploited to apply leakage energy control mechanism. In most of the previous work, loop is modeled as the DAG part of a DFG in which only intra-iteration dependencies are considered. As shown in Section 2, by exploring inter-iteration dependencies, we can get more opportunities to schedule nodes of DFG to better locations in a schedule to achieve more leakage energy saving. The leakage-aware modulo scheduling algorithm is shown in Algorithm 4.1.

In the algorithm, G is the data flow graph of the loop, TC is the timing constraint and the $BudgetRatio$ is the ratio of the maximum number of operation scheduling steps attempted before giving up to the number of operations in the loop. This parameter determines how hard the function IterativeSchedule() tries to find a legal

Algorithm 4.1 The leakage-aware modulo scheduling algorithm.

Require: The data flow graph G=⟨V, E, d, t⟩, the timing constraint TC, BudgetRatio.
Ensure: A schedule S with minimum leakage energy Min_Energy.

```
    // Initialize the value of II to the Minimum Initiation Interval
 1: II := MII();
 2: Min_Energy ← ∞;
 3: while II < TC do
 4:     Budget := BudgetRatio * NumberofOperations;
 5:     while IterativeSchedule( II, Budget)!= SUCCESS do
 6:         II := II+1;
            // Calculate the leakage energy based on the power model [9]
 7:         S' := The generated legal schedule;
 8:         E_S' := CalculateEnergy(S');
 9:         if Min_Energy > E_S' then
10:             S ← S' and Min_Energy ← E_S'
11:         end if
12:     end while
13: end while
```

schedule for a candidate II before giving up. The output of the algorithm is a legal schedule S with minimum leakage energy Min_Energy.

In the algorithm, we first initialize the value of II to the minimum initiation interval. Then, function IterativeSchedule() is used to perform the actual scheduling as shown in Algorithm 4.2. After all operations have been scheduled and a legal schedule is generated, we record the energy of it and compare it with Min_Energy. The algorithm terminates when the timing constraint is achieved.

4.2 Function IterativeSchedule()

In function IterativeSchedule(), we first calculate the priority for each operation based on the height-based priority function ComputePriority() of modulo scheduling, and pick up the operation with highest priority to be scheduled. In function ComputePriority(), we calculate the longest path from the node to the end of the data flow graph. This function gives higher priority to operations on the critical path in order to achieve a good schedule.

Then, the schedule time bounds for the current operation are calculated according to the data dependence constraint. We use function FindTimeSlot() to find a legal time slot for the current operation within the range ($MinTime$, $MaxTime$). $MinTime$ is the earliest start time for an operation as constrained by its dependences on its predecessors. $MaxTime$ equals to $MinTime + II - 1$ since each iteration in modulo scheduling begins exactly II cycles after the previous one.

In function FindTimeSlot(), the goal is to find an empty block to put the operation $CurrOper$ in. We first calculate the start time and the end time of each empty block in each functional unit.

Algorithm 4.2 Function IterativeSchedule().

Require: Graph *G*, the initiation interval II and the Budget.
Ensure: A schedule *S* or failure information.

```
    // Calculate the schedule time bounds for the current operation
 1: ComputePriority();
 2: while (the list of unscheduled operations is not empty) & (Budget > 0) do
 3:     CurrOper = HighestPriority();
        // Calculate the schedule time bounds for the current operation
 4:     (MinTime,MaxTime) = ComputeSlack(CurrOper);
        // Select the time slot for leakage energy optimization
 5:     SchedSlot = FindTimeSlot(CurrOper,MinTime,MaxTime);
        // Perform the actual scheduling
 6:     Schedule(CurrOper,SchedSlot);
 7:     Budget–;
 8: end while
 9: if all operations are scheduled then
10:     return SUCCESS;
11: else
12:     return FAILURE
13: end if
```

In order to enlarge the idleness in FUs, we always start to search from FU_1 and try to find the earliest empty block on it. If we can not find an empty block in FU_1, FU_2 will be tried next time; then we try $FU_3, \cdots, FU_n$, until we can find such an empty block. In this way, we can schedule operations onto one functional unit as much as possible. And thus, we can enlarge the idleness of FUs and increase the number of unused FUs. The benefit is that we have more chance to put the total unused functional units into low leakage mode before entering the loop body. It is possible that we can not find any empty block to put the operation in. In this case, we employ the same backtracking method as that of Rau's modulo scheduling algorithm [2].

After finding the suitable empty block, we compare the earliest schedule time of the operation and the start time of the empty block in order to determine whether the operation should be scheduled at the beginning or at the end of this empty block. By scheduling the nodes into locations close to each other, we can maximize the consecutive idle period in functional units.

5 Experiments

We have implemented our technique into the Trimaran compiler [1] and conduct experiments using a set of benchmarks from DSPstone [11] and MiBench [7] on the cycle-accurate VLIW simulator of Trimaran. In this section, we first discuss the setup of our experiments in Section 5.1, and then present experimental results in section 5.2.

5.1 Experimental Environment

To compare our technique with the leakage-aware scheduling technique [8], we use the VLIW simulator of Trimaran [1] as our test platform. The configuration for the VLIW Trimaran simulator is shown in Table 1.

Table 1 The configurations of Trimaran.

Parameter	Configuration
Functional Units	4 integer ALU, 2 floating point ALU, 2 load-store units 1 branch unit, 5 issue slots
Instruction Latency	1cycle for integer ALU, 1 cycle for floating point ALU 2 cycles for load in cache, 1 cycle for store, 1 cycle for branch
Register file	32 integer registers, 32 floating point registers

5.2 Results and Discussion

In the experiment, we obtain the results of the leakage energy reduction of Integer ALUs and performance penalty on the code generated by our technique. We compare the energy results with that of the code generated by the leakage-aware scheduling algorithm [8] with leakage management scheme [9]. We compare the performance penalty results with that of the code generated by the modulo scheduling [2].

We assume that the technology is 65nm and 50% of the total energy of the VLIW processor is the leakage energy. In the following, we present and analyze the results in terms of leakage energy reduction and performance penalty.

5.2.1 Leakage Energy Reduction

We compare our algorithm with leakage-aware scheduling technique [8] in Trimaran [1], the percentage of reduction in the leakage energy consumption is shown in Figure 4. In Figure 4, the results for Nagpal's technique and our technique are presented in bars with different color, and the right-most bar "AVG." is the average result.

Our algorithm reduces leakage consumption in the functional units by scheduling operations using less functional units to maximize the idleness of the functional units. Moreover, in the loop-level granularity, our technique minimizes the number of transition time between low level and high level leakage mode by turning off the totally unused functional units before entering the loop body. The experimental results show that our algorithm significantly reduces the leakage energy of the processor. Compared with leakage-aware scheduling technique [8], on average, our algorithm achieves 14.73% reduction for the benchmarks.

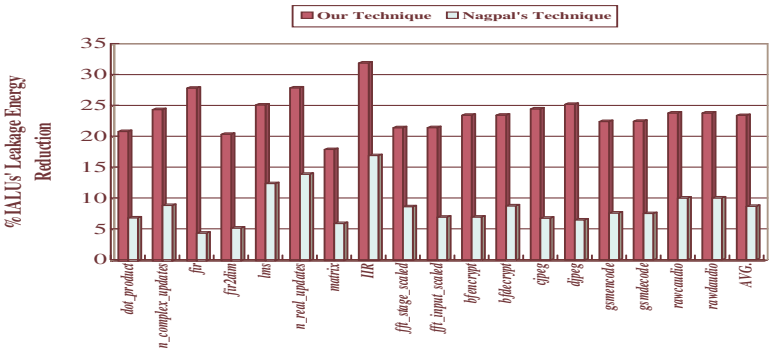

Fig. 4 Leakage energy reduction due to our leakage-aware scheduling algorithm compared with leakage-aware scheduling technique [8].

5.2.2 Performance Penalty

We compare our technique with the performance-oriented modulo scheduling [2], and the percentage of performance penalty is shown in Figure 5. On average, the results show that our technique leads to a 1.74% performance penalty for the benchmarks.

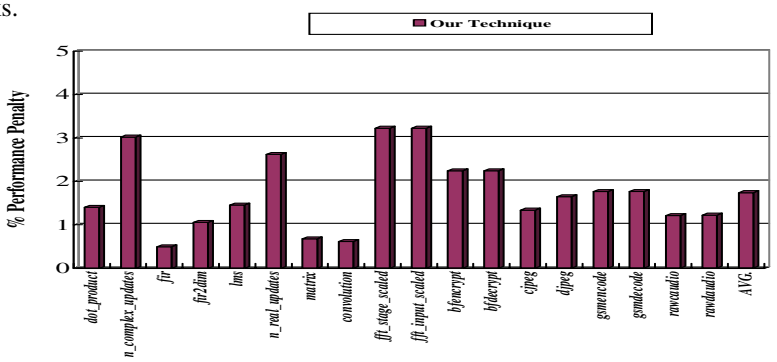

Fig. 5 Performance penalty.

The reason of the performance loss is that our technique may use less functional units to schedule the operations and try to schedule them close to each other. Thus, it may enlarge the schedule length to achieve the goal of maximizing the idleness of functional units. However, in our technique, the performance penalty is controlled by the timing constraint that determines whether employing our technique or not. In the experiment, the maximum number of delays is set as 1.3 * MII (Minimum Initiation Interval). Therefore, the performance penalty is very small. With such small performance loss, our technique is suitable for embedded systems.

6 Conclusion

In this paper, we proposed a leakage-aware modulo scheduling algorithm to reduce leakage energy for DSP applications with loops on VLIW architectures. The proposed algorithm is designed to maximize the idleness of function units integrating with leakage management scheme [9], and reduce the number of transitions between active and sleep modes. We have implemented our technique into the Trimaran compiler [1] and conducted experiments using a set of embedded benchmarks from DSPstone [11] and Mibench [7] on the cycle-accurate VLIW simulator of Trimaran. The results show that our algorithm achieves significant leakage energy savings compared with the leakage-aware scheduling technique [8].

Acknowledgements The work described in this paper is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (CERG 526007 (PolyU B-Q06B), and PolyU A-PA5X).

References

1. *The Trimaran Compiler Research Infrastructure*. http://www.trimaran.org/.
2. B.R.Rau. Iterative modulo scheduling: an algorithm for software pipeling loops. In *27th Annual International Symposium on Microarchitecture*, pages 63–74.
3. J. Butts and G. Sohi. A static power model for architects. In *Proceedings of The 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 191–201, 2000.
4. D.Sylvester and H.Kaul. Power-driven challenges in nanometer design. *IEEE Design and Test of Computers*, 18:12–22, 2001.
5. H.S.Kim, N.Vijakrishnan, M.Kandemir, and M.J.Irwin. Adapting instruction level parallelism for optimizing leakage in vliw architectures. In *ACM SIFPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 275–283.
6. W. Liao, J. M.Basile, and L. He. Leakage power modeling and reduction with data retention. In *2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 714–719, 2002.
7. M.R.Guthaus, J.S.Ringenberg, D.Ernst, T.M.Austin, T.Mudge, and R.B.Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization*, pages 3–14, 2001.
8. R.Nagpal and Y.N.Srikant. Compiler-assisted leakage energy optimization for clustered vliw architectures. In *6th ACM/IEEE International Conference on Embedded Software*, pages 233–241, 2006.
9. S.Dropsho, V.Kursun, D.H.Albonesi, S.Dwarkadas, and E.G.Friedman. Managing static leakage energy in micro-processor functional units. In *the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 321–332.
10. T.N.Mudge. Power: A first class design constraint for future architecture and automation. In *the 7th International Conference on High Performance Computing*, pages 215–224.
11. V. Zivojnovic, J.Martinez, C.Schlager, and H.Meyr. Dspstone: A dsp-oriented benchmarking methodology. In *Proceedings of the 1994 International Conference on Signal Processing Applications and Technology*, 1994.