저작자표시-비영리-변경금지 2.0 대한민국

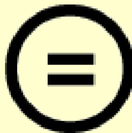
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사 학위논문

Implementation and Improvement of a Swing Modulo Scheduler for VLIW Architecture

VLIW 아키텍처를 위한 스윙 모듈로 스케쥴러의 구현 및 개선

2015년 08월

서울대학교 대학원

전기컴퓨터공학부

정현균

Implementation and Improvement of a Swing Modulo Scheduler for VLIW Architecture

지도교수 백윤흥

이 논문을 공학석사 학위논문으로 제출함

2015년 08월

서울대학교 대학원

공과대학 전기컴퓨터공학부

정현균

정현균의 공학석사 학위논문을 인준함

2015년 08월

위 원 장	문 수 묵	(인)
부위원장	백 윤 흥	(인)
위 원	이 혁 재	(인)

Abstract

Implementation and Improvement of a Swing Modulo Scheduler for VLIW Architecture

Xuanjun Zheng

Electrical and Computer Engineering

The Graduate School

Seoul National University

For VLIW architectures, compiler is in charge of statically scheduling instructions since there are no hardware for hazard detection in this kind of architecture. Thus, instruction scheduling techniques for VLIW architectures have critical influences on both correctness of parallel executions and effective utilization of hardware resources. Software pipelining is one of the popular instruction scheduling techniques which enables overlapped execution of successive loop iterations. We implemented a module of compiler, a swing modulo scheduler, to achieve software pipelining for target VLIW architecture. Experiments on a set of multi-media applications show that with swing modulo scheduler, it has up to 2.6 times speed-up in performance when comparing to the basic list scheduling implementation.

Keywords: VLIW architecture; instruction level parallelism; instruction scheduling; software pipelining; swing modulo scheduling;

Student Number: 2013-23852

Contents

1. Introduction……………………………………………………………………………1
2. Background……………………………………………………………………………3
2. 1 Very Long Instruction Word (VLIW) Architecture…………………………………… 3
2. 2 Instruction Scheduling for VLIW Architecture…………………………………………4
2. 3 Software Pipelining for VLIW Architecture………………………………………………5
2. 4 LLVM Compiler Infrastructure……………………………………………………………… 6
3. Swing Modulo Scheduling…………………………………………………………8
3.1 Build Data Dependence Graphs…………………………………………………………… 8
3.2 Calculate Minimum Initiation Interval (MII)………………………………………………9
3.3 Analysis and Computation……………………………………………………………… 10
3.4 Order Nodes……………………………………………………………………………… 11
3.5 Schedule Nodes…………………………………………………………………………… 12
4. Implementation and Improvement……………………………………………13
4.1 Preprocess Basic Blocks…………………………………………………………………… 13
4.2 Build Scheduling Graphs…………………………………………………………………… 14
4.3 Find or Build Basic Induction Variables………………………………………………… 15
4.4 Calculate Resource MII…………………………………………………………………… 16
4.5 Find All Circuits for Calculating Recurrence MII…………………………………… 17
4.6 Break Anti-dependences…………………………………………………………………… 19
4.7 Compute Partial Order……………………………………………………………………… 20
4.8 Compute Final Order……………………………………………………………………… 21
4.9 Construct Prologue, Kernel and Epilogue……………………………………………… 22
4.10 Check Register Pressure………………………………………………………………… 23
4.11 Adjust Loop Iteration Count…………………………………………………………… 23
5. Experimental Results………………………………………………………………25
5.1 Environment……………………………………………………………………………… 25
5.2 Performance……………………………………………………………………………… 26
5.3 Effectiveness……………………………………………………………………………… 27
6. Conclusion and Future Work……………………………………………………29
Reference………………………………………………………………………………30

Table

[Table 4-1] Computation Time of Finding Circuits....19
[Table 5-1] Experimental Environment....25
[Table 5-2] Benchmark Details....26

Figure

[Figure 2-1] Normal Compiler Structure....7
[Figure 2-2] LLVM Compiler Structure....7
[Figure 4-1] Destination of Loop Exit Branch....14
[Figure 4-2] Remove Weaker Dependences....15
[Figure 4-3] Build Basic Induction Variables....16
[Figure 4-4] Find All Circuits....18
[Figure 4-5] Impacts of Division Operands....18
[Figure 4-6] Computation Time Ratio....19
[Figure 4-7] Break Anti-dependences....20
[Figure 4-8] Recurrences Sharing Same Nodes....21
[Figure 4-9] Situation of Scheduling Recurrences....22
[Figure 5-1] Schedule Quality....27
[Figure 5-2] Speed-up Ratio....27
[Figure 5-3] Time Consumption....28

1. Introduction

One of the methods to improve processing speed of modern processors is providing them with a parallel instruction executing capability. When executing instructions in parallel, it is necessary to ensure that there are no conflicts or hazards between them. This guarantee action called hazard detection can be handled by either hardware or compiler.

To this purpose, there are two kinds of famous approaches, superscalar architecture and VLIW architecture. In a superscalar architecture, hardware is in charge of hazard detection which leads to the architecture consisting of a complex hardware but a simple compiler. In the opposite way, a VLIW architecture is complex in compiler but simple in hardware since hazard detection is managed by compiler. Both architectures have their advantages but when it comes to situations with circuit area limitation or power consumption limitation, for instance embedded systems, it is more inclined to choose the VLIW architecture which has a simpler hardware design.

Apparently, the design of compiler in VLIW architectures is considerable since it is in charge of hazard detection. Among all the compilation steps, it is important to pay more attention to the instruction scheduling step which generates a schedule of executing orders for instructions. Software pipelining is an instruction scheduling technique which exploits parallelism in loops. The key idea is to find a fixed pattern of instructions which can be executed repeatedly by each iteration of the loop. This technique is proved to be effective since most programs spend a lot of execution time in loops.

In this paper, in order to exploit parallelism of instructions in a VLIW architecture, we implemented an instruction scheduler using a software pipelining approach called swing modulo scheduling and made several improvements on it. The implementation of this swing modulo scheduler is based on an open source project called LLVM compiler infrastructure. The improved swing modulo scheduler is proved to get an average speed-up of 2.04 times in test benchmarks when comparing to the basic instruction scheduler provided by the LLVM compiler infrastructure.

The rest of this paper is organized as follows. Section 2 introduces the background knowledge including VLIW architecture, instruction scheduling, software pipelining, LLVM compiler infrastructure and related terms. Section 3 overviews the swing modulo scheduling technique, focusing on its characteristics

and procedure. Section 4 discusses the improvements we made when implementing the swing modulo scheduler. Section 5 shows the results of experimental evaluation in both performance aspect and effectiveness aspect. Section 6 makes a conclusion of this paper and talks about the works to do in the future.

2. Background

2. 1 Very Long Instruction Word (VLIW) Architecture

In order to provide performance improvements, modern processors are equipped with multiple functional units to enable simultaneous execution of multiple instructions. In these architectures, multiple instructions can be executed in parallel if there are no conflicts or hazards between instructions. This kind of parallelism is called Instruction Level Parallelism (ILP). To gain better performance, we need to identify and utilize as much ILP as possible. The representative architectures which exploit ILP to improve performance are superscalar architecture and VLIW architecture.

In a superscalar architecture [2], after fetching multiple instructions from memory, hardware will decide which ones can be executed in parallel without resource conflicts or data dependency hazards, and then dispatch them to corresponding functional units. Resource conflicts will occur when same part of hardware component, such like an arithmetic logic unit, is required by different instructions at the same time. Data dependency hazards will cause problems in situations like when an instruction in progress depends on the result of another instruction which has not been executed yet. In general, in a superscalar architecture, complex hardware dispatching logic is required to manage the hazard detection.

In the other hand, there is no such complex hardware dispatching logic in a VLIW architecture [2]. In this architecture, a fixed number of instructions are fetched and issued directly to corresponding functional units. To guarantee the execution correctness in a VLIW architecture, hazard detection is handled by a software approach. Here compiler is in charge of analyzing and scheduling instructions to ensure no resource conflicts or data dependency hazards will happen.

Since the VLIW architecture is free of complex hardware dispatching logic, it is much preferred to be used in embedded systems where area and power constraints are critical. With reduced hardware complexity, which indicates lower costs, smaller circuits and fewer power consumptions, VLIW architectures can exploit ILP with much simpler hardware design. And in exchange, compiler, especially the instruction scheduling step of compiler, becomes more complex and should be implemented with care.

2. 2 Instruction Scheduling for VLIW Architecture

A Compiler [6] transforms programs written in a high level programming language like C/C++, into lower level target machine instructions. There are several steps in the whole compilation processes and one of these important steps is the instruction scheduling step. The instruction scheduling step can be used to improve ILP by scheduling and reordering the machine instructions which are selected in previous instruction selection step. By doing this step, in architectures like VLIW, programs can be executed correctly and efficiently on target machines.

There are many instruction scheduling techniques focusing on different aspects, such like list scheduling, trace scheduling, and loop scheduling.

List scheduling is a most basic and simple instruction scheduling technique which schedules instructions within a basic block at a time. Basic block is a sequential part of code with only an entry point and an only exit point. First, a scheduling graph is constructed to represent the dependencies between instructions. This graph is then traversed either in a top-down pattern which tries to schedule instructions as soon as possible or in a bottom-up pattern which tries to schedule instructions as late as possible. When there is a dependence relationship between two instructions, the one need to be executed first is called a predecessor and the other one is called a successor. For example in a top-down pattern, scheduling starts from the top of the graph. If an instruction does not have any predecessors in the graph will be scheduled as soon as a corresponding functional unit is available. After the instruction has been scheduled, its successors will be scheduled after specific instruction latency cycles if corresponding functional unit is available at that time. Although list scheduling is very basic and simple to implement, it is very limited in performance since usually it is difficult to find out enough parallelism within a single basic block to fill in all the schedule slots.

Trace scheduling is another instruction scheduling technique which attempts to schedule instructions across basic block boundaries. The name trace means a trace of code execution flow for arbitrary inputs. The basic idea is trying to optimize the control flow path which is executed most frequently. For this purpose, it will schedule instructions on the most frequently executed path together, and perform measures to ensure correct execution by duplicating the branch codes when it is necessary. Trace scheduling increases the code size, but has been proved to be effective in practical since in many cases one control flow is taken much

more frequently than the other one.

Loop scheduling [3] is a third kind of instruction scheduling techniques which focuses on exploiting ILP in loops. This technique is quite effective since in reality programs spend a lot of time dealing with loops, which means loop scheduling can benefit much from little work.

Two kinds of famous loop scheduling techniques are loop unrolling and software pipelining.

Loop unrolling duplicates the loop body several times so that instructions across a loop boundary can be scheduled together. The more times we unroll the loop body, the better performance we can get. Therefore to obtain sufficient performance improvement, we have to unroll many times which increases the code size severely.

On the contrary, software pipelining can get optimal performance improvement without increasing code size. When unrolling loops, we can recognize a fact that there are repetitive portions in every iterations. If we can form that repetitive portions in a pattern for one iteration and repeat this pattern for every iterations of the loop, we can get optimal performance improvement without increasing code size. Software pipelining utilizes this idea. The name software pipelining comes from the fact that instructions from a latter loop iteration are executed in an overlapped fashion with the earlier iteration, similar to the hardware pipelining technique where multiple instructions are executed in flight at the same time.

2. 3 Software Pipelining for VLIW Architecture

There are several terms used in software pipelining technique [2]. The goal of software pipelining is to find a schedule for one loop iteration and exploit ILP just by repeating the schedule for every iterations of the loop.

The repetitive portions of the schedule is called “kernel”. In other words, kernel is a code pattern of instructions that can be executed repeatedly by each iteration of the loop. Similar to the hardware pipelining, the part before pipeline fulfilling with the kernel is called “prologue”, and the corresponding part of pipeline flushing out the kernel is called “epilogue”.

To execute consecutive iterations in an overlapped fashion, an iteration is divided into several stages where each stage has a length of II. The number of stages in one iteration is called “stage count”. A fixed cycle number called “initiation interval (II)” is used to indicate how many cycles later the next iteration

can be initiated after current iteration initiates.

Total execution time of the whole loop is approximately proportional to II, so it is important to find out the minimum initiation interval (MII) of software pipelining schedules. Finding out a software pipelining schedule with II = MII is known as an NP-Complete problem. Hence most software pipelining techniques use heuristic methods to find a schedule with a reasonable II as close to MII as possible. The most popular one is called modulo scheduling.

Modulo scheduling is a trial-and-error heuristic approach to implement software pipelining. First, it computes the MII based on two aspects, resource MII (ResMII, to avoid resource conflicts) and recurrence MII (RecMII, to avoid data dependency hazards), and MII = max (ResMII, RecMII). Then, it tries to compute a schedule for one iteration of the loop with II = MII. If it succeeds in obtaining such a schedule, loop iterations can be executed every II cycles without conflicts or hazards. If fails, it will try to compute another schedule with II = II + 1. Modulo scheduling has been proved to be able to find out a schedule with a time complexity proportional to the square of total instruction numbers in the loop.

In order to achieve software pipelining on target VLIW architecture, we employed a kind of modulo scheduling technique called swing modulo scheduling (SMS). When implement the swing modulo scheduler, we also made several modifications and improvements for obtaining better performance. Details about swing SMS will be discussed in section 3.

2. 4 LLVM Compiler Infrastructure

The LLVM project [1], formerly short for Low Level Virtual Machine, was a research project started in 2000 at the University of Illinois at Urbana–Champaign. With the development of this project, it has lost its original meaning. At present, the LLVM compiler infrastructure is an open source compiler toolchain including a variety of components like optimizer, code generator, and assembler.

Unlike GCC, the LLVM compiler infrastructure [5] is written in C++ and designed as a set of reusable libraries with well-defined modular APIs. With the goal of providing an efficient compiler infrastructure capable of supporting both static and dynamic compilation of arbitrary programming languages, it is designed in 3 separate stages: independent frontend, LLVM optimizer, and independent backend. Therefore it is considered much easier to add support for a new target, by just adding a new target-dependent backend without modifying frontend and optimizer.

Figure 2-1 Normal Compiler Structure

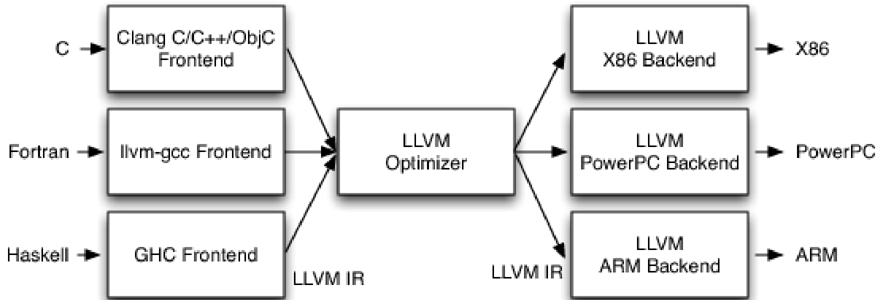


Figure 2-2 LLVM Compiler Structure

Nowadays, the LLVM infrastructure is widely used in many areas. Apple has been using LLVM as an important part of development tools for Mac OS X and iOS since 2005. Also, the primary frontend of LLVM, the Clang compiler, is adopted in the SDK of Play Station 4 console. Clang has been proved to be about 3 times faster than GCC when compiling Objective-C code in a debug configuration.

We modified the instruction scheduling step of the LLVM optimizer using swing modulo scheduling technique to achieve software pipelining for target VLIW architecture by implementing an improved swing modulo scheduler.

3. Swing Modulo Scheduling

Swing modulo scheduling (SMS) [4] is a heuristic software pipelining approach which can generate high quality schedules with low compilation time. It can be used in code generations for innermost loops without subroutine calls of if statements. Loops containing if statements can be handled with if-conversion. It is near optimal in terms of initiation interval, stage count and register requirement.

Initiation interval determines the issue rate of loop iterations and stage count determines the number of iterations of epilogue, so it is better to reduce both of them to obtain a more efficient schedule. To this end, it will consider how critical and how important an instruction is.

Since register is also a limited resource as functional units, lower register requirement is better. Register requirement approximately equals to the maximum number of simultaneously live values. To reduce register requirement, SMS will try to keep every instructions as close as possible to both its predecessors and successors.

SMS first computes a schedule trying to minimize II regarding register pressure and after that it allocates variables to registers. If register requirement is less than actual available register numbers, the schedule is feasible and will be adopted. If not, actions will need to be taken like increasing current II to try rescheduling or adding memory spill codes to fit the number of available registers.

We will first give a brief overview on the procedures of SMS and then discuss the details of each procedure.

At first, it builds a graph called data dependence graph (DDG) to represent basic blocks. Secondly, DDG will be analyzed and several computations will be done. Thirdly, the DDG and computation results are delivered as inputs of the ordering nodes step. Then, a scheduling order will be computed where the order is generated only once. At last, it tries to find a valid schedule according to the order provided in previous step. When finding out a valid schedule successfully, it will generate the prologue, kernel and epilogue for the schedule.

3.1 Build Data Dependence Graphs

First of all, a data dependence graph is built to represent basic blocks.

A data dependence graph consists of a set of 4 elements, DDG = { node, edge, latency, distance }. Each instruction in basic blocks becomes a node in DDG. Each

edge indicates a dependence relationship from one node to another. Latency and distance are two attributes attached to every edges.

Latency is a delay value that indicates no earlier than how many cycles the successor can be executed after its predecessor executes. Only data dependences are included since SMS can handle only loops including one branch instruction at the end of the loop with an iteration count. There are 3 types of data dependences: true dependence, anti-dependence, and output dependence. True dependence is the situation that a successor is going to read a variable when the variable is being written by its predecessor. Anti-dependence is the opposite way of true dependence that a predecessor is going to read a variable when the variable is being written by its successor. Output dependence is the situation that both the predecessor and the successor are going to write to same variable, in other words, the variable is being overwritten.

When employing SMS, despite these 3 kinds of data dependences, a kind of additional dependence is also needed to consider. Since software pipelining is a loop scheduling technique trying to schedule instructions across loop boundaries together, the dependences across loop iterations must be considered. This kind of dependence is called loop-carried dependence. For instance, in the case of memory instructions, there can be loop-carried dependences between instructions belonging to several iterations later.

To represent loop-carried dependences, information called iteration distance is needed. The iteration distance is a nonnegative value. If the distance equals to 0, it means that there is only intra-iteration dependence. And if the distance is greater than 0, it means there is also loop-carried dependence. When there are loop-carried dependences in DDG, there will be cycles or recurrences in the graph. In this case, to enforce the dependence relation between corresponding instructions, II need to be greater than a certain value.

3.2 Calculate Minimum Initiation Interval (MII)

Minimum initiation interval (MII) is the ideal II that can be achieved with a modulo scheduling technique. Since modulo scheduling algorithms first try to find a schedule for a certain II and when it fails, algorithms will have another try with an increased II = II +1. It saves a lot of computation time if MII is known and scheduling starts with II = MII.

It is somehow difficult to find out the optimal II directly, so instead in modulo scheduling algorithms, two kinds of sub-MII are adopted instead. MII is based on

the greater value of following two aspects: resource MII used to avoid resource conflicts and recurrence MII used to avoid data dependence hazards. In short, MII = max (ResMII, RecMII).

Resource MII can be calculated easily. If an instruction is scheduled at cycle x, it will also be executed at cycle x + II, x + 2 x II, and so on. This means same piece of resource will be occupied by the instruction repetitively, indicating resource conflicts should be considered. It calculates resource MII of each kind of resource functional unit, and the maximum value will be the final resource MII. ResMII = max (number of one kind of functional unit required in one iteration / number of one kind of functional unit available in total). For example, considering there are 6 add instructions in one iteration and total 2 add functional units, the ResMII will be 3, which means because of the limited number of add functional units, next iteration cannot initiates within 3 cycles.

Recurrence MII can be computed using the information in DDG. Two elements of DDG, latency and distance, are mainly used to calculate RecMII. Considering 3 kinds of data dependences and the loop-carried dependence, for a scheduling function S(x) indicating the cycle when node x is scheduled and there are two nodes u and v, to avoid all the dependence hazards it need to satisfy S(v) – S(u) >= latency(u, v) – distance(u, v) x II. Similar to the way computing ResMII, RecMII = max (sum of total latencies in one iteration / sum of total distances in one iteration). For example in one iteration, if total latencies is 9 and total distances are 3, the RecMII will be 3. This means to avoid dependence hazards, next iteration cannot initiates within 3 cycles.

In this way, MII is calculated by MII = max (ResMII, RecMII), which means the greater value of the constraints determines final MII. Actual optimum II of the schedule may be greater than the MII calculated, but since it is difficult to find out the optimum II, this MII calculated is used to initiate the II value to start scheduling.

3.3 Analysis and Computation

In previous steps, information of node, edge, latency, distance and MII have been computed. Then the DDG is analyzed and a data table is generated to store information. In this step, one backward edge of each recurrence is ignored to avoid cycles.

There are 5 kinds of data in a data table. SMS computes ASAP, ALAP, Mobility, Depth and Height for each node in DDG. ASAP indicates the earliest time at which the corresponding node could be scheduled. On the contrary, ALAP indicates the

latest time. Mobility is a value denotes the number of slots at which corresponding node could be scheduled. As one of the key ideas, SMS will consider how critical and how important a scheduling instruction is. Mobility is the used to represent this feature. If a node is located in the most critical path, which indicates the longest path in latency, mobility of the node will be zero. This means the node must be scheduled at corresponding cycle to ensure the iteration can be finished in time. Depth is the maximum number of predecessors weighted by their latency. On the contrary, height is the maximum number of successors.

In this way, all the information needed is computed and prepared in a data table.

3.4 Order Nodes

In this step, a scheduling order is computed and this order will be generated only once. After this step, all nodes in DDG will be set in a final order.

When ordering the nodes, two main principles are followed. The first one is giving priority to the nodes in the most critical path. This is achieved by using the mobility information. Lower mobility means the node is located in a more critical path and need to be considered earlier. The other principle is scheduling a node as close as possible to both its predecessors and successors. This is used to reduce the register pressure. For instance, when a node has been already set in an order list, the later its successor schedules, the longer this value occupies a register. So in this situation it is preferred to schedule its successor as soon as possible. This step has two kind of situations: DDG without recurrences and DDG with recurrences.

Let us first consider the simple situation that there are no recurrences in DDG. The name of SMS actually comes from this part of algorithm. When traversing the DDG, it uses two kinds of basic list scheduling technique alternatively: Top-down scheduling and Bottom-up scheduling. The algorithm swings between these two scheduling techniques, so it has the word swing in its name. It first finds out the most critical path in DDG, in other words, the longest path when considering latency. Then it will start a Bottom-up scheduling from the bottom node of the critical path. It goes upwards and visits all the predecessors depending on the depth information. When the predecessors have same depth, it will visit the node with a lower mobility. In this way, after traversing all the predecessors, it comes to the top of the DDG. Then it will swing the Bottom-up scheduling to a Top-down scheduling and travel downwards. This time it considers the height and mobility information. The algorithm will stop after visiting all the nodes in DDG by swinging

between Top-down scheduling and Bottom-up scheduling techniques.

Then we will see how to handle the complex situations with recurrences. First, the DDG will be divided in several sets called partial orders. The first set is the nodes in the recurrence with highest RecMII. Backward edges are ignored to avoid cycles here. It only focuses on current recurrence and traverses it using the previous swing algorithm. After doing this, it will move to next recurrence with the second highest RecMII in the same way. Nodes between these two recurrences are also considered to avoid the situation that when we scheduling a node, both its predecessors and successors are already scheduled. Remaining recurrences and nodes are handled in the same way.

In conclusion, ordering the nodes in practice will have two stages. First, nodes in DDG will be allocated to different partial orders according to their RecMII, higher RecMII gets higher priority. Then these partial orders are used to generate a final order from highest priority to lowest priority.

3.5 Schedule Nodes

In previous step, the final order of DDG is computed. No matter how many time this step repeats, this final order is computed only once.

Nodes are scheduled to an empty slot of a schedule with II length cycles and this schedule with nodes already scheduled inside is called a partial schedule. Similar principles are adopted in this step. It tries to schedule nodes as close as possible to its neighbors. It schedules all the nodes in the final order one by one, and there are different scheduling methods in different situations.

If a node has only predecessors in the partial schedule, this node is going to be scheduled into an empty slot as soon as possible, or if a node has only successors in the partial schedule, it will be scheduled as late as possible.

If a node to be scheduled with both its predecessors and successors in partial order, algorithm will compute the available scheduling period of the node and set it into a free slot. This situation will happen exactly only once for each recurrence. If a node to be scheduled has neither predecessors nor successors in partial order, it will be simply scheduled in a free slot as soon as possible.

Scheduling starts with II = MII and all the nodes in a final order are scheduled according to their situations until scheduling successes or fails. Once it fails because of there is no empty slot to schedule a node, meaning it is invalid to find out a suitable schedule with II = MII. Then II will be increased as II = II + 1 and this scheduling step will repeat until finding out a valid final schedule.

4. Implementation and Improvement

4.1 Preprocess Basic Blocks

In order to apply swing modulo scheduling on target machine, we need to do some preprocessing on basic blocks first.

The first thing to do is converting all the pseudo instructions into real machine instructions. In LLVM, after the instruction selection step, some instructions are still remaining in status of pseudo instructions, which are not real machine instructions. They are kept in pseudo instructions status for the reason of optimization or convenient expansion. Expansion is an action that pseudo instructions can be simply expanded into real machine instructions in later compilation steps. But since we are going to implement a swing modulo scheduler with LLVM, these pseudo instructions must be converted into real machine instructions. This is because when using modulo scheduling techniques, schedules must be fixed after the instruction scheduling step. Also, another reason is that pseudo instructions do not take along information required in modulo scheduling techniques, such as latency information.

Then, we need to preprocess for the some situations containing branch instructions.

Sometimes, operand of branch instructions is stored in a register even though the loop iteration count is a constant number. In this case, we need to change the type of corresponding operand from register into immediate to reduce register requirement.

Also in some other cases, destination of a loop exit branch is not the loop itself. It points to a fall-through block and another jump instruction points back to the loop. It makes the graph becoming very complicated. To handle this, destination of the loop exit branch is changed to point back to the loop itself with the branch condition reversed, then the redundant jump instruction is removed. This case is illustrated in Figure 4-1.

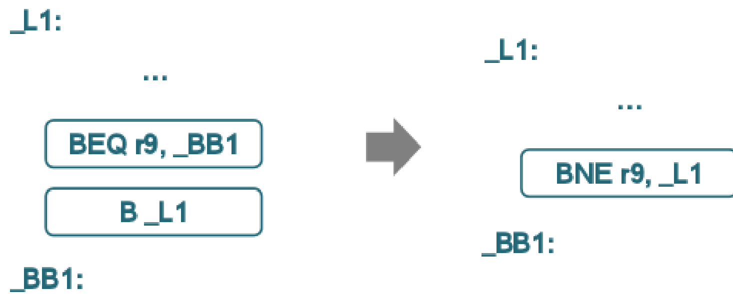


Figure 4-1 Destination of Loop Exit Branch

4.2 Build Scheduling Graphs

LLVM provides APIs for building scheduling graphs. But these scheduling graphs do not employ software pipelining techniques. Coming out with the graphs do not contain information about loop-carried dependencies. Thus loop-carried dependencies are added to the graphs manually by new functions.

After adding loop-carried dependencies manually, there is another problem that graphs quickly become very complicated. This is because for every intra-loop register dependence, we need to add an inter-loop dependence which connects to the same instruction in next iteration.

To reduce the complexity of graphs, we handle this by a method that if there exists more than one kind of dependences between the source and destination instruction, the weaker one is removed. This action is safe because when calculating latencies, greater values are in the leading position. For example, in Figure 4-2, there are total 4 dependences between two instructions. Since true-dependence which always greater than 1, is stronger than anti-dependence which is always 0, the anti-dependences can be removed safely to reduce the graph complexity.

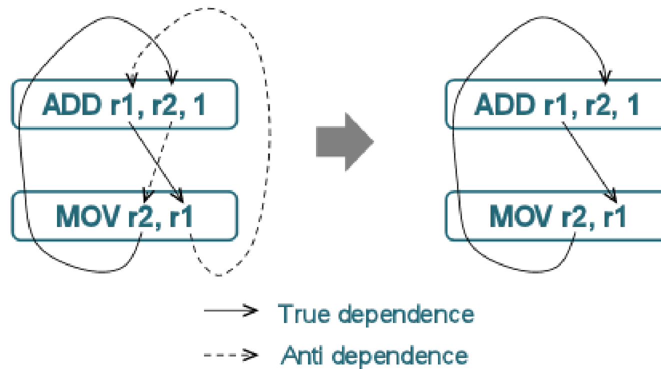


Figure 4-2 Remove Weaker Dependences

4.3 Find or Build Basic Induction Variables

In this step, dependence information included in scheduling graphs is used to find or build basic induction variables. Purpose of this step is to simplify instructions related to the loop iteration count.

There exist dependencies between instructions defining/using induction variables and the branch instructions. Unlike other instructions, when scheduling branch instructions, control dependences need to be considered as well as data dependences. Thus, a branch instruction is required to be the last one of all the instructions in same iteration.

For this reason, if there exist many instructions directly/indirectly related to branch instructions, scheduling will become very difficult. So in this step, induction variable instructions are made by the form R = R + c. And for branch instructions, an additional instruction is added in the loop, which changes the register values in the same way as the original induction variables. By adding this redundant instruction, scheduling constraint is greatly mitigated.

As an example, in the left part of Figure 4-3, the BEQ instruction is a branch instruction, which must be the last instruction in the iteration. In this situation, it cannot be scheduled without violating control dependences. In the right part of the figure, a new instruction using register r11 is added and the BEQ instruction is changed to use this r11 value instead. In this way, original induction variable instruction is free from the branch instruction and can be scheduled more than II cycle away from the branch instruction.

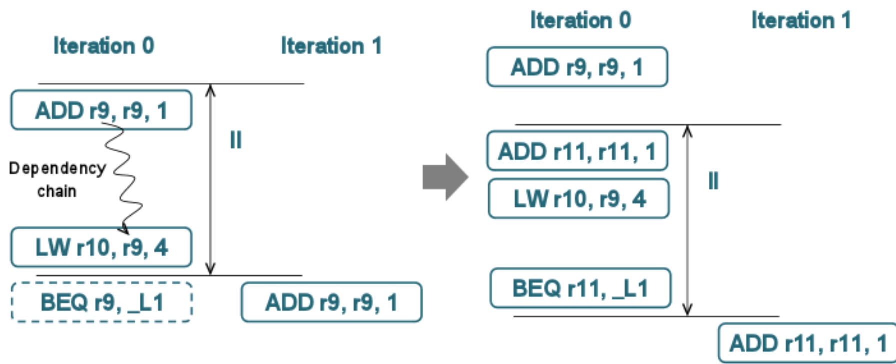


Figure 4-3 Build Basic Induction Variables

4.4 Calculate Resource MII

Resources are limited because some instructions can be only executed on specific function units. This means even though the addition function units are all idle, we still cannot execute a multiply instruction on them. Besides, there is also a little difference that in VLIW architectures, a functional unit can handle not only one kind of computations, but several kinds of them. This makes the original computing method of resource MII being imprecise.

Thus, simply calculating the number of total instructions dividing by the number of total function units gives a conservative resource MII which may smaller than the actual resource MII. For example, assume that we have two function units in a VLIW architecture. FU1 can handle both Add and Load instructions and FU2 can handle only Add instructions. When there are 4 Add instructions and 2 Load instructions, if we compute resource MII in the original way, resource MII will be max (4/2, 2/1) = 2. But since Load instructions cannot be executed in FU2, actual resource MII is 3.

For this reason, when calculating resource MII in our implementation, we do it in a little different way.

First, for each kind of function unit, we calculate the number of instructions that can be only executed in this kind of function unit. Then, for instructions which can be executed anywhere in several kinds of function unit, they are distributed one by one to a less occupied function unit. At last, we check for the maximum number of allocating instructions for each function unit and this value becomes the resource MII.

In this way, if there are a lot of instructions in the loop that can only be executed in a specific kind of function unit, we can avoid setting the resource MII

to a lower value than the value actual can be, avoiding to start scheduling with an impossible II.

4.5 Find All Circuits for Calculating Recurrence MII

To calculate recurrence MII, and further to compute the scheduling order of instructions, cycles or recurrences in graphs need to be found out. For this purpose, Johnson's algorithm which finds out all circuits in graphs is adopted. But when we first implemented this algorithm, it just took too much time to find all circuits in DDG. To handle this problem, we made a modification to Johnson's algorithm by finding out only recurrences with larger II values.

Before introducing the modification to Johnson's algorithm, it is better to explain the original Johnson's algorithm briefly with an example.

Original Johnson's algorithm first starts looking for all cycles from the first node and after that the node is removed from the graph. Removed nodes are pushed into a visited node stack and set with a blocked status, meaning when looking for cycles in later steps, these blocked nodes cannot be passed through. After handling the first node, it searches for all cycles starting from the second node and after that, the second node is also removed and pushed into the stack. Rest nodes are proceeded in same manner until there is no more node left in the graph. During this process, Johnson's algorithm takes some measures to reduce the processing time.

In Figure 4-4, assume we have already traversed nodes 0-1-2-3. Since node 1 has already been visited, no more paths can reach node 0. So we need to pop node 3 from the visited node stack and try to search for another path through node 0-1-2-4. Since from earlier processes, we have already known that we cannot reach node 0 through node 3 by visiting node 0-1-2-4-3, which means that there is no need to visit node 3 again. So we need to backtrack all the paths to node 0 and try to visit by node 0-3 first. But in this situation, if node 3 is still considered to be blocked, we could miss the cycle 0-3-1-5-0.

To handle this problem, we need to unblock node 3 at an appropriate moment. Here in this graph, we can see that when node 1 is popped from the visited node stack, a path from node 3 to node 0 is available to find out the cycle 0-3-1-5-0. So, when node 3 is marked as blocked by node 1 because of the block path of node 0-1-2-3-1, we can add node 3 to the block list of node 1. Now when node 1 is popped out from the stack, we can recursively unblock the nodes registered in the block list of node 1 to active node 3.

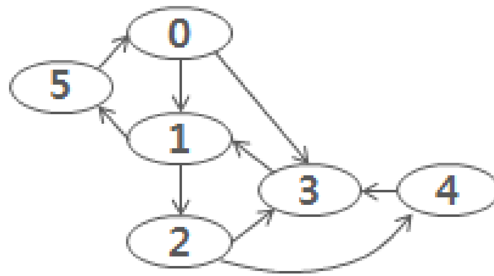


Figure 4-4 Find All Circuits

The original Johnson's algorithm is good enough to find out all the cycles in a graph but it has a high time consumption. So we make some modifications to reduce the computing time by a little trick of ignoring some paths with obviously small II.

II of recurrences is determined by sum of the latencies of all the instructions divided by sum of the iteration distances. For example in Figure 4-5, the recurrence MII of the left graph equals to (2+2+0)/1 = 4, while the right side is (2+2+2+0)/(1+1) = 3. We can realize something from results of this division example, when the operands of a division computation becoming grater, the divisor has greater impacts than the dividend. Therefore, for a recurrence, if the divisor, sum of iteration distance is a large value, the resulting II is likely to be small.

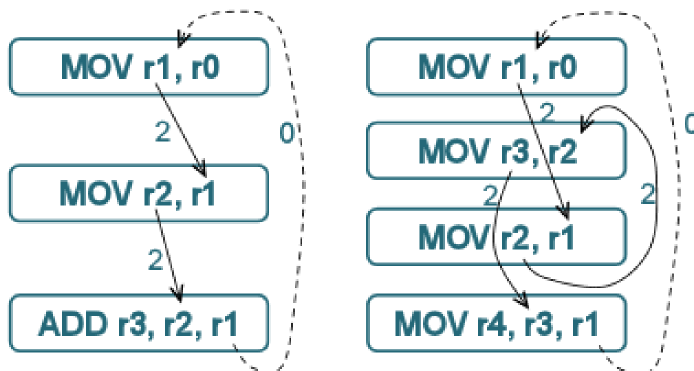


Figure 4-5 Impacts of Division Operands

Based on this fact, the original Johnson's algorithm is modified to find out only recurrences with a limited total iteration distance. During the graph traversal process, we keep a current sum of iteration distance that we have crossed. During

traversing edges, whenever the current sum value gets larger than the predetermined limit value, we could choose not to cross the edges to save time.

This modification has reduced the computation time in traversal significantly and the negative impact on the resulting schedules is negligible. Table 4-1 and Figure 4-6 shows the computation time spent in initial implementations for several applications. We can see that especially in the case of FFT_loop2, the modified algorithm achieved a speed-up over 1000 times.

	Full Search	Improved Search
FFT Loop 1	0.395	0.010
FFT Loop 2	32.287	0.017
HPF	0.009	0.004
FIR	0.036	0.012

Table 4-1 Computation Time of Finding Circuits

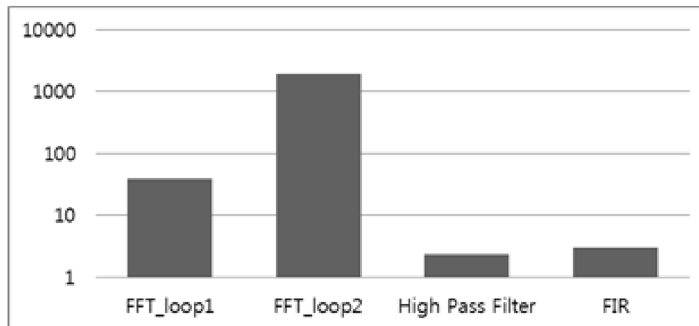


Figure 4-6 Computation Time Ratio

4.6 Break Anti-dependences

Among all the recurrences found in previous stages, some may have larger recurrence MII than resource MII. In this situation, if the recurrence MII is computed including anti-dependences, these anti-dependences can be broken into two pieces to reduce recurrence MII.

Figure 4-7 shows how to break anti-dependences to reduce recurrence MII. If we add a “move” instruction to the left graph, which became the graph in the right, the anti-dependence from the last instruction to the first instruction is broken into two anti-dependences.

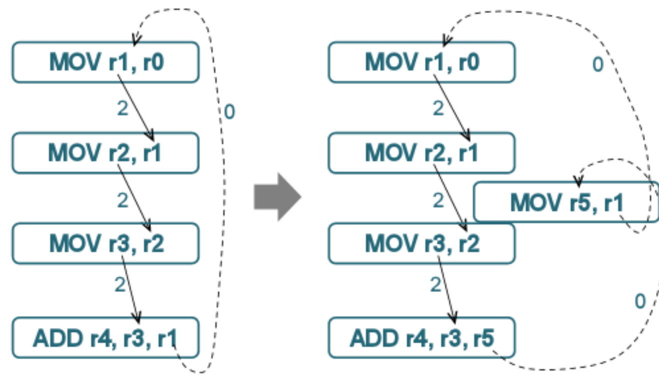


Figure 4-7 Break Anti-dependences

In this way, even though the total iteration distance has been increased slightly, the recurrence MII is halved. Intuitively, by breaking anti-dependences, there is no need for the first instruction of next iteration to wait until executing the last instruction of current iteration. After breaking anti-dependences in the graph, the scheduling process will build the scheduling graph again.

4.7 Compute Partial Order

For recurrences with a large MII value, original method of SMS which gives higher scheduling priorities to instructions in these recurrences is used. These recurrences must be scheduled tightly in the schedule in order to get a valid schedule with the given II. Possibility of finding out a successful schedule is increased if these instructions with higher priorities are considered first. On the contrary, if other instructions are scheduled earlier, when it comes to schedule these instructions with higher MII, it will be more difficult to schedule them close to their dependent instructions due to there remains less empty slots, which increases possibility of scheduling failure. So instructions are set to different partial orders according to their MII.

The problem is, additional complexity will arise when multiple recurrences with similar II values are overlapped. That is, some instructions will be shared by

several recurrences. In this case, assigning scheduling priority to one of the recurrences may lead to a scheduling failure.

Example of this kind of situation is shown in Figure 4-8. There are two recurrences with the same MII. Assume that all the instructions in Figure 4-8 need to be scheduled in same function unit FU1. If we schedule the instructions belonging to the recurrence denoted by bold lines first without considering instruction 5 and 6, it will lead to a scheduling failure when it schedules instruction 6.

In this case, if instruction 4 is scheduled one cycle later, instruction 6 could be able to schedule. But since instruction 5 and 6 are not considered at that moment, algorithm will always schedule instruction 4 as close as possible to instruction 3, leading to a scheduling failure. This problem cannot be solved even we increase II and try to schedule again.

In order to solve this problem, both recurrences need to be scheduled together and set into same partial order when they are sharing same nodes.

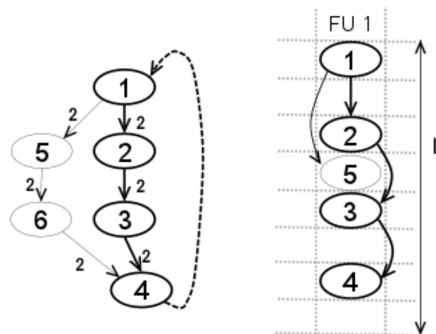


Figure 4-8 Recurrences Sharing Same Nodes

4.8 Compute Final Order

In the previous step of computing partial order, higher scheduling priority is given to recurrences with higher MII, but this principle is only for those with recurrence MII higher than resource MII. The idea behind this is for situations of recurrence MII smaller than resource MII, there is almost no benefit of scheduling these recurrences first since II has been already bound by resource MII, in other words, instructions in these recurrences will have plenty of empty slots.

Furthermore, it may even lead to a worse schedule since we cannot decide the relative positions of recurrences. Figure 4-9 illustrates an example of this case.

If we schedule recurrences first, some of dependent instruction pairs will be scheduled far away from each other. Since there is a loop-carried dependence for every intra-loop register dependence, and if the distance between these instructions is bigger than II, a valid schedule cannot be obtained anyway.

To avoid this situation, the solution is recurrences with recurrence MII smaller than resource MII are scheduled together with the rest of instructions, using basic top-down and bottom-up traversal techniques.

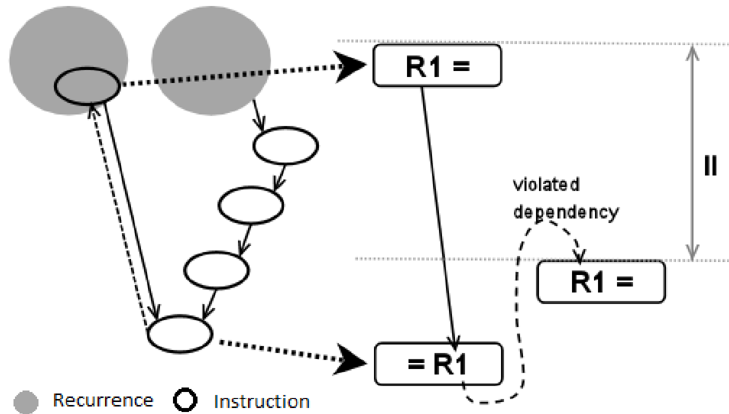


Figure 4-9 Situation of Scheduling Recurrences

4.9 Construct Prologue, Kernel and Epilogue

After a valid schedule is computed successfully by SMS, prologue, kernel and epilogue are generated. In both prologue and epilogue, pipeline is not filled. So there will be many “nop” instructions indicating empty slots.

Thus, it might be better to merge prologue and epilogue with their neighbor basic blocks and re-compute the schedule using basic list scheduler. That is a prologue is merged with its predecessors and an epilogue is merged with its successors. When merging prologue and epilogue, there can be ambiguities between instructions scheduled in the same cycle because merging may destroy the possible dependences between instructions.

To solve the problem, we look for the case where a register is defined in one instruction and used in another instruction in the same cycle. In this case, the defining instruction is placed after the using instruction, otherwise the defining instruction will override the value used in the using instruction.

When it comes to scheduling epilogue after merging, values computed in kernel may not be ready for the instructions in epilogue. Normally, the LLVM

scheduler checks and makes sure that all the values defined in a basic block will get ready for the following basic blocks. The result is that in the kernel, some instructions related to epilogue have to move to other slots to fit this feature, which leads to an inefficient schedule in modulo scheduling techniques.

Also keep tracking of all the possibilities of this situation is very difficult, so instead, we simply add a few “nop” instructions to the start of epilogue manually to ensure the correctness of execution.

4.10 Check Register Pressure

After generating prologue, kernel and epilogue, register pressure will be checked using LLVM APIs. If register pressure is higher than the number of available registers, the schedule computed by SMS becomes useless since it is impossible to fit in spill code in this step. If live interval analysis information is available, more accurate register pressure can be computed. But the information is destroyed due to the highly aggressive modulo scheduling technique.

Therefore, before starting SMS, information about live-in registers and live-out registers are computed and saved, to be used when calculating register pressure. Since live-in and live-out registers will be the same before and after SMS, we can safely use these register information. If register pressure is found to be higher than the number of available registers, the SMS computed schedule is simply abandoned and the basic list scheduler is used instead.

4.11 Adjust Loop Iteration Count

Because of the structure with prologue, kernel, and epilogue in modulo scheduling techniques, there are some restrictions in loop iteration counts. For example when stage count = 1, it is impossible to execute loops in parallel using a prologue, kernel and epilogue structure, which at least need a stage count equals to 2.

For similar reasons, if the result of modulo scheduling has a stage count greater than 1, the formed loop cannot handle the case when loop iteration count is less than the stage count. Thus, if the iteration count is not guaranteed to be greater than the stage count, additional code is added before entering loops.

The additional code checks if the loop iteration count is greater than or equal to the stage count. If so, modulo scheduled loops with a structure of prologue,

kernel and epilogue is executed. If not, original version of loop is executed.

Also, when modulo scheduled loops are executed, the loop trip count should be adjusted before entering loops, since by default it will execute the code corresponding to several iterations of the loops. If it can be guaranteed that the loop trip count is always greater than stage count, the original version of loops can be easily abandoned and modulo scheduled loops are always executed.

5. Experimental Results

In this section, we will present the results of our experimental study. We have implemented an improved Swing Modulo Scheduler for target VLIW architecture based on LLVM compiler infrastructure. We have mainly made modifications in the instruction scheduling step, which is acted after the pseudo-code elimination step but before the register allocation step.

As stated before, Swing Modulo Scheduler is applied to handle innermost loops without subroutine calls or if-statements. The situation of loops containing if-statements actually can be handled by using a skill called if-conversion, but we have not implemented this part of function yet in this project.

5.1 Environment

The experimental environment details are shown in the following Table 5-1.

Experimental Environment	
CPU	Intel Xeon E5-2687W 3.1 GHz
RAM	Up to 60 GB
OS	CentOS 6.5
Language	C++
Target Architecture	SRP4 (Samsung Reconfigure Processor 4)
VLIW Architecture	4 issue (2 scalar, 2 vector)
Register	64 x 32bit scalar registers 32 x 128bit vector registers
Instruction Latency	1 - 5 cycles

Table 5-1 Experimental Environment

To measure the performance and effectiveness of Swing Modulo Scheduler, we adopted several benchmarks of multi-media applications such like Fast Furrier

Transform and so on. We tested both basic VLIW list scheduler and our Swing Modulo Scheduler on main loops of these benchmarks and compared their results of schedule quality and time consumption.

The benchmark details are shown in the following Table 5-2.

Benchmark Details	
Bilateral	Bilateral Filter
BC	Bit Conversion
CSC	Color Space Conversion
FFT	Fast Furrier Transform
GSF	Gussian Smoothing Filter
Histogram	Histogram
HPF	High Pass Filter
Huffman	Huffman Decoder
Median	Median Filter
SAD	Sum of Absolute Difference

Table 5-2 Benchmark Details

5.2 Performance

Performance is measured in two aspects: schedule quality and time consumption.

The first aspect, schedule quality, can be measured by the II value of each final schedule. The II value of each benchmark are showing in the following Figure 5-1.

In the graph, II Before is the final II value found out by the basic VLIW list scheduler when computing a successful valid schedule, and II After is the final II our Swing Modulo Scheduler found out. We can see in most cases, II found by Swing Modulo Scheduler is very close to MII, which means the quality of it is quite

excellent. Especially in the case of bilateral benchmark, the Swing Modulo Scheduler almost reduced the II value 3 times.

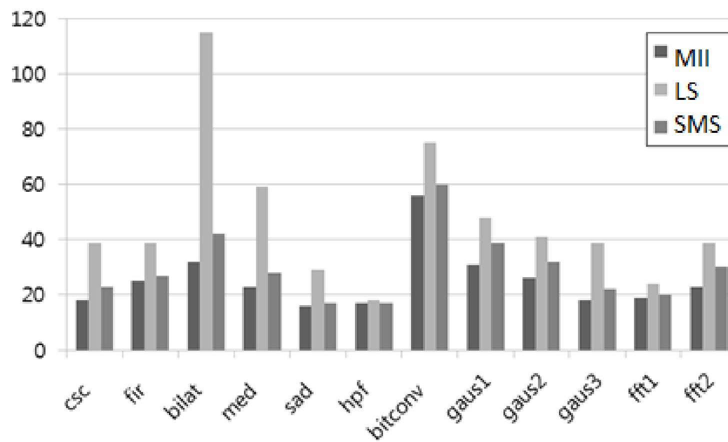


Figure 5-1 Schedule Quality

The second aspect, time consumption is illustrated in Figure 5-2 by the time consumption speed-up ratio of running two schedulers on each benchmark. This ratio is calculated by doing division of the time consumption of Swing Modulo Scheduler to basic VLIW list scheduler.

From the graph we can see that average performance speed-up is 2.04 times. In the best case of bilateral filter benchmark, Swing Modulo Scheduler earns a highest speed-up up to 2.74 times. On the contrary, in the worst case of Huffman decoder benchmark, it gets only a 1.21 time speed-up. The reason of obtaining a lower speed-up is that in this Huffman decoder benchmark, recurrences are longer than ones in other benchmarks.

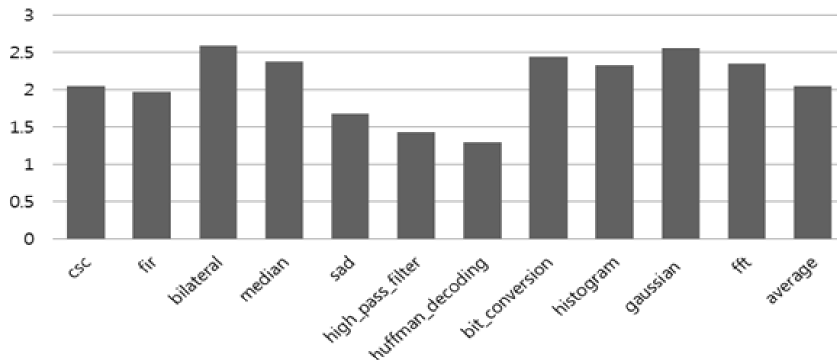


Figure 5-2 Speed-up Ratio

5.3 Effectiveness

Figure 5-3 demonstrates the effectiveness results by showing the compile time consumption of each benchmark.

Since SMS is a heuristic technique to achieve Software Pipelining, it should compute a valid schedule successfully with a reasonable time consumption.

From the graph we can see the compile time in the worst case is still less than 2 seconds and the average compile time is 0.92 second. So we can say effectiveness of the improved Swing Modulo Scheduler is good enough.

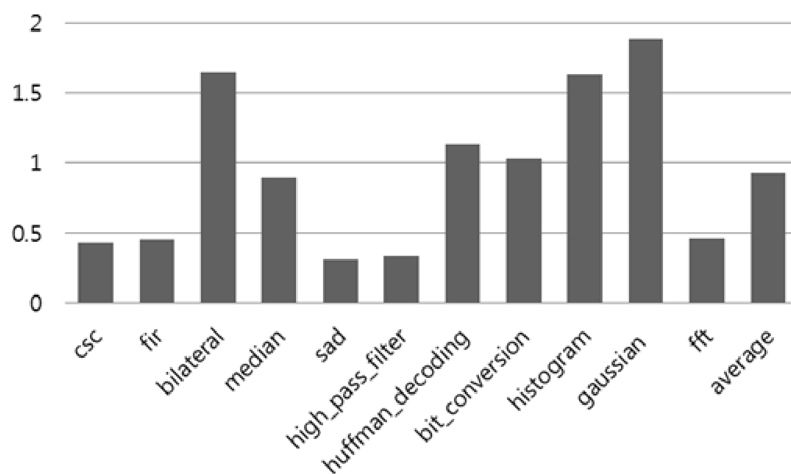


Figure 5-3 Time Consumption

6. Conclusion and Future Work

Processors equipped with multiple functional units are widely used today and parallelism in them are exploited to improve performance. To ensure the correctness of parallelism, hazard detection are needed. In VLIW architectures, due to its simple hardware design, compiler, especially the instruction scheduling step of compiler, is in charge of hazard detection.

In this paper, in order to achieving software pipelining to exploit parallelism of instructions for target VLIW architecture, we implemented a Swing Modulo Scheduler based on LLVM compiler infrastructure and made several improvements. We implemented this function by adding a module of instruction scheduler using swing modulo scheduling technique to the LLVM compiler. With modifications and improvements we made, the swing modulo scheduler for target architecture is proved to have excellent performance and effectiveness comparing to the basic list scheduler provided by LLVM.

These are still a lot of works to do in the future. For instance, the modified algorithm used in finding all circuits in graphs is effective in practice, but has not been perfectly proved. Also, the if-conversion feature which can help the scheduler to handle if statements or while loops in not implemented in this project. These insufficient points will be fixed and completed in the future.

Reference

[1] LLVM, http://llvm.org

[2] Wikipedia, http://wikipedia.org

[3] RAU, B. Ramakrishna, SCHLANSKER, Michael S, TIRUMALAI, Parthasarathy Pm "Code generation schema for modulo scheduled loops", IEEE Computer Society Press, 1992

[4] J. Llosa, et. al., "Swing Modulo Scheduling: A Lifetime-Sensitive Approach", IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques, October 1996

[5] C. Lattner, V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation", Proceedings of the 2004 International Symposium on Code Generation and Optimization, pp. 75-86, March 2004

[6] Y.N. Srikant, P. Shankar, "The compiler design handbook: optimizations and machine code generation", CRC Press, 2007

요약

하드웨어가 해저드 검출(hazard detection)을 지원하지 않는 멀티이슈 VLIW프로세서의 성능을 높이기 위해서는 컴파일러가 명령어 의존성과 하드웨어 자원의 제약을 지키는 범위 안에서 최대한 명령어수준 병렬성(ILP)을 활용하는 것이 중요하다. 기본 블록(basic block) 스케쥴링은 제어 흐름(control flow)의 경계를 넘어선 스케쥴링을 행하지 않아 효과가 제한적이다. 소프트웨어 파이프라이닝(software pipelining)은 루프(loop)의 경계를 허물어 여러 반복(iteration)의 명령어가 동시에 수행되도록 하는 것으로 스윙 모듈로 스케쥴링(swing modulo scheduling)은 그 중에 한 범주의 스케쥴링 기법들을 일컫는다. 본 연구에서는 스윙 모듈로 스케쥴링 기법을 활용해 스케쥴러를 구현하여 2.6 배 의 성능을 행상 시켰다.

학번: 2013-23852