Power-Sensitive Modulo Scheduling in LLVM

Yuze Dai
Computer Science and Engineering
University of Michigan
Ann Arbor, United States
yuzedai@umich.edu

Qinjuan Xie
Computer Science and Engineering
University of Michigan
Ann Arbor, United States
qinjuanx@umich.edu

Yin Yuan
Electrical and Computer Engineering
University of Michigan
Ann Arbor, United States
aoyin@umich.edu

Jiayun Zou
Computer Science and Engineering
University of Michigan
Ann Arbor, United States
alicezou@umich.edu

***Abstract*—This paper proposes a power-sensitive modulo scheduling algorithm based on swing modulo scheduling algorithm for VLIW processors. The algorithm aims at reducing power consumption by producing an instruction scheduling with more balanced power consumption. We implement this algorithm on the basis of LLVM MachinePipeliner code generation pass. Experimental results shows that the proposed algorithm can produce scheduling that requires less power while still inherits the high scheduling quality from base algorithm.**

***Index Terms*—LLVM, code generation, modulo scheduling, power, VLIW processor, Qualcomm, hexagon**

I. Introduction

Software pipelining [1] is a technique that overlaps multiple iterations of a same loop. It aims at finding a pattern that a new iteration of the loop can be triggered before the last iteration ends. Since different code blocks are executed simultaneously from different iterations of the same loop, the pipelined schedule has to consider data dependency and hardware resource constraints between iterations. This is an aggressive scheduling technique that requires both compiler design and hardware architecture support. Software pipelining is generally applied to VLIW processors which are designed for high throughput works such as digital signal processing [2].

Since computing an optimal scheduling is known to be NP-hard, practical modulo scheduling algorithms implemented in the real-life compilers use heuristic methods. Among them, swing modulo scheduling seems to be the winner [3] by achieving nearly optimal scheduling result with similar computing complexity compared to other algorithms. The llvm project [4] implements a software pipeliner pass using swing modulo scheduling and integrates it to its back-end code generation optimization procedure.

Although swing modulo scheduling algorithm has succeeded in generating schedules with low register pressure with time efficiency, it only evaluates quality of the scheduling with register lifetime. Nowadays, power consumption has become an increasingly critical problem for the efficiency, cost and reliability of integrated circuits, especially of low-end processors. In the view of compiler researchers, we would like to evaluate the impact of scheduling to the hardware power consumption.

There are two kinds of power that we should consider: step power and peak power [5]. Step power is defined as the difference in the average power between consecutive clock cycles. It can cause inductive noise leading to bit flips and logical errors in chips. Peak power is defined as the maximum power dissipation during the execution of a given program. The cooling component and electrical power source of a system must endure the peak power of chip.

To reduce step power and peak power when executing the modulo scheduled program, we design a power-sensitive swing modulo scheduling algorithm that aims at balancing power consumption between each cycle. We implement this algorithm on llvm project by modifying the MachinePipeliner pass. Code of our work is open sourced at https://github.com/aoyin1106/EECS_583_PROJ.

II. Related Work

The idea of designing a power-sensitive modulo scheduling is firstly proposed by H.-S. Yun and J. Kim [5]. They firstly raise up a formula to evaluate the power consumption by each instruction in the partial schedule. At each time slot i, the power consumed, written by $P_{L,i}$, is estimated by:

$$P_{L,i} = \sum_{j} \sum_{op \in O_t} p(op, j) \tag{1}$$

where $p(op, j)$ is the power of some instruction op at time slot j and the set O_t contains all operations at time slot t, where t satisfies $t = (i - j + 1)\%II$.

Then they redesign the Iterative Modulo Scheduling (IMS) to make it power-sensitive when scheduling instructions, which is called as BIMS. Each time a new instruction is picked to be scheduled, the BIMS algorithm calculates the power of instructions in the partial schedule at each time slot and send the new instruction to time slot that consumes less power at this point.

Finally, they implement the modified algorithm and test it on a SPARC-based VLIW test environment. Experimental result shows that their algorithm reduces on average 37.1% of power consumption than original IMS algorithm without little performance sacrifice.

Inspired by their work, we decide to modify the swing modulo scheduling algorithm with similar logic to make it power-sensitive. Although we aim to reduce power consumption of pipelined loop, the performance of our schedule is also expected to be close to schedule optimized for performance only. This objective is made possible by the ordering scheme of SMS algorithm, which picks nodes in the order of how critical they are. Our implementation keeps the ordering part of SMS unchanged and also modify node placing algorithm. We will show how we modify the existing machine pipeliner based on SMS algorithm in llvm project and how we evaluate our work in the following sections.

III. Approach

LLVM has a MachinePipeliner.cpp file which implements its machine software pipeliner pass. We will discuss the implementation in the existing pass and the modification we made to show that in the new implementation, how the power consumption flattened without increase in initiation interval in most cases.

After the ordering step, the compiler needs to schedule each node in a proper sequence so that no memory dependences or control dependences would be broken and the resources would be used in a time efficient way.

A. Existing Traditional Pass

In LLVM's design, this step is completed by brute-force with only As-Soon-As-Possible rule or As-Late-As-Possible rule applied. The values of StartCycle and EndCycle provided are determined by whether the current node has predecessors or successors. It is possible that StartCycle is bigger than or smaller than EndCycle.

If As-Soon-As-Possible rule applies, the algorithm checks the earliest cycle that the node could potentially insert. If so, insertion is done. Otherwise, the algorithm checks the next earliest cycle. Similar procedures are applied to nodes with As-Late-As-Possible rule.

Notice that if with current II, there is no schedule can be found, within the MachinePiperliner.cpp, another function SwingSchedulerDAG::schedulePipeline would try to increase II and redo the above algorithm again.

B. New Power-Sensitive Pass

In the new power-sensitive design, the most important principle is the cost. We currently define that each instruction has the same power consumption or cost, except for zero-cost instructions for the sake of simplicity and time limitation. In an ideal situation, we shall have the data of the power consumption for all kinds of operations. For example, we may safely assume that a multiplication requires more power than an addition, but it is difficult to estimate in numbers that how much greater power a multiplication requires than an addition.

Given a node, our new algorithm calculates for all potential cycles from FirstCycle to LastCycle, with current II, the maximum cost it could possibly take. In other words, for example, for instructions happening in time 0 with II $= 2$, we need to accumulate the cost of instructions happening time $0, 2, 4, ...$ because they could all be running at the same time assuming that we have enough resources. After calculating the cost, the algorithm sorts all potential cycles by cost in ascending order. For the case that two cycles have the same cost, the algorithm follows the traditional rule that with As-Soon-As-Possible rule, the earlier cycle has a higher priority, and with As-Late-As-Possible rule, the later cycle has a higher priority. Meanwhile, the algorithm ignores all instructions with zero cost which can be returned by a corresponding API provided by LLVM.

Our new algorithm would not change the result that whether a schedule can be found or not. If with the limitation of current II, no schedule can be found, it jumps out the function and tries again with a larger II like how the traditional algorithm works. Therefore, it is possible that our new power-sensitive algorithm has a higher II than the II generated by the traditional algorithm, but our experimental results show that for most cases, our II is the same as the II of the original method in many cases.

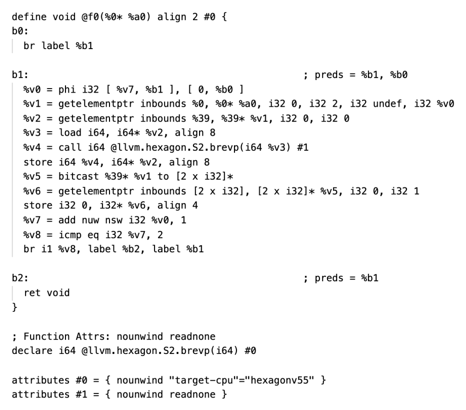
IV. Experimental Result

We tested our algorithm on platform LLVM 14.0. We compile the IR code for hexagon architecture since it a typical target that supports swing modulo scheduling. We use test cases in the regression test suite for hexagon code generation provided by LLVM.

The influence of the algorithm can be best described the following two examples.

A. Examples

1) swp-loop-carried-crash.ll: A typical improved example is swp-loop-carried-crash.ll. It has the same II compared to swing modulo scheduling while the instructions are distributed more evenly. The source code is shown in figure 1.



```
define void @f0(%0* %a0) align 2 #0 {
b0:
  br label %b1

b1:                                               ; preds = %b1, %b0
  %v0 = phi i32 [ %v7, %b1 ], [ 0, %b0 ]
  %v1 = getelementptr inbounds %0, %0* %a0, i32 0, i32 2, i32 undef, i32 %v0
  %v2 = getelementptr inbounds %39, %39* %v1, i32 0, i32 0
  %v3 = load i64, i64* %v2, align 8
  %v4 = call i64 @llvm.hexagon.S2.brevp(i64 %v3) #1
  store i64 %v4, i64* %v2, align 8
  %v5 = bitcast %39* %v1 to [2 x i32]*
  %v6 = getelementptr inbounds [2 x i32], [2 x i32]* %v5, i32 0, i32 1
  store i32 0, i32* %v6, align 4
  %v7 = add nuw nsw i32 %v0, 1
  %v8 = icmp eq i32 %v7, 2
  br i1 %v8, label %b2, label %b1

b2:                                               ; preds = %b1
  ret void
}

; Function Attrs: nounwind readnone
declare i64 @llvm.hexagon.S2.brevp(i64) #0

attributes #0 = { nounwind "target-cpu"="hexagonv55" }
attributes #1 = { nounwind readnone }
```

Fig. 1. swp-loop-carried-crash.ll content

After running the algorithm and verifying with the debug information, we can visualized the dependence graph for swp-loop-carried-crash.ll.

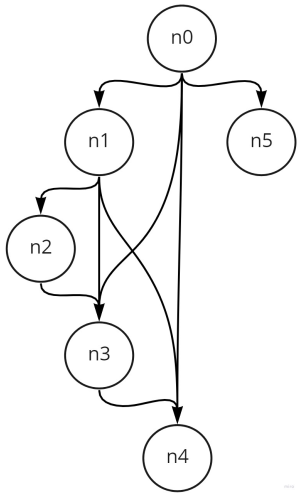
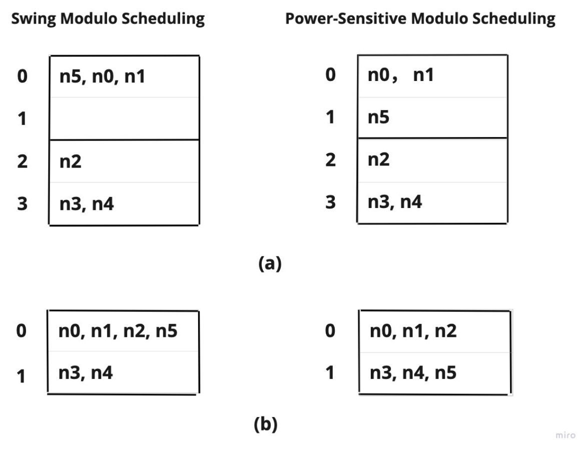


Fig. 2. swp-loop-carried-crash.ll dependence graph

The difference between swing modulo scheduling and power sensitive modulo scheduling is shown in figure 4. Subgraph (a) shows the schedule of one iteration, and subgraph (b) shows the rolled schedule. As we can see, the II keeps to be 2. The largest number of parallel instructions for swing modulo scheduling is in stage 0 where we have 4 parallel instructions n0, n1, n2, and n5. For power-sensitive modulo scheduling, the number of instructions is the same for stage 0 and stage 1. Since we weight every instruction evenly, the schedule generated by power-sensitive modulo scheduling is smoother, meaning its peak power consumption is less than swing modulo scheduling.



Swing Modulo Scheduling

0	n5, n0, n1
1	
2	n2
3	n3, n4

Power-Sensitive Modulo Scheduling

0	n0， n1
1	n5
2	n2
3	n3, n4

(a)

0	n0, n1, n2, n5
1	n3, n4

0	n0, n1, n2
1	n3, n4, n5

(b)

Fig. 3. swp-loop-carried-crash.ll comparison: a) Schedule of one iteration, b) Kernel of the scheduling.

Since swing modulo scheduling focuses on the register pressure, we also compared the register pressure between two schedules. In this example, the register pressure is not increased for power-sensitive modulo scheduling.

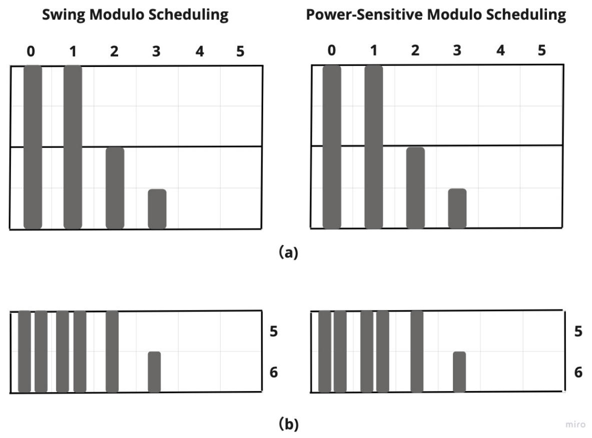


Fig. 4. swp-loop-carried-crash.ll register pressure: a) Lifetimes of variables, b) Register requirements.

Overall, this example showcase the ability of this algorithm to decrease the peak power, which the maximum number of parallel instructions scheduled at the same time.

2) swp-max-stage3.ll: There is another type of example among the test cases, which has a evenly distributed schedule after applying power-sensitive modulo scheduling but having II increased. The test case swp-max-stage3.ll belongs to this scenario with its dependence graph shown in figure 5.

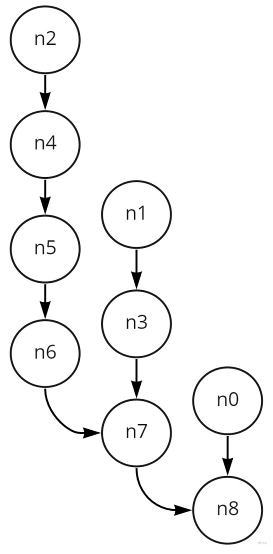
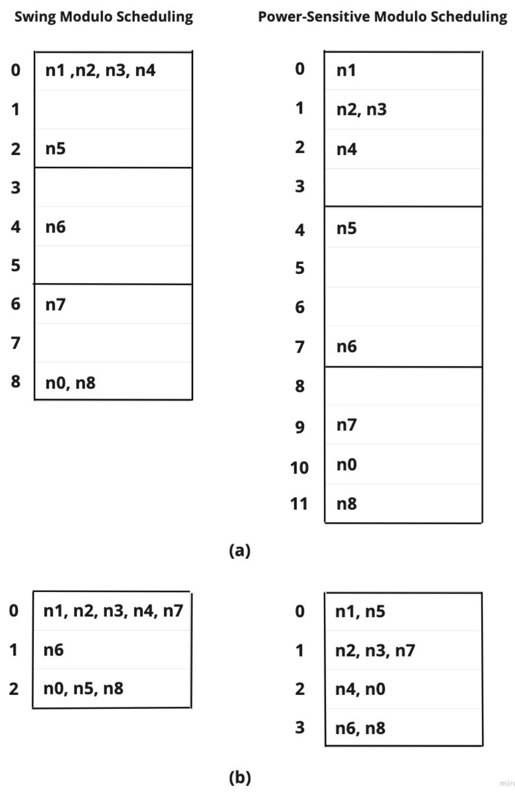


Fig. 5. swp-max-stage3.ll dependence graph

Using the test case swp-max-stage3.ll, the II of the Swing Modulo Scheduling is 3 while the Power-sensitive Modulo Scheduling has a slight higher II, which is 4. Figure 6a shows the schedule of one iteration for each method and Figure 6b shows the kernel of the scheduling. There could be 5 instructions parallelly executed when the Swing Modulo Scheduling method is applied. Compared with it, the Power-sensitive Modulo Scheduling could have at most three instructions at the same time. By the way we define the calculation of power consumption, the power-sensitive modulo scheduling is a more power-wise algorithm due to the decreased peak power of the scheduling.



Swing Modulo Scheduling (a)

0	n1 ,n2, n3, n4
1	
2	n5
3	
4	n6
5	
6	n7
7	
8	n0, n8

Power-Sensitive Modulo Scheduling (a)

0	n1
1	n2, n3
2	n4
3	
4	n5
5	
6	
7	n6
8	
9	n7
10	n0
11	n8

(a)

0	n1, n2, n3, n4, n7
1	n6
2	n0, n5, n8

0	n1, n5
1	n2, n3, n7
2	n4, n0
3	n6, n8

(b)

Fig. 6. swp-max-stage3.ll comparison: a) Schedule of one iteration, b) Kernel of the scheduling

Figure 7 shows the register requirements for both methods. The maximum number of simultaneously live values at any cycle, is the same for the swing modulo scheduling and the power-sensitive modulo scheduling. The $MaxLive$ of the two schedulings are both 6, which means the swing modulo scheduling and the power-sensitive modulo scheduling have the same register requirement in this scenario.

In summary, the example of swp-max-stage3.ll consumes less peak power and requires the same amount of registers by applying the power-sensitive modulo scheduling. But with the II increased, the performance may sacrifice, which is considered possible in our current stage.

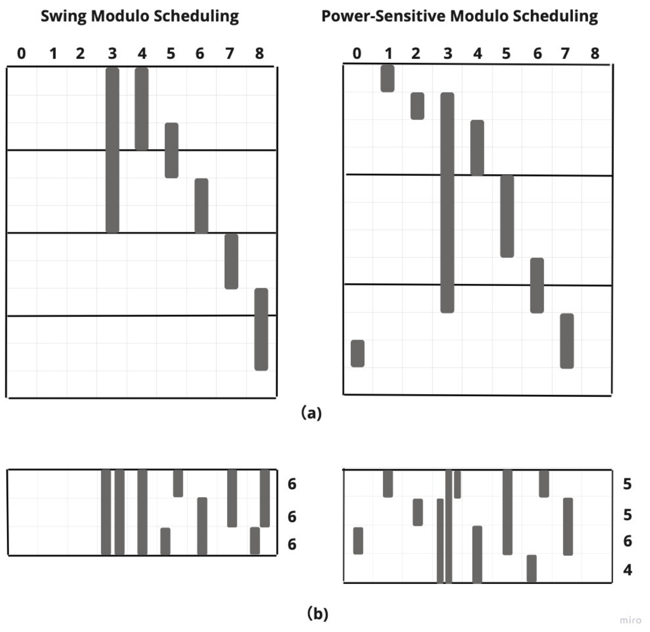


Fig. 7. swp-max-stage3.ll register pressure: a) Lifetimes of variables, b) Register requirements.

B. Evaluation

As shown in the examples, our algorithm is designed to decrease the peak power consumption. In the test cases we run, we can observe that the instructions are distributed more evenly. However, we do not consider the II or register pressure. Therefore, comparing to Swing Modulo Scheduling, sometimes(like the example swp-max-stage3.ll) this algorithm will increase II and decrease performance.

V. Future Work

There is room for improvement in this project.

A. Trade Offs

Power and performance trade off is complex. In our current implementation, we put priority on power consumption without considering the performance. This implementation, in some situations, leads to an increase of II which sacrifices performance. Currently we do not have access to any machine with hexagon architecture, and the simulator is not open sourced which means we cannot modify the compiler in the simulator. A better evaluation standard can be established if we can conduct more experiments on the actual platform.

As a result of the power & performance trade off, energy, which is power $\times$ time, and power trade off has not be determined yet. Our pass only flattened the peak power. If the overall time is increased, it still can consume more energy than the original swing modulo scheduling.

B. Evaluation Matrix

Each instruction counts for one unit of power in our pass, while in the real scenario, the power cost for different actions vary. A more comprehensive evaluation matrix need to be established to make the power consumption calculation more realistic. But this is more of an architecture dependent calculation, and our algorithm is not affected.

VI. Conclusion

As an important heuristic method of optimal scheduling, Swing Modulo Scheduling can generate schedules with low register pressure and time efficiency but it only evaluates the quality of scheduling using register lifetime. In order to evaluate the scheduling in hardware power consumption about step power and peak power, we proposed and implemented a power-sensitive modulo scheduling algorithm based on the LLVM MachinePipeliner pass.

We evaluated our method on the regression test cases for hexagon code generation given by LLVM and found two typical results. Having applied the power-sensitive modulo scheduling, the generated schedule could have a more evenly distributed register requirements but the II could either increase or decrease. The experimental results indicate the power-sensitive modulo scheduling can successfully reduce the peak power but there might be a trade off between the peak power and the performance when the number of cycles in each stage increases.

In the future, a more complex evaluation matrix will be established to make the calculation of power consumption more realistic. And the experiments on the actual platform will also be conducted to address the issue of performance sacrifice.

References

[1] T. M. Lattner, “An Implementation of Swing Modulo Scheduling with Extensions for Superblocks,” Master’s thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, June 2005, *See* `http://llvm.cs.uiuc.edu`.

[2] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero, “Swing module scheduling: a lifetime-sensitive approach,” in *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique*, 1996, pp. 80–86.

[3] J. M. Codina, J. Llosa, and A. González, “A comparative study of modulo scheduling techniques,” in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS ’02. New York, NY, USA: Association for Computing Machinery, 2002, p. 97–106. [Online]. Available: https://doi.org/10.1145/514191.514208

[4] LLVM, “Llvm::machinepipeliner class reference.” [Online]. Available: https://llvm.org/doxygen/classllvm_1_1MachinePipeliner.html

[5] H.-S. Yun and J. Kim, “Power-aware modulo scheduling for high-performance vliw processors,” in *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, ser. ISLPED ’01. New York, NY, USA: Association for Computing Machinery, 2001, p. 40–45. [Online]. Available: https://doi.org/10.1145/383082.383091