FAKULTÄT FÜR !NFORMATIK

Software Pipelining in a C-Compiler

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Diplomstudium Informatik

eingereicht von

Benedikt Lukas Huber
Matrikelnummer 0025355

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: ao.Univ.Prof.Dipl.-Ing.Dr.techn. Andreas Krall

Wien, 14. 10. 2008

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Abstract

Very long instruction word (VLIW) processors exploit *instruction level parallelism (ILP)* to reduce the execution time of programs by issuing multiple operations in parallel. Since scheduling—especially parallelization—has to be done entirely by the compiler sophisticated algorithms are necessary to utilize the available resources efficiently.

Software pipelining is a scheduling technique to increase the ILP in basic block loops by overlapping the execution of consecutive iterations. The software pipelining heuristic we implemented for the *LLVM* compiler infrastructure is called *swing modulo scheduling (SMS)*. SMS creates dense schedules while keeping register pressure and compile time low.

Zusammenfassung

Very long instruction word (VLIW) Prozessoren nutzen *instruction level paralellism (ILP)* um die Ausführungszeit von Programmen zu verkürzen, indem sie mehrere Operationen zeitgleich verarbeiten. Da das Scheduling, insbesondere die Parallelisierung, ausschließlich vom Compiler durchgeführt wird, sind durchdachte Algorithmen notwendig um die zur Verfügung stehenden Ressourcen effizient zu nutzen.

Software pipelining ist eine Technik um ILP in Basic-Block-Schleifen zu erhöhen, indem die Ausführung von aufeinanderfolgenden Iterationen überlappt wird. Die Software-Pipelining-Heuristik, die wir für die *LLVM* Compiler-Infrastruktur implementiert haben, heißt *swing modulo scheduling (SMS)*. SMS erzeugt dichte Schedules und schafft es die Anzahl der nötigen Register und die Compile-Zeit niedrig zu halten.

Acknowledgments

The development was supported by the *Christian Doppler Forschungsgesellschaft* and *ON DEMAND Microelectronics*.

Contents

1 Introduction

Software Pipelining is a scheduling technique for *very long instruction word (VLIW)* processors. Being a loop scheduling technique or cyclic scheduling technique it tries to maximize the throughput of a given loop by increasing the *instruction level parallelism (ILP)* thus utilizing the available resources of the processor more efficiently. This is achieved by starting the next iteration of a loop before the preceding has finished and thus executing parts of two or more adjacent iterations in parallel.

As programs spend most of their time inside of loops it is of crucial importance for the overall execution time of the application program to find an efficient schedule for loop bodies, i.e. a schedule with a minimized number of clock cycles.

1.1 Motivation

Due to the inability of arbitrarily increasing the clock speed of processors one way to enlarge the amount of data that can be processed in a given time span is to issue more than one computation at a time. This strategy is called *parallel* execution in contrary to *sequential* execution where one instruction after the other is carried out. There are a number of approaches to parallelize computation.
For example *vector processing* also referred to as *single instruction multiple data (SIMD)* where the same operation is applied to a number of independent operands simultaneously. This technique is often applied in the field of high-performance computing, but also graphic processing units found in game consoles and home computers make use of vector processing to render 3D graphics. Vector processing can yield significant performance gains. However it is limited to certain specialized domains.

A more versatile example is *thread level parallelism (TLP)* where multiple sequential threads execute at the same time on different processing units. This form of parallelism is rather coarse grained as the individual threads run relatively independently from each other. Again TLP can only lead to reduced execution time if it is possible to split a program into threads which has to be done by the programmer.

The most fine grained form of parallelism is *instruction level parallelism (ILP)*. In this case the program is analyzed in order to determine which particular operations can execute in parallel.

VLIW

The main characteristic of the *very long instruction word (VLIW)* design philosophy for micro processors lies in the emphasis of *explicit* ILP. The term originates

from the fact that one instruction word actually encodes more than one operation. Explicit ILP means that the programmer, respectively the compiler has to provide the information which operations are executed in parallel. As the processor does not incorporate any hardware features for dependence analysis, this information has to be computed at compile time. The selection of the set of operations that is carried out at a given clock cycle is called *scheduling*. The difficulty is to utilize the resources that the processor provides as good as possible while respecting the dependencies of the operations and the constraints that are imposed by the hardware or the instruction encoding. To guarantee that these constraints are met and all dependencies are respected is the responsibility of the programmer respectively the compiler.

That is one of the main distinguishing features of VLIW machines as opposed to *super scalar* processors, which also exploit ILP. Super scalar machines offer additional logic to analyze dependencies in a sequential stream of scalar operations to determine which operations can safely be issued in parallel without violating any dependencies. Of course this dependence analysis is limited to a certain scope, on the other hand the dispatch unit of a super scalar machine has a lot of runtime information about the program that can be used, for example to predict branches. At compile time this kind of information can only be given in the form of profiling data.

In VLIW processors the ILP is exposed in the architecture and all scheduling issues are left to the compiler, thus shifting complexity from the processor to the compiler. I.e. to trade less complexity in the hardware for more complexity in the software.

The slogan for the VLIW design philosophy is

> If you can do it in software, do it in software! [FFY05]

This exposure of architectural details makes binary compatibility hard to maintain over VLIW processor generations as almost every change in the architecture is reflected in the instruction set. Hence programs have to be recompiled to run for instance on a newer generation of processor, only so called *source compatibility* [FFY05] is offered. However this once again emphasizes the preference of software over hardware in VLIW.

Embedded Systems

The reduced hardware complexity is the main motivation for the use of VLIW processors. As a rule of thumb one can say that less hardware complexity yields less transistors on the chip, what again implies less silicon area needed for implementation. This in turn produces a lower price per item which is of crucial importance for large volume production. That is the reason why VLIW processors are especially

of interest in the realm of embedded systems where production costs outweigh the costs of development because of the large number of manufactured pieces. Less transistors also reduce the heat dissipation in the chip and consequently it can be run at higher clock rates.

Embedded micro processors can be found in a wide variety of consumer products as part of micro controllers that are for example integrated into washing machines or microwave ovens.
With the arrival of digital media devices like digital cameras or MP3 players the demand for higher computing power has risen, as those devices contain *digital signal processors (DSP)* that run the computational intensive decoders and encoders. Due to their properties VLIW processors are a reasonable choice to implement those DSPs. Another attribute of embedded systems is that the software which runs on them does not change very frequently and is most often used only for a single purpose. So these programs are highly optimized towards a given platform. As programs spend most of their time inside of loops one promising optimization is to keep the loop bodies as small and efficient as possible by exploiting as much parallelism as possible. One technique that targets this problem is Software Pipelining.

2 Software Pipelining

Structure of a Compiler

The structure of a compiler can be divided into the machine independent *front end* and the machine dependent part, the *back end*.

Front end

The front end is responsible for the programming language specific analysis and transformations. It examines the program from a very high level point of view. The most important tasks of the front end are lexical and syntactical analysis. Further more in programming languages where a static type system is present the front end ensures compliance to this type system. As it only depends on the programming language the front end of the compiler can be reused for a wide variety of processors. The output of the front end in turn is a machine independent and programming languages independent representation of the program, that is semantically equivalent to the original source program.

This stage is called *intermediate representation (IR)* and forms the connecting link between the front end and the back end of the compiler. There are a number of possible layouts of this IR that all have different properties, however they all have in common that they decompose the program into very fundamental operations. Normally the IR assumes an infinite number of registers that are used for the register operands called *pseudo registers*.

Back end

The back end is responsible for the generation of machine code which is specific to the machine language and independent of the programming language. The most important steps to generate machine code are *instruction selection*, *scheduling* and *register allocation*.

Instruction selection maps IR operations or sequences of IR operations on operations that are actually part of the target language and are semantically equivalent. To achieve an efficient selection the compiler tries to prefer operations that are cheap on the target architecture in terms of resource usage as there is in general more than one mapping from the IR to the actual machine program.
In the case of VLIW architectures this is normally done ignoring any parallelization opportunities as they are handled by the scheduling phase. The selection process also ignores the proper number of physical registers the architecture offers and uses pseudo registers as operands.

During register allocation the compiler attempts to map pseudo registers on machine registers. As there is an infinite number of pseudo registers available, the intermediate program before register allocation can contain a number of different register operands that exceeds the number of machine registers available. Register allocation tries to reuse machine registers of operations that are independent of each other. Where this is not possible it inserts *spill code* to write the operands into the memory and load them to registers when needed. Spilling in general degrades performance as the memory system is very slow compared to register access so places where spill code is inserted have to be selected very carefully.

The phase in which the compiler decides *when* a certain operation has to be issued in the target program is called scheduling. A correct schedule must not violate any data dependencies. To make a schedule as efficient as possible the compiler has to take into account attributes of the processor's pipeline. For example, the compiler must try to fill delay slots if present. A delay slot occurs if for example a jump instruction does not take effect immediately after the cycle the instruction was issued but some cycles later. Thus offering the possibility to schedule instructions after the jump instruction that however will be executed before the jump effectively takes place. Also important to consider is the behavior of the memory system.
Scheduling is the part of a VLIW compiler that differs the most from a compiler without explicit notion of parallelism. The scheduler in a VLIW compiler has more freedom of placing the operations in the schedule as it not only has to decide the sequence of operations but also which operations can be safely executed in parallel. This adds to the complexity of the scheduling problem as one strives to uncover as much parallelism as possible.
A scheduling algorithm can consider the whole program which is called scheduling on a *global* scope. Or it can consider only parts of the program like traces or basic blocks which is called a *local* scope. Where the former may theoretically lead to an optimal schedule the latter is more feasible as global scheduling in general tends to be costly due to its computational complexity. Scheduling algorithms that examine loops in the control flow graph (CFG) are called *cyclic* or *loop* scheduling techniques. Software pipelining falls in this category.

Phases are not independent from each other. And it is also possible to change the order in which they are carried out. For example the scheduling can be done before register allocation or after. If scheduling is performed before register allocation one speaks of a *pre pass* scheduler in the other case of a *post pass* scheduler. This problem is called *phase ordering problem.*
A pre pass scheduler has more freedom of placing the operations that can lead to a better schedule. On the other hand it may increase the minimum number of

necessary registers called *register pressure*. This in turn can lead to spilling which again decreases performance. Register pressure in general is increased because the more ILP the compiler achieves the more registers are live at the same time thus reducing the possibility of reusing physical registers.

2.1 Terminology

The jargon in the realm of VLIW processors is different from the one used in the field of RISC machines and may lead to confusion.

Operation

An operation is the most basic unit of computation. It more or less corresponds to a RISC instruction. On a VLIW processor more than one operation can be issued in parallel if the machines resources and the instruction encoding allow it.

Nop

The special operations that does nothing useful for the computation is called *Nop* which stands for *no operation*. In VLIW processors Nops are explicit in the program. Since they do nothing the goal of many optimizations, especially in the scheduler is to reduce their number. Unfortunately in many cases Nops are inevitable when for example the result of an operation is not available immediately and no other useful and independent operations are at hand. Also instruction words where parallelization is not possible have to be "filled up" with Nops. The responsibility of the scheduler is to fulfil all dependences and constraints and reduce the number of Nops. On the other hand, a schedule with more Nops may nonetheless perform better than an aggressively optimized one due to cache effects.

Instruction vs. Bundle

A set of operations that is executed in parallel is called instruction or group. A bundle is the set of operations that is encoded in the same VLIW. These operations do not have to be independent and are not necessarily executed in parallel, this depends on the binary encoding scheme used. Often however a bundle corresponds to an instruction.
For the scheduler only the creation of instructions by putting together operations is of interest as the final encoding into VLIWs is done by the assembler.

Register Liveness

A register is said to be *live* between the point of its definition and its final use before the next redefinition or if there is no redefinition from the last definition to the final use. This interval is called *live range*.

```
s1  r0 = r1 + r2
s2  r1 = r0 << 2
s3  r5 = r6 + 28
s4  r4 = r0 | 1
s5  r8 = 128
s6  r0 = r9 + r10
s7  jump (.label10)
```

Listing 1: Liveness example

In the example `r0` is live from $s0$ until $s4$. A new live range starts at $s6$.

The number of registers that are live at the same time can never exceed the number of physical registers available. In such situations the register allocator has to insert spill code which in general leads to decreased performance. Many parallelization techniques such as software pipelining are known to increase register pressure, thus the programmer has to be careful how to apply these techniques. Often VLIW processors have a large number of general purpose registers to help parallelizing the programs, but register pressure still remains a problem.

Resource Constraint

Constraints that are caused by the hardware architecture are called resource constraints. Since the architecture offers only a limited set of resources, such as load-store units or arithmetic-logic units *(ALUs)*, the scheduler can only place the operations into time slots where the necessary resources are available. Resource constraints are the main reason why parallelization fails even though all data dependences are met. To address all resource constraints the scheduler needs a detailed description of the available resources and the behavior and resource usage of all operations.

Data Dependence Graph (DDG)

A DDG is a directed graph that represents data dependences in a program. It is an important part of the input for every scheduling algorithm. The nodes of a DDG are individual operations of a program or a part of a program. The edges represent the dependences between the operations. These dependences impose constraints on the order the operations can be scheduled without altering the result

in an unwanted way. In particular they impose constraints on the possibilities of parallelizing operations. A number of different dependences can occur.

- *True dependences* arise if one operation needs the result of an earlier operation as an operand. The true dependence is also called *read after write (RAW)* dependence. An example for a true dependence is:

  ```
  s1  r0 = r1 + r2
  s2  r3 = r0 + r3
  ```

  Operation $s1$ needs the result `r0` of operation $s0$.

- *Anti dependences* are those dependences that represent the fact that a operand has to be read before it is overwritten. Another name for this dependence is also *write after read (WAR)* dependence.

  ```
  s1  r0 = r1 + r2
  s2  r2 = r3 + r4
  ```

  Operation $s1$ overwrites register `r2` which is an operand of $s0$. Thus creating a WAR dependence since the two operations can not switch places without altering the result.

- *Output dependences* occur when two operations write to the same location. They are also called *write after write (WAW)* dependences.

  ```
  s1  r0 = r1 + r2
  s2  r0 = r3 + r4
  ```

  After both operations `r0` should contain the return value of $s1$ thus it is not possible to exchange the position of these operations.

- *Control dependences* are caused by jumps in the control flow.

  ```
  s1  r0 = r1 + r2
  s2  jump (.label12)
  ```

  If jump operations have a delay slot it is also possible to put $s0$ *after* $s1$ depending on the size of the delay slot.

All of these dependences can also occur with memory operands. Alias analysis can determine if a potential dependence between memory accesses exists or not, i.e. whether there has to be an edge in the DDG or not.
The edges of the DDG are normally labeled with the minimal delay the source node must be scheduled before the destination node. This value can also be negative for example in the case of control dependences if there is a delay slot.
There can also be more than one edge between two nodes. The scheduler must then of course satisfy all of them, i.e. take care of the most rigid dependence.

```
s1  .loop:
s2    r1 = r1 + r3
s3    r4 = r4 + 1
s4    if ( r1 > 100 ) jump (.loop)
```

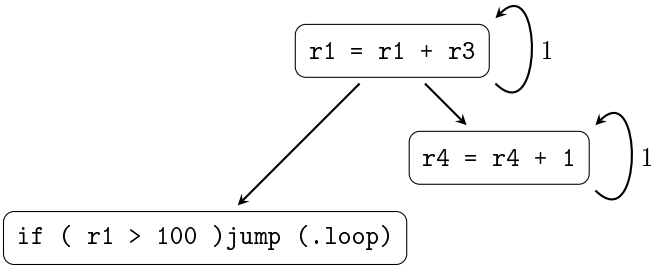

Figure 1: A simple loop containing recurrences.

Initiation Interval (II)

The II [Lam88] is the number of cycles that elapse between the initiations of two subsequent iterations in a software pipelined loop. Algorithms try to minimize the II.

Recurrence

Recurrences are inter iteration dependences in a loop. A more formal term for recurrence is *loop carried dependence*. They occur for example when an operation reads a result that is defined in an earlier iteration of the loop, a RAW dependence. All other forms of dependences can of course also cause a recurrence. Often recurrences are caused by memory accesses since with the absence of proper alias information it is safer to assume a dependence. The number of iterations between the read and the definition is called *iteration distance*. This value is often also a label on the edge. The name recurrence comes from its representation in the DDG which is a loop.
Often the name *cycle* is used, but this term can be mistaken with the meaning of *clock cycle*. Recurrences are the main reason besides resource constants for the increase of the II. In the example 1 the operations $s0$ and $s1$ depend on the previous iteration hence the iteration distance is 1.

Stage

The body of a loop that is going to be software pipelined is divided into disjunct sets of consecutive operations. These sets are called *stages*. Stages do not have

to be of the same length. However, since all stages are overlapped in the kernel shorter stages have to be expanded to the II. This is the reason why in general the latency of one iteration is reduced. Software pipelining tries to maximize the throughput of a loop and not to speed up the individual iterations. How the stages are formed depends on the algorithm.

Kernel

The code that forms the steady state of a software pipelined loop is called *kernel*. It is the part of the schedule that runs repeatedly. Software pipelining algorithms are designed to optimize the kernel as in general it is the part the loop spends the most of its time. The kernel consists of the individual stages of the consecutive iterations that are overlapped. All operations of the original loop body are contained in the kernel thus the pressure on the instruction cache does not rise significantly, at least for the iterative part of the software pipelined schedule. In some circumstances and depending on the hardware architecture copy operations may have to be inserted in the kernel which increase the code size compared to the original loop. The II is the length of the kernel since with every kernel iteration a new iteration of the original loop is started.

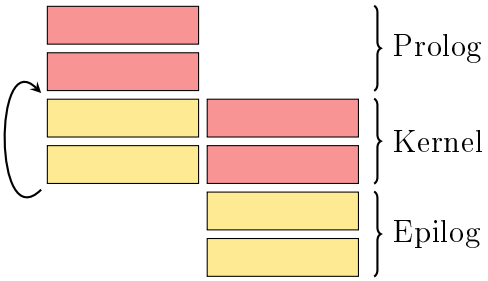

Figure 2: Parts of a software pipelined loop.

Prolog and Epilog

The kernel per se is not semantically equivalent with the original loop. There has to be initialization code in the *prolog* to fill the pipeline. The *epilog* that runs after the kernel empties the pipeline in an analogous way. This additional code increases the code size of software pipelined loops compared to the original. Both the prolog and the epilog are executed only once. Since prolog and epilog are linear code not containing cycles they can be scheduled using traditional acyclic scheduling algorithms.

Stage Count (SC)

The stage count is the number of stages a loop body is divided. The stage count is of importance for the construction of prolog and epilog and in turn for the resulting code size. It plays a role for the register pressure because registers may be live over more iterations of the kernel and need to be buffered.

2.2 Modulo Scheduling

The name software pipelining reminds of the technique of pipelining instructions in RISC processor. This technique splits the execution of instructions into stages and gains efficiency by overlapping some stages of subsequent instructions. Thus starting the execution of an instruction before the preceding one has finished and that way performing some stages of different instructions simultaneously. Software pipelining takes this idea and applies it to loops.

Since programs spend most of their time executing loops much effort goes into their optimization. Optimization for VLIW processors in general means optimal use of ILP. The problem is that schedulers which try to find opportunities for parallelization work better on larger regions of code where they have greater possibilities to find independent operations. A single basic block often is too small to exhibit a lot of parallelism. To find more parallelism many scheduling algorithms in the realm of VLIW consider larger regions like traces [Fis81], superblocks [HMC+95], hyperblocks [MLC+92], and treegions [HBC98].

Profiling data is needed to form the regions, since one tries to optimize the common case, which often leads to penalties for execution paths that are taken less frequently. The penalties arise from compensation code that has to be inserted.

The problem with those region scheduling techniques is that they only work on acyclic regions and thus are not applicable to loops. One approach to enlarge the region for the scheduler in loops is *loop unrolling*. This technique duplicates the loop body a number of times thus allowing the scheduler to operate on a larger block of code. The number of the loop body's duplications is called *unrolling factor*. The scheduling algorithm applied to the enlarged loop body in turn is an acyclic one that does not take recurrences into account. Loop unrolling reduces the overhead caused by a loop since one iteration of the unrolled loop corresponds to more than one iteration of the original loop so less jumps are taken. The duplication of the loop body increases the code size and thus the pressure on the instruction cache. If the number of iterations, the trip count is not divisible by the unrolling factor the original loop body has to be run some iterations until the modulus is 0. This is called *preconditioning*. Due to the duplication of the

loop body some inter iteration dependences become normal dependences inside the basic block thus allowing to schedule the loop such that parts of consecutive iterations overlap and this increases the ILP. We assume a load delay of one cycle in the example showing the principle.

```
s1  .loop:
s2    r1 = port32[r7] // Load operation
s3    r7 = r7 + 4     // Increase array pointer
s4    r2 = r2 + r1
s5    r4 = r4 + 1     // Increase induction variable
s6    port32[r8] = r2 // Store operation
s7    r8 = r8 + 4     // Increase array pointer
s8    if ( r4 != 99 ) jump (.loop)
```

The loop body can be scheduled with an unrolling factor of three such that a part of three consecutive iterations is run in parallel. The example assumes a VLIW processor that can execute up to four operations simultaneously. A conditional jump counts as two operations. There are no other restrictions on the placement of operations. The syntax of the example writes operations that run in parallel—i.e. operations that are part of the same instruction—enclosed between braces and terminated by a semicolon. Nops are left out for clarity [1]. When the read and the definition of one register are part of the same instruction then all reads are finished before the definition takes place.

```
.loop:
//  iteration i         iteration i+1       iteration i+2
{ r1 = port32[r7]  ;                   ;                    ;  ;}
{ r7 = r7 + 4      ;                   ;                    ;  ;}
{ r2 = r2 + r1     ; r1 = port32[r7]   ;                    ;  ;}
{ r4 = r4 + 1      ; r7 = r7 + 4       ;                    ;  ;}
{ port32[r8] = r2  ; r2 = r2 + r1      ; r1 = port32[r7]    ;  ;}
{ r8 = r8 + 4      ; r4 = r4 + 1       ; r7 = r7 + 4        ;  ;}
{                  ; port32[r8] = r2   ; r2 = r2 + r1       ;  ;}
{                  ; r8 = r8 + 4       ; r4 = r4 + 1        ;  ;}
{                  ;                   ; port32[r8] = r2    ;  ;}
{                  ;                   ; r8 = r8 + 4        ;  ;}
{ if ( r4 != 99 ) jump (.loop)         ;                    ;  ;}
```

Listing 2: Unrolled loop

The number of jumps is reduced by the factor three further more, the execution time of three iterations is reduced to 11 opposed to $3 \cdot 7 = 21$ in the original loop due to overlapping of iterations. Two out of three iteration are initiated before the previous has finished. However every third iteration must finish before the next iteration starts. Software pipelining tries to achieve a schedule that continuously overlaps iterations without holes. Moreover the example shows that the trip count

[1] This notation is very similar to the CHILI's assembler language.

has to be a multiple of three because the branch condition is checked only every three iterations. To exploit even more ILP output anti dependences can be resolved with the renaming of registers with the consequence of increased register pressure.

Like software pipelining loop unrolling allows the parallel execution of different loop iterations and may lead to noteworthy speedups. Moreover it is possible to combine both techniques to achieve even higher ILP. Again increased code size may destroy all performance gains due to cache misses.

Software pipelining also analyzes inter iteration dependences to find a compact schedule. This is the reason why the DDG that is used as input for software pipelining is not an acyclic graph but contains recurrences that represent loop carried dependences. The predominant approach to software pipelining is *modulo scheduling*. Modulo scheduling is a family of software pipelining algorithms that try to form a loop without a previous scheduling pass. Another way are code motion techniques that try to improve an existing schedule by moving operations across the back edge of the loop [CLG].

Modulo scheduling tries to find a pattern of operations that is, if executed repeatedly, equivalent to the original loop. This pattern is the steady state or kernel. Every iteration of the kernel initiates a new iteration of the original loop. Once the kernel is found the prolog and epilog can be constructed since they are multiple partial duplications of the kernel. The prolog is necessary preconditioning code to reach the steady state. It starts all iterations that are not started in the kernel. The prolog fills the pipeline. The epilog on the other hand finishes iterations that have not been finished by the kernel yet. It empties the pipeline. The goal is to find a kernel that is as short and as efficiently scheduled as possible. Normally this is done by first estimating a minimum initiation interval (MII), that is a lower bound for all initiation intervals hence a lower bound for all kernel lengths. This estimation has to take into account the available resources and the structure of the DDG, especially the recurrences. If the MII is estimated too low a lot of unnecessary computations will be made during compilation. On the other hand, if the MII is estimated too high there could be a kernel with an II that is lower than the MII thus wasting an opportunity for optimization. In general it is better to underestimate the MII.

Then the algorithm tries to place all operations of the original loop inside the kernel without violating dependences and without demanding more resources than are available. If that fails the II is increased by one and scheduling is tried again. This is repeated at the latest until the II reaches the length of the original loop, because then modulo scheduling cannot provide any improvements and the original loop is used instead.

Analogous to the pipelining technique in processors, loop bodies are split into stages. These stages of subsequent iterations are then executed in parallel. This

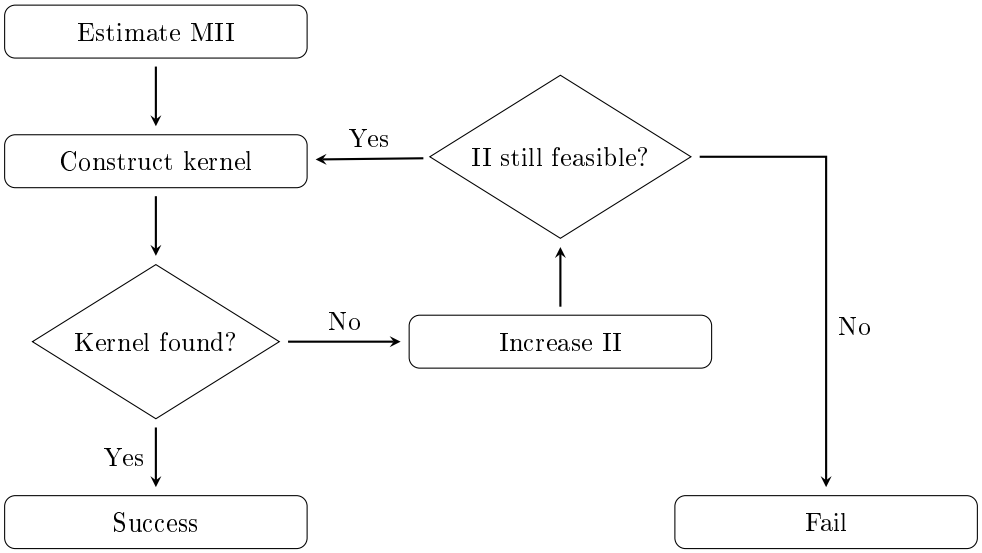

Figure 3: Modulo scheduling strategy [CLG]

is best shown using an example.

In the simplified example of figure 4[2] the loop body consisting of the operations A through I is divided into three stages, the stage count (SC) is three. This way it is possible to overlap the execution of three consecutive iterations of the loop and to exploit ILP. The initiation interval (II) is four thus one iteration of the kernel takes four clock cycles. Even though one stage only consists of two operations that could finish after two cycles it has to be expanded using Nops to also correspond to the II. The stage of three operations is handled in an analogous way.

The simple example assumes that an acyclic scheduler was not able to find any parallelization possibilities which is easily possible if for example every operation depends on the result of the preceding one. Also assuming that the parallelization of the stages does not violate any recurrences and that the machine is able to execute three or more operations in parallel one can see, that the kernel is only four clock cycles long compared to the original loop with a loop body of length 9. Given a sufficiently large trip count (TC) this leads to a great reduction of execution time. Even though the time needed for one iteration in the pipelined loop is greater than in the original, 12 compared to 9 cycles, the throughput of the kernel is greater. This is why the trip count is important. Software pipelining optimizes the throughput of a loop not the time necessary for one iteration. The

[2]This figure is similar to the one used to explain software pipelining in the lecture [BE08].

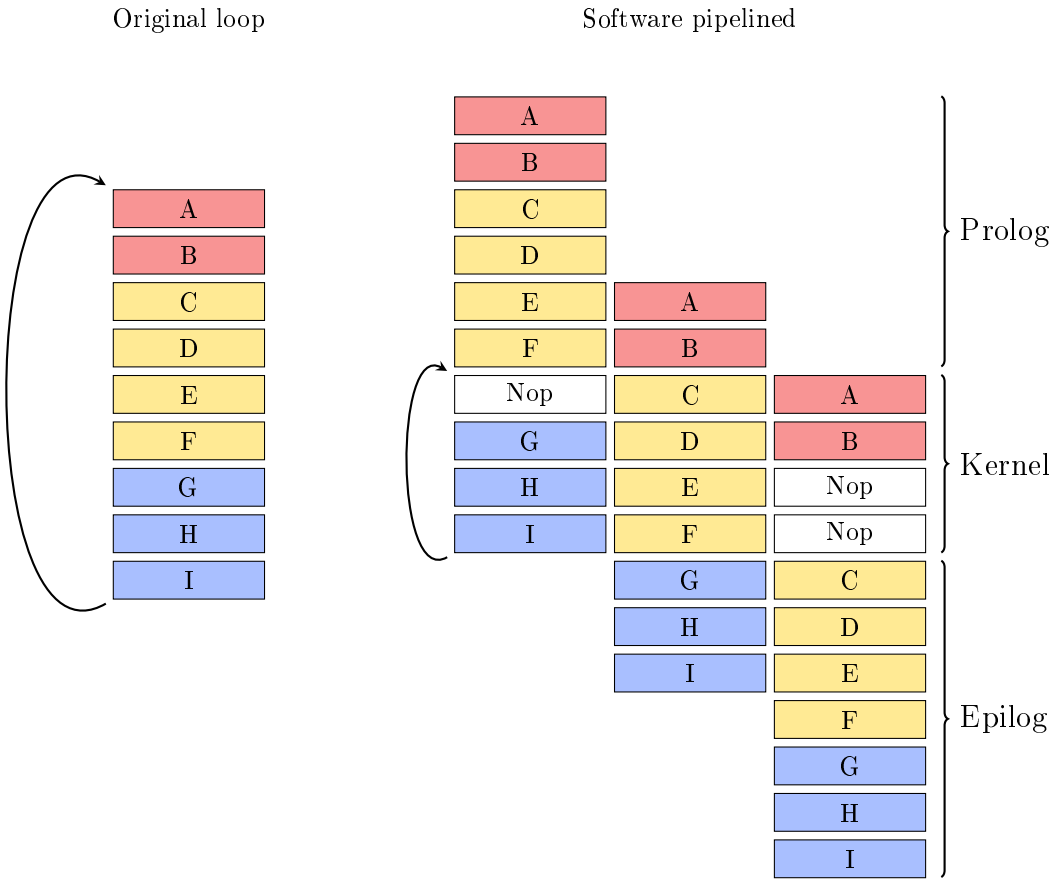

Figure 4: How the iterations are overlapped in modulo scheduling.

total time spent in the software pipelined loop is

$$\underbrace{(SC-1)\cdot II\cdot 2}_{prolog\ and\ epilog}+\underbrace{(TC-SC+1)\cdot II}_{kernel}$$

This formula holds only if the prolog and the epilog are not further optimized by an acyclic scheduler which would be possible as they are formed of straight line code.

If we assume a trip count of 10 iterations the original loop would take

$$10\cdot 9=90$$

cycles to complete. The software pipelined version would only need

$$(3-1)\cdot 4\cdot 2+(10-3+1)\cdot 4=16+32=48$$

cycles.
The loop overhead caused by jumps is not significantly reduced by software pipelining because the number of jumps that are taken by kernel iterations is only reduced by $SC - 1$ compared to the original loop. Normally it is possible to use the delay slot of jump operations for operations of previous stages and that way using the available clock cycles of the delay slot.

Restrictions

The example shows a single basic block loop without control flow. That is, there are no branches or subroutine calls inside the loop body. Most software pipelining algorithms without extensions work on single basic block loops. The same approach was chosen for the scheduler that we implemented for this thesis. Single basic block loops are easily recognized in the control flow graph because every basic block that is the successor of itself is a single basic block loop. Basic block loops are per definition also innermost loops since the basic block does not contain branches or jumps it cannot contain loops.

This restriction disqualifies a lot of loops that can be handled by software pipelining, however many loops that do time-consuming computations are basic block loops and for those software pipelining can bring good improvements.

Furthermore the example shows how the code size is increased which may lead to decreased performance in the instruction cache if the working set of operations does not fit into the cache and cache misses are inevitable. One can see that the code of the loop body exists three times because of the stage count of three. Most of the additional code is contributed by the prolog and the epilog, which both are only executed once. The loop kernel consists of the same operations as the original loop, hence if the iteration count is large enough the effects of the instruction cache decreases as the kernel is not bigger than the original loop. In more realistic examples there would be some additional operations in the kernel, they do not have large consequences on the code size though. The code size multiplies by the stage count compared to the original loop.

A further requirement for the loop to be suitable for software pipelining is that the trip count in general has to be *loop invariant*. That is, the information of how often the loop body is going to be executed has to be available before the loop is entered. The trip count has to be independent of the computation done inside the loop. However, it does not have to be a compile time constant even though a trip count that is known at compile time of course also satisfies the condition.

The reason for this restriction is that the number of kernel iterations is smaller than the trip count of the original loop. Thus the branch condition of the kernel's

branch has to be adapted adequately. Since the kernel runs $SC - 1$ iterations less the branch from the kernel to the epilog has to be taken earlier. If the branch condition is not adapted, that is it remains the same as in the original loop, $SC-1$ iterations are started that would never be executed in the original loop. If those additional iterations cause any side effects, which is normally the case, the result of the loop is altered. This can only be done if all but the final stage do not have any side effects, in this case the epilog can be left out altogether.

Another reason why the trip count has to be known is that it must not be too low. The TC must at least be equal to the SC. The problem is again that iterations would be started that should not be executed. If the TC is small the effects of the increase in code size are predominant and the overhead of prolog and kernel may outweigh the advantages gained by modulo scheduling. For the case that the TC is smaller than the SC the program has to conditionally branch to a not software pipelined version of the loop. Since the SC is not too big a totally unrolled version would be possible. This again contributes to the code size.

Another problem is the increased register pressure. Often more than one copy of a register is necessary to prevent that values are overwritten too early.

This can be shown in another example.

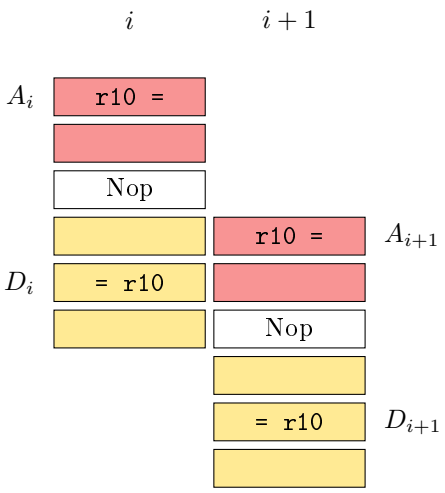

Figure 5: The problem of live ranges.

The modulo scheduled loop has a stage count of two and an initiation interval of three. The problem that is depicted in this example is that the operation D_i of iteration i has to read the register `r10` that is defined by the operation A_i also of iteration i. In this version however, this is not possible because operation A_{i+1} of the next iteration redefines the register `r10` thus overwriting the value A_i is

supposed to read. The live range of `r10` is four thus the redefinition in a later iteration happens before the read of the current iteration.

This situation is not uncommon in modulo scheduling and happens every time a live range of a register exceeds the initiation interval. There is a number of possible solutions to remedy this problem.

One way is to increase the initiation interval by loop unrolling such that no live range of a register is greater than the II. This technique is called *modulo variable expansion*. Again loop unrolling contributes to the code size which is undesirable. Especially in the kernel it is important that the working set of operations fits into the instruction cache. Of course it increases also the number of necessary registers. It also changes the one to one relation that every kernel iteration also initiates one iteration of the original loop. If the trip count is not divisible by the unrolling factor preconditioning is needed. This code again enlarges the code. Also the enlargement of the loop body complicates the DDG the modulo scheduling algorithm has to work with.

The solution chosen in the system we implemented is the introduction of copy operations to break the live ranges of registers into intervals less than or equal to the II. This technique also increases register pressure and needs the necessary resources available for the introduction of the copy operations. The answer to the problem of figure 5 using the copying technique is depicted in figure 6.

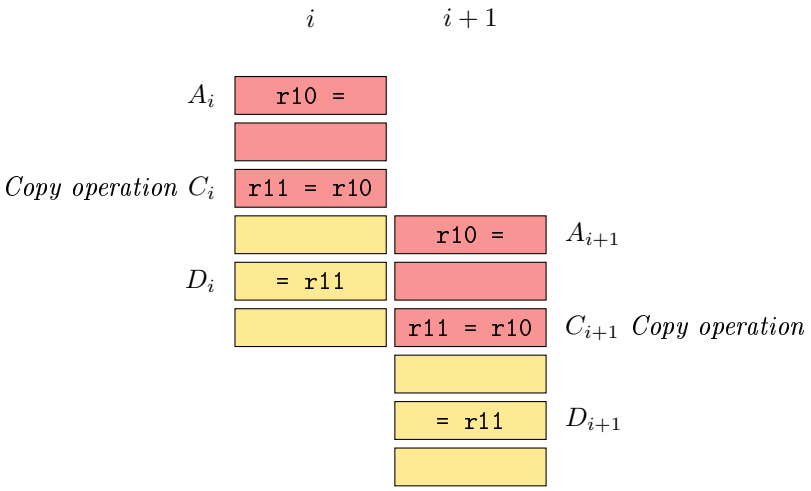

Figure 6: Shortening live ranges with copy operations.

Now the live ranges of `r10` and `r11` are both reduced to 2 which is lower than the II thus no unwanted overwriting occurs. The price we had to pay is that

an additional register was needed increasing register pressure. The number of additional copies of a register with live range LR is at least

$$\left\lfloor \frac{LR}{II} \right\rfloor$$

The copy operations have to be placed in such a way that no live range is greater than the II. If the available resources are not enough for example there is no place to schedule all the copy operations the II has to be increased. Normally this is not necessary since copy operations typically require only one cycle and do not need a lot of resources.

A hardware feature that solves the problem of overlapping live ranges which can be found in some VLIW processors are *rotating register files*. Using rotating register files every register is replaced by an array of registers that can be addressed using a base pointer and an offset. Now renaming does not have to be done by copy operations but by simply augmenting the base pointer. All registers are renamed at once. The reading operation only needs the negative offset, that is the number of IIs that pass between definition and use. At every kernel iteration the base pointer is augmented and no copying and renaming is necessary. Augmenting and offset follow a modulo semantic such that the register array appears like a cyclic buffer.

Rotating register files can also be used to handle trip counts that are *not* loop invariant. It can be used to make register definitions of iterations that should not have been executed "undone" by simply decreasing the base pointer by $SC - 1$. After that the current registers hold the values of the iteration where the branch condition became *true* and the branch was taken. Again this code does not have an epilog and the program is continued right after the kernel. This works only if the operations that are undone cannot cause exceptions since they work with operands that were not intended in the first place. Also unintentional memory store operations are not prevented by rotating register files.

Rotating register files do not increase register pressure from the point of view of register allocation. Of course they add complexity to the hardware of the register file and the number of real physical registers increases. They are only addressed in another way, not just by the register name but by base register and offset. Another disadvantage is that modulo scheduling using rotating register files only works if the stage count is not greater than the number of instances the register file can hold for every register. Normally however the SC is rather small.

If-conversion

One drawback of software pipelining is that it can only handle loops without internal control flow. The technique called *if-conversion* can help to enlarge the

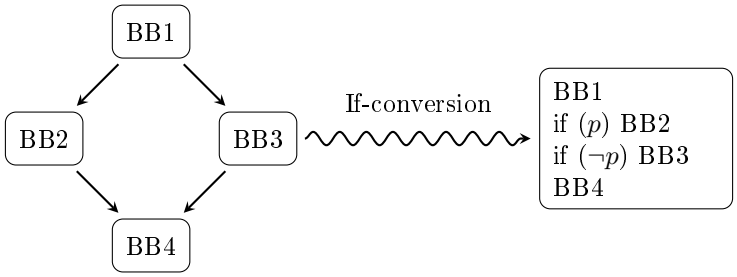

Figure 7: Transformation of the CFG with if-conversion

body of basic block loops or form basic block loops out of code that contains control flow. If-conversion needs a hardware feature called *predicated execution*. Fortunately this is very common in VLIW architectures. Predicated execution means that some or all operations can have an additional predicate as an operand that can be either *true* or *false*. Predicated execution can make the effects of a predicated operation undone or suppress its execution if the predicate turns out to be *false*. This feature can be used to turn control dependences into data dependences and to make larger basic blocks. The simplest version of if-conversion turns diamond like structures in the control flow graph into a single basic block and supplies all operations of the two branches with a predicate.

If-conversion helps not only to find more opportunities for software pipelining but also exposes more ILP for acyclic schedulers because the control flow graph is simplified and the scope examined is increased. On architectures with large jump delay slots it also reduces the number of jumps and helps to create denser schedules. A simple example for if-conversion, assuming a jump delay slot of two cycles:

```
 s1    r1 = r7 - 3
 s2    if ( r1 < 0 ) jump (.labelT)
 s3    Nop
 s4    Nop // last cycle of the delay slot
 s5    r4 = 20
 s6    jump (.labelExit)
 s7    Nop
 s8    Nop // last cycle of the delay slot
 s9  .labelT:
s10    r4 = 10
s11  .labelExit
s12    r5 = r4 + 30
```

Listing 3: Without if-conversion

```
 s1    r1 = r7 - 3
 s2    if ( r1 < 0 ) r4 = 10 // conditional move
```

```
s3    if ( r1 >= 0) r4 = 20
s4    r5 = r4 + 30
```

Listing 4: If-converted

The *true block* and the *false* block have been merged into one thus eliminating control flow and reducing the effects of the jump delay slot. The construction algorithm for the DDG has to be aware of predication and has to include the additional edges. In the example of listing 4 there must be RAW edges from both $s1$ and $s2$ to $s3$. It is obvious that if conversion should be applied before software pipelining to exploit its benefits. On many architectures the predicate can be computed independently from the predicated operation offering even more freedom for scheduling. On the other hand full predication is rather uncommon and normally only some operations offer a predicated version.

2.3 Comparison to other Modulo Scheduling techniques

Since optimal modulo scheduling is not feasible due to its complexity and long compile time many heuristics have been developed. In general they follow the strategy depicted in figure 3 however they differ in various ways:

- The method the operations are placed in the schedule.
- The method register pressure is reduced.
- The order in which operations are put into the schedule.
- Whether an operation can be unscheduled and rescheduled, i.e. backtracking is applied.

Iterative Modulo Scheduling

Iterative modulo scheduling (IMS) was proposed in [Rau94]. Operations are put into the partial schedule taking into account the operations that are already in the partial schedule, the height in the DDG, and the recurrences they belong to. The operations are scheduled always as early as possible. If no feasible time slot is found some operations are taken out of the partial schedule and will be rescheduled later, i.e. backtracking occurs.

IMS does not feature any heuristics to reduce register pressure. A comparative study [CLG] showed that IMS produces schedules with a small II, however register requirements and compilation time is higher compared to other techniques.

Slack Modulo Scheduling

Slack modulo scheduling was presented in [Huf93]. The terms "slack" means the degree of freedom an operation offers to the scheduler depending on the predecessors and the successors already placed in the partial schedule. The priority function for the operation selection is based on the slack. Like swing modulo scheduling—the technique used in our system described in section 2.4—slack modulo scheduling has a bidirectional approach for placing operations into the partial schedule. Depending on predecessors and successors a heuristic determines whether an operation is scheduled as early as possible or as late as possible. This measure helps to reduce the live intervals of registers and thus register pressure. If no place is found for an operation the algorithm uses limited backtracking.

The study [CLG] showed that slack modulo scheduling leads to schedules with low register requirements however compared to other techniques the II tends to be longer.

Integrated Register Sensitive Iterative Software Pipelining

Integrated Register Sensitive Iterative Software Pipelining (IRIS) tries to keep the the II low while minimizing the register requirements. Like slack modulo scheduling it uses a combined bottom-up and top-down approach for placing the operations to keep register pressure low. The priority function used for the node selection is the same height-based as in IMS. Like IMS backtracking is also applied in IRIS.

According to [CLG] IRIS produces schedules of good schedules at the cost of increased compilation time and register requirements.

2.4 Swing Modulo Scheduling

Swing modulo scheduling (SMS) is a modulo scheduling technique that focuses on reducing unnecessary register pressure while generating near optimal schedules [Llo96] in terms of initiation interval and stage count. It is a heuristic approach with fairly low computational cost keeping compilation time low. Like all modulo scheduling algorithms it constructs the kernel of the loop pipelined schedule not depending on previous scheduling. One distinguishing feature of SMS is its awareness of register live ranges. This is attained by dealing with the most critical recurrences in the DDG earlier and scheduling dependant operations closely together. Unlike other modulo scheduling algorithms SMS works without backtracking, that is no scheduling decision is ever undone. This helps to reduce computational complexity. Before scheduling the order in witch the operations are scheduled is determined. This order is of essential importance since every scheduled operation also reserves the necessary resources that are then no more available. If scheduling is not pos-

sible due to resource conflicts the II is incremented by one like in other modulo scheduling algorithms. The scheduling order of the operations is independent from the II thus it needs to be computed only once in advance which also contributes to reducing computational complexity. The basic steps of SMS are

- Computation and analysis of the DDG
- Ordering of the nodes
- Scheduling
- Construction of prolog, kernel, and epilog.

The complexity of the algorithm is strongly influenced by the structure of the DDG, because to determine the scheduling order the DDG has to be inspected for recurrences. Every recurrence can contribute to the II. The name *swing* is drawn from the fact that the ordering of the nodes is done in alternating up and down sweeps across the DDG. Also the scheduling itself is done in corresponding up and down movements. We felt that SMS accommodates the CHILI architecture which does not offer rotating register files due to the reduction of necessary copy operations which can reduce the quality of the schedule.

3 The LLVM Compiler Infrastructure

LLVM stands for *low level virtual machine*. It is a compiler infrastructure that allows for multi-stage optimization of programs [Lat02]. Multi-stage means that it is not only a static compiler but also features link time optimization and optimization at run time. The advantage is that also profiling information gathered during the use of the program can be incorporated into the optimizations. For optimizations that are too complex to be carried out at runtime the possibility of re-optimizing a program off-line exists, for example when the user's computer would be idle. However we made only use of the static compiler to implement modulo scheduling. The basic steps in the static compiler are the following.

A front end translates the source language into LLVM's intermediate representation language. This intermediate representation is target and source language independent. Most of the optimizations incorporated in LLVM are carried out on this intermediate representation. After that, the intermediate representation is mapped to a machine *dependent* representation that resembles the actual machine language and uses almost only operations of the target machine's instruction set. The modulo scheduler works on this representation of the program. Finally a printing pass generates the assembler files.

LLVM optimizations and analyzes are implemented as passes regarding basic blocks or functions. Since many passes depend on each other LLVM takes care that the individual passes are run in the right order. It also makes sure that information that is invalidated by a pass is updated timely before it is needed again. Every pass can specify which information it needs and which information it invalidates.

3.1 Intermediate representation

The machine independent intermediate representation used by LLVM is inspired by assembly languages used for RISC machines. Most operations are in three address code, i.e. they have two reading operands and one result. Memory accesses are done explicitly by load and store operations. LLVM provides an infinite number of typed virtual registers. Operations are polymorphic such that a single operation can work with various operand types. For example floating point multiplication has the same opcode as integer multiplication. The semantics of an operation is determined by its opcode and the type of its operands [Lat02].

A distinguishing feature of LLVM's intermediate representation is the use of *static single assignment form (SSA form)* for its code representation. A program is in SSA form if every virtual register is defined exactly *once* before it is read. This simplifies dependence analysis and other analysis and optimization passes. As a consequence every operation that defines a register creates a new virtual register.

To handle control flow special operations are used, so called phi nodes. Phi nodes reside at the beginning of every basic block that has more than one predecessor in the control flow graph. Every phi node has as many reading operands as the block has predecessors. The phi nodes select the incoming values according to the predecessor the control flow comes from and assign them to a new virtual register.

SSA form is kept up until register allocation. After register allocation only the limited number of physical registers is available making redefinitions of registers inevitable. Even after instruction selection the program is still in SSA form. After instruction selection the program is in a target dependent form containing operations of the target language, however phi nodes are still in the program.

This is the representation the modulo scheduler works on.

3.2 Extended hardware description for VLIW

Unfortunately LLVM does not include explicit support for VLIW architectures. There is no notion of parallel execution. Thus there is no way to tell the scheduler that an operation can be scheduled in the same cycle as another one even if the necessary resources are available. The time slot acts like a unique resource that can only be reserved by one operation. In VLIW architectures resources are typically duplicated a number of times. For example there can be two load-store units. That means a load operation can either use the first or the second load-store unit, one can say there are two versions of the load operation.

In LLVM every operation has exactly one version. We solved this problem by supplying a resource model independent from LLVM's that features all the characteristics needed. The modulo scheduler depends only on this model. To specify parallelism we included a special operation that does nothing but signify the end of a VLIW instruction. That is for example, four operations enclose between two of the special operations are meant to be executed in parallel.

4 Implementation

The modulo scheduler is implemented as a so called "machine function pass" in the LLVM infrastructure. I.e. the pass works on the scope of the whole function. Nevertheless it schedules basic blocks only. The pass is located before register allocation such that dependences that can by avoided by register renaming do not occur, thus allowing to exploit more parallelism. I.e. anti and output dependences appear only between memory accesses. Those dependences cannot happen between register operands since the program in this intermediate representation is still in SSA form thus every register is defined only once. SSA form is destroyed only shortly before register allocation therefore the modulo scheduler has to make sure that the program is still in SSA form after it has finished. Many subsequent passes depend on the program being in SSA form and would not work otherwise. For example the liveness analysis that follows directly needs a program in SSA form.

We do this by first substituting all phi operations by copy operations, hence destroying SSA form. The copy operations define the same registers as the phi operations, but the operand that is copied corresponds to the phi node's back edge operand. This is possible since the kernel has exactly two predecessors namely the prolog and the kernel. Then the modulo scheduling is done like on any other basic block, not depending on SSA form. The description of the algorithm in [Llo96] does not use SSA form. Finally SSA form is regained by inserting phi operations and renaming of operands. The modulo scheduling pass tries to postpone any transformations and changes to the original function as long as possible to only invalidate informations of previous analysis passes if it is sure that modulo scheduling is possible. Otherwise it does not alter the function and the control flow graph thus ensuring that computations do not have to be repeated unnecessarily.

Curtailments

As mentioned in 3.2 LLVM does not have any notion of parallelism thus we encountered difficulties to do the whole modulo scheduling before register allocation because many passes run afterwards. It was also impossible to schedule the branch instruction the way it would be in the target program, namely also accounting for the delay slot. In an LLVM program the branch operation has always to be the last one in the basic block.

The solution was to output the operations the modulo scheduler issued to one VLIW instruction in sequential order. Thanks to the SSA form no unnecessary dependences are introduced. After register allocation the basic blocks are scheduled by the normal list scheduler, hence the modulo scheduling pass can be seen as a preconditioning for the acyclic scheduler. The schedules produced are different from the ones found in the modulo scheduling pass, however similar in quality

since the preconditioning exposes more ILP to the acyclic list scheduler. As soon as LLVM also explicitly supports VLIW architectures the pass can run without the help of the list scheduler.

We also implemented a version of the modulo scheduler that runs after register allocation, when the intermediate representation is not in SSA form anymore, the algorithm does not depend on the SSA form anyway. The disadvantage is that WAW and WAR dependences have been introduced by the register allocator, the advantage is that the modulo scheduler can run *instead* of the list scheduler for modulo schedulable loops, hence it outputs the exact schedule it finds, including instructions and delay slots. Another advantage is that in our system the if-conversion pass runs also after register allocation making more candidates for modulo scheduling available.

4.1 Finding suitable basic blocks

Swing modulo scheduling works on single basic block loops. In the control flow graph those basic blocks are immediate successors of themselves, i.e. they have a back edge that is both an outgoing edge and an ingoing one.

The problem is that LLVM generates programs where this kind of basic blocks is very rare. LLVM substitutes the direct back edge by an additional jump operation which forms a new basic block such that the loop consists of two basic blocks instead of one. This is called to *break a critical edge* and is useful for other passes. However it prevents modulo scheduling if not some preconditioning is done. To make the loop suitable for modulo scheduling the CFG has to be changed slightly.

First the CFG is examined whether the basic bock in question has exactly two successors. This can be done by using the control flow information LLVM provides. If one of the successors is the basic block itself no further transformations are necessary. Otherwise if one of the successor basic blocks only consists of a jump back to the top of the loop it is possible that a single basic block loop can be formed by some transformations. Furthermore the successor block consisting of a single jump operation must have exactly one predecessor and exactly one successor which is the loop body. If these conditions are met a single basic block loop can be constructed.

There are two kinds of loops that can be found in many LLVM programs which are disguised single basic block loops. One case is when the fall through path is only taken at loop exit. Then the back edge consists of a jump out of the loop and a jump back to the top.

This can be repaired by simply exchanging the jump target of the loop's branch condition to the label corresponding to the loop instead of the intermediate jump. The basic block that consists of a single jump operation can be safely deleted

afterwards as it is not reachable anymore due to the fact that it only had the loop body as a predecessor. This situation is shown in figure 8.

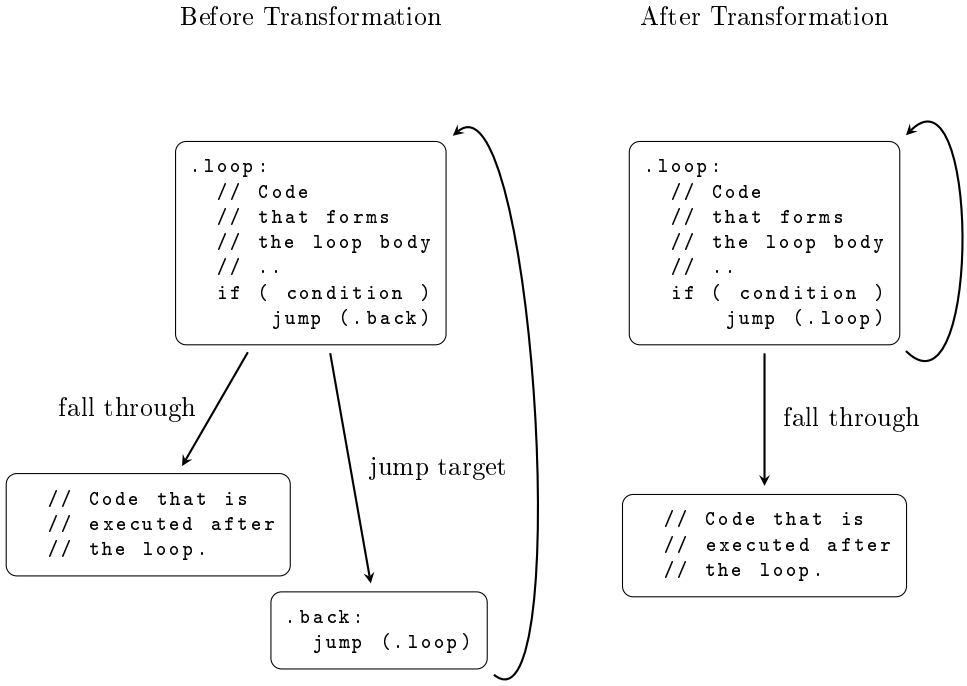

Figure 8: Broken back edge in the CFG with jump

The slightly more complicated case is when the loop's branch condition is met it does not jump back to the head of the loop but out of the loop. That means that the fall through path is taken in every iteration except the last one. The fall through block in this case must be the one with the single jump back to the loop. Therefore the condition of the loop's branch has to be negated, which is possible with help of a LLVM function, and the jump target has to be set to be the loop. Furthermore the jump target of the fall through block has to be set to the original jump target of the loop. The single jump operation can not simply be deleted because it is not sure, that the exit block is placed right after the loop in the program. This situation and its solution is shown in figure 9.

Then LLVM's CFG has to be updated accordingly. One draw back is that these transformations can invalidate some earlier analysis passes even if it turns out that modulo scheduling is not possible at all.

After ensuring that the loop is a single basic block loop, some other conditions have to be met for the loop to be suitable for modulo scheduling. The loop body must not contain inline assembly code or function calls. In general it must only

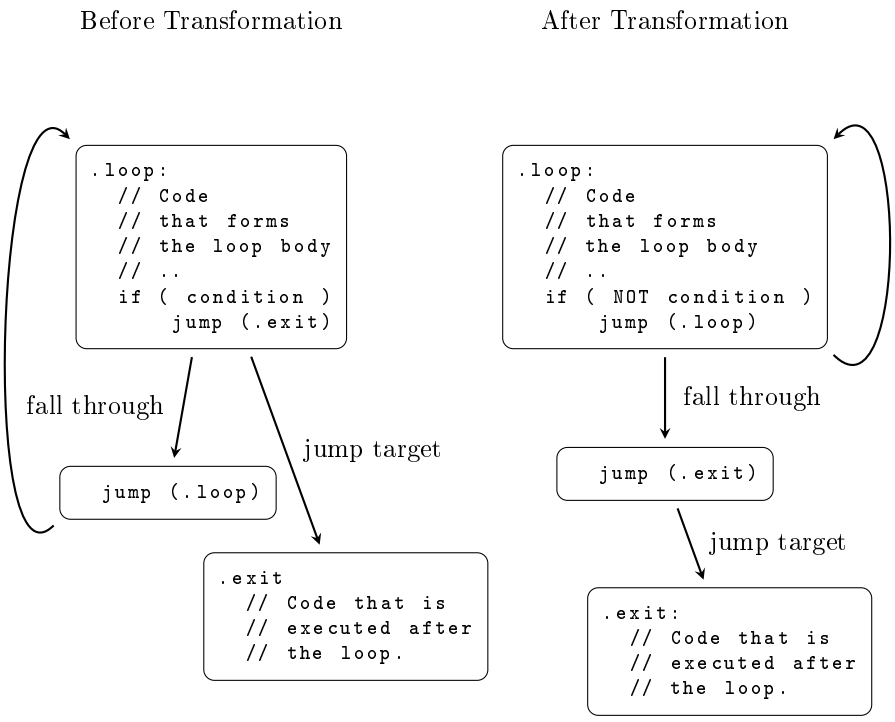

Figure 9: Broken back edge in the CFG with fall through block

contain one branch operation at the end. For modulo scheduling the trip count must be loop invariant. This is checked in the branch condition. All operands but the induction variable have to be loop invariant. The easiest case is when the induction variable is simply compared to an immediate constant. Thanks to LLVM's standard transformation of loops one can assume, that the induction variable is a register which is incremented once every iteration by one. This is important when we have to update the branch condition. Only loops with exactly one branch operation at the end that is associated with the trip count can be handled. Further more it is checked if the loop is an innermost loop additionally by using LLVM's loop analysis information. If all these conditions are met the loop can be modulo scheduled. The scheduling passes of the loops are independent from each other so we save every suitable basic block loop in a work list and schedule one after the other.

4.2 Computation of the dependence graph

Since we do not know in advance whether modulo scheduling will be successful the whole algorithm will be carried out on a copy of the basic block which can be discarded if the algorithm fails to find a valid schedule. This working copy is

changed from the original loop in such a way that all phi operations are substituted by copy operations. The registers which are defined are the same as the registers that are defined by the phi operations, the register operands read are the ones corresponding to the back edge of the phi operations. This is necessary as phi operations have to be scheduled at the beginning of a basic block in LLVM. However the modulo scheduling algorithm regards phi operations as normal operations and would not necessarily schedule them at the beginning of the basic block.

After this step the program is not in SSA form anymore, nevertheless we can still be sure that every register is only defined exactly once inside the loop body. SSA form will be regained after modulo scheduling was successful by renaming of operands and insertion of phi operations.

The dependence analysis which constructs the DDG works with this transformed loop. Our straight forward implementation of the dependence analysis which also finds loop carried dependences consists of two backward passes over the loop body. It is a slight adaption of the very common DDG construction algorithm found in acyclic schedulers. This analysis conservatively sets all iteration distances greater than 0 to one. To exploit more ILP further analysis of memory accesses would be necessary. It collects all definitions and uses of registers and memory accesses and inserts the adequate edges. Since every register is only defined once there can only be RAW dependences between operations that only have register operands. This helps to keep the DDG simple. Memory accesses can have also WAR and WAW dependences if one of the operations involved is a store operation. The result of the algorithm is a directed graph where every node corresponds to an operation of the program. Edges represent a dependence and are labeled with the iteration distance and the type of the dependence—RAW, WAW or WAR. The dependence type is important to compute the minimum delay of the depending operations. Dependences that do not cross iteration boundaries have a distance of 0, these are *intra* iteration dependences.

4.2.1 Loop carried dependences

Loop carried dependences also known as *recurrences* emerge when a value defined in a former iteration is read or its register of memory cell is overwritten. This is very common in loops, for example if the sum of all array entries is computed the intermediate results are read and updated iteratively. The iteration distance between register definitions and uses can at most be one, since the register would be overwritten in every iteration.

Memory accesses can have iteration distances greater than one as it is possible to read for example a memory cell that was defined more than one iteration ago which would result in a RAW dependence. To find out such connections it is necessary to analyze the memory access behavior of a loop, how array pointers are moved

```
for dst ← 0 to 1 do
    forall instructions i in reverse order do
        U ← uses of i;
        D ← definitions of i;
        // insert the edges
        forall d ∈ D do
            if def[d] ∧ d is not a register operand then
                insert WAW edge from i to def[d] with distance dst;
            end
            forall p ∈ uses[d] do
                insert RAW edge from i to p with distance dst;
            end
        end
        forall u ∈ U do
            if def[u] ∧ u is not a register operand then
                insert WAW edge from i to def[u] with distance dst;
            end
        end
        // update the data structures
        forall d ∈ D do
            if dst = 0 then
                def[d] ← i;
            else
                def[d] ← undefined;
            end
            uses[d] ← ∅;
        end
        if dst = 0 then
            forall u ∈ U do
                insert i into uses[u];
            end
        end
    end
end
```

Algorithm 1: Construction of the DDG

and other aliasing information. This is not implemented in this system, and thus recurrences in the DDG all have iteration distance one, which is a conservative assumption. It is save to underestimate the iteration distance since a value that was defined more than one iteration ago is still defined one iteration ago. It can result in a loss of ILP, however iteration distances of one are found very frequently in programs and thus we believe the effect is not very large. Furthermore the ILP is determined mostly by the smallest iteration distance of any recurrence. Edges with an iteration distance do not necessarily have to form recurrences, but those edges do not impose any constraints on the ILP.

The DDG construction algorithm considers recurrences in its second pass. In the second pass the data structures are updated such that only dependences are found that range across the iteration boundary.

4.3 Minimum initiation interval

Once the DDG is constructed a starting point for the modulo scheduling algorithm can be computed. The MII[Rau94] is a lower bound for the II and in turn the first length of a kernel that is tried for scheduling. The MII is affected by two factors: the available resources and the most critical recurrence in the DDG.

The resource minimum initiation interval (ResMII) is limited by the resource usage of the loop to be scheduled and the resources available in the processor like arithmetic-logic units or load-store units. It is obvious that even if all data dependences would allow it, parallelization is only possible if there are enough resources available necessary for the operations executed in parallel.

The recurrence minimum initiation interval (RecMII) is the number of cycles that is necessary to satisfy all loop carried dependences between iterations. For example if an operation reads a value defined in the previous iteration the reading operation cannot be scheduled before the computation of the value has finished. This form of RAW dependence frequently arises from operations with a high latency, like load operations.

The MII must not be smaller than either the ResMII or the RecMII thus

$$MII = max(RecMII, ResMII)$$

To compute the exact value of the MII requires a lot of complex computation and thus one settles with an estimation that tries to be as close as possible. If the MII is overestimated the modulo may find a solution faster which results in a reduced compilation time. It may, however, generate kernel that is larger than it needs to be resulting in wasted performance. On the other hand an underestimated MII leads to more futile scheduling attempts which results in increased compilation time. We feel that in SMS underestimating the MII is better since the algorithm

itself is not computationally complex and it does not incorporate backtracking. Thus some failed scheduling attempts are acceptable while finding the smallest possible kernel.

4.3.1 Resource MII

To compute the ResMII a description of the available resources of the machine and a description of the resource usage of each operation of the loop is necessary. A simple estimation of the resources that the loop body needs is to sum up all resource claims of all operations and divide it by the total number of available resources. This is the way chosen in our implementation since the CHILI provides every functional unit four times in parallel. To refine the estimation it is possible to separate the needed resources by their type, for example ALUs or load-store units. Then the most critical resource type determines the ResMII.

$$ResMII = max_{t \in T} \left(\left\lceil \frac{\sum_{i \in P} needs(i,t)}{number(t)} \right\rceil \right)$$

Where T is the set of different resource types, P the set of operations forming the program, $needs(i,t)$ the number of cycles operation i needs resource of type t and $number(t)$ the number of instances of a resource of type t.

4.3.2 Recurrence MII

A circuit in the DDG signifies that an operation depends on the invocation of the same operation in an earlier iteration. In this circuit there must be at least one edge with a distance greater than 0. The distinction between the terms *latency* and *delay* and the description of the term *elementary circuit* is necessary.

Latency is the number of clock cycles an operation needs before its result is available. The latency of an operation o is written $latency(o)$ or λ_o.

Delay is the minimum number of clock cycles that must pass between two dependent operations. When the operation *succ* depends on the operation *pred*, that is there exists an edge in the DDG pointing from *pred* to *succ*, at least $delay(pred, succ)$ clock cycles have to pass between *pred* and *succ*. The delay depends on the type of dependence.

- RAW: $latency(pred)$
- WAR: $1 - latency(succ)$
- WAW: $1 + latency(pred) - latency(succ)$

Theses formulae show that for WAR and WAW dependences the delay also can be negative, which means that the depending operation actually can be scheduled before the one it depends on.

Elementary circuit in an directed graph is a path through the graph that starts and ends in the same vertex and which does not visit any vertex on the circuit more than once [Rau94].

The clock cycles that pass between the two invocations of the operation must be long enough to meet all delays along every elementary circuit that can be found in the DDG. This is expressed in the following inequation.

Let c be an elementary circuit of the DDG then $delay(c)$ is the sum of all delays along c. Further let $distance(c)$ be the sum of all distances along c.

$$delay(c) - II \cdot distance(c) \leq 0$$

must be true for all elementary circuits c [Rau94]. Hence the initiation interval has to be chosen accordingly. Using this inequation and the set of all elementary circuits C the RecMII can be expressed as

$$RecMII = max_{c \in C} \left(\left\lceil \frac{delay(c)}{distance(c)} \right\rceil \right)$$

The DDG has to be analyzed to find all elementary circuits. The MII of every recurrence is computed and the recurrences are saved in a sorted list from high MII to low MII. The recurrence responsible for the RecMII, that is the most critical recurrence, corresponds to the critical path that is given priority in acyclic schedulers. In SMS the recurrence is prioritized accordingly. Hence SMS seems promising compared to acyclic scheduling whenever the critical path is long compared to the MII.

4.4 Computation of node properties

To determine the order in which the operations are scheduled, some properties of every node have to be computed according to their position in the DDG. All these properties have to be computed only once and can be used at every scheduling attempt.

- $\delta_{u,v}$ is the iteration distance on the DDG edge from the node u to the node v. It means that the operation v of iteration I depends on operation u of iteration $I - \delta_{u,v}$.
- λ_v describes the latency of operation v.

- $Suc(v)$ is the set of all direct successors of v in the DDG.
- $Pred(v)$ analogously is the set of all predecessors of v in the DDG.

For every operation in the DDG five properties are computed. To avoid cycles during the computation of the node properties one back edge of every recurrence is ignored, thus gaining an acyclic graph. The properties are computed as follows:

- $ASAP_u$ stands for "as soon as possible" and defines the earliest clock cycle the operation u can possibly be scheduled.

  if $Pred(u) = \emptyset$ **then**
  $ASAP_u \leftarrow 0$;
  else
  $ASAP_u \leftarrow max(ASAP_v + \lambda_v - \delta_{v,u} \cdot MII) \forall v \in Pred(u)$;
  end

- $ALAP_u$ stands for "as late as possible" and defines the latest clock cycle the operation u can possibly be scheduled.

  if $Suc(u) = \emptyset$ **then**
  $ALAP_u \leftarrow max(ASAP_v) \forall v \in Suc(u)$;
  else
  $ALAP_u \leftarrow min(ALAP_v - \lambda_v - \delta_{u,v} \cdot MII) \forall v \in Suc(u)$;
  end

- MOV_u is the mobility of operation u. It describes the number of clock cycles u can be scheduled in, that is the freedom the scheduler has to put the operation into the schedule. The more critical the operation is, the smaller is the value.

  $MOV_u \leftarrow ALAP_u - ASAP_u$;

- D_u is called the depth of operation u in the DDG. It depends on how far the operation is from the top of the DDG.

  if $Pred(u) = \emptyset$ **then**
  $D_u \leftarrow 0$;
  else
  $D_u \leftarrow max(D_v + \lambda_v) \forall v \in Pred(u)$;
  end

- H_u is called the height of operation u in the DDG. It depends on how far the operation is from the bottom of the DDG.

if $Suc(u) = \emptyset$ **then**
    $H_u \leftarrow 0$;
else
    $H_u \leftarrow max(H_v + \lambda_u) \forall v \in Pred(u)$;
end

4.5 Ordering of the nodes

One of the most distinguishing features of SMS compared to other software pipelining techniques is how the order in which the operations are put into the schedule is determined. The ordering algorithm produces a list of operations. The scheduling algorithm takes this list and tries to find a suitable time slot for every operation starting with the first one in the list. Obviously the more operations that are in the partial schedule the more difficult it becomes to find an appropriate time slot since every operation creates new scheduling constraints that have to be met. This is why operations with less freedom for the scheduler are placed more towards the front of the list such that they are handled earlier than operations with more freedom. The ordering of nodes tries to achieve two goals.

It tries to give priority to operations with a low MOV value, that is operations that give the scheduler less freedom to find an appropriate time slot. This helps to keep the II and the stage count low.

The other goal is reducing register pressure by keeping live ranges low. This is achieved by scheduling each operation close to its predecessors and successors. To make this possible the operations are ordered such that no operation is put into the schedule after both its predecessors and successors. Recurrences obviously violate this principle since one operation of every recurrence has to be handled after successors and predecessors.

The ordering algorithm consists of two parts. The first one creates a partial order, that is a list of mutually disjunct sets of operations, to give priority to the more critical recurrences. Hence it addresses primarily the first goal of the ordering. The first set of the partial order is the most critical recurrence, that is the one with the highest RecMII. If there are recurrences with the same RecMII chose any. The second set in the list contains the second critical recurrence and all nodes that lie on a path between this recurrence an any node already in the partial order. This is repeated until all recurrences are in the partial order. The operations not handled so far are grouped by their connected components and appended to the partial order. The pseudo code of this procedure can be found in [Lat05], $L \mid v$ appends a value v to a list L.

$P \leftarrow$ Empty list of sets of operations;
$RL \leftarrow$ All recurrences;
while $RL \neq \emptyset$ **do**
    $R \leftarrow$ Recurrence with highest RecMII $\in RL$;
    $RL \leftarrow RL \setminus R$;
    if $P = \emptyset$ **then**
        $P \leftarrow P \mid R$;
    else
        $A \leftarrow$ all operations that are already in any set of P;
        $R \leftarrow R \setminus A$;
        $C \leftarrow$ all operations that lie on a connecting path between R and A;
        $P \leftarrow P \mid (R \cup C)$;
    end
end
$N \leftarrow$ operations that are not already in any set of P;
forall *connected components* $C \in N$ **do**
    $P \leftarrow P \mid C$;
end

Algorithm 2: Partial ordering of operations

The second part creates the final order of operations placing adjacent operations of the DDG next to each other in the list. It takes the partial order of the first part ordering one set after the other. Hence enforcing the second goal of the ordering to place an operation close to its predecessor respectively successor to keep live ranges short.

The algorithm starts with the node with the highest $ASAP$ value of the set with the highest priority. It then visits all ancestors in an bottom-up manner, nodes with higher depth have priority over nodes with lower depth. If two nodes with equal depth are found the one with less mobility is chosen. Once all ancestors are in the list the direction is switched to top-down and the descendants of the already inserted nodes are visited. Now the height determines the priority. These upward-downward sweeps are repeated until all nodes of the set are in the list. Then the next set is dealt with in the same way. The initial direction depends on whether there are predecessors or successors already in the list. This is repeated until all nodes of all sets are in the list. The sets $Pred\_L(O)$ and $Suc\_L(O)$ are defined as follows:

$$Pred\_L(O) = \{v \mid \exists u \in O \text{ where } v \in Pred(u) \text{ and } v \notin O\}$$

$$Suc\_L(O) = \{v \mid \exists u \in O \text{ where } v \in Suc(u) \text{ and } v \notin O\}$$

The pseudo code can be found in [Llo96]. For every recurrence one back edge is

ignored to gain an acyclic graph, similar to the computation of the node properties. Like the node attributes the order of the nodes remains the same throughout every scheduling attempt, thus it only has to be computed once.

$P \leftarrow$ partial order of nodes;
$O \leftarrow$ Empty list of operations;
foreach *set of nodes* $S \in P$ **do**
  if $Pred\_L(O) \neq \emptyset$ *and* $Pred\_L(O) \subseteq S$ **then**
    $R \leftarrow Pred\_L(O) \cap S$;
    order $\leftarrow$ bottom-up;
  else if $Suc\_L(O) \neq \emptyset$ *and* $Suc\_L(O) \subseteq S$ **then**
    $R \leftarrow Suc\_L(O) \cap S$;
    order $\leftarrow$ top-down;
  else
    $R \leftarrow$ {One node with highest $ASAP \in S$};
    order $\leftarrow$ bottom-up;
  end
  repeat
    if *order* = *top-down* **then**
      while $R \neq \emptyset$ **do**
        $v \leftarrow$ Element of R with highest H_v
        if more than one, choose node with lowest MOV_u;
        $O \leftarrow O \mid v$;
        $R \leftarrow (R \setminus \{v\}) \cup (Suc(v) \cap S)$;
      end
      order $\leftarrow$ bottom-up;
      $R \leftarrow Pred\_L(O) \cap S$;
    else
      while $R \neq \emptyset$ **do**
        $v \leftarrow$ Element of R with highest D_v
        if more than one, choose node with lowest MOV_u;
        $O \leftarrow O \mid v$;
        $R \leftarrow (R \setminus \{v\}) \cup (Pred(v) \cap S)$;
      end
      order $\leftarrow$ top-down;
      $R \leftarrow Suc\_L(O) \cap S$;
    end
  until $R = \emptyset$;
end

Algorithm 3: Final ordering of operations

4.6 Scheduling

The actual scheduling phase examines the operations in the order computed in the previous step and tries to find a time slot depending on the available resources and the nodes that are already in the partial schedule. One operation after the other is inserted until either all operations are scheduled or we encounter a situation where one operation cannot be scheduled without violating a dependence. In this case we increase the II, clear the partial schedule and start anew, as depicted in figure 3. No backtracking is made, i.e. no scheduling decision is ever undone.

The reservation of the resources are entered in the so called *modulo reservation table*. Like a reservation table for acyclic scheduling it holds entries for the resource requirements of all operations in the partial schedule and when they are needed. However the resources are reserved not only for the stage the operation is scheduled in but for *all* stages. The resource needs are computed modulo the II, hence the name.

The algorithm tries to schedule adjacent operations of the DDG as closely as possible together, thus the search direction for a free time slot depends on which neighbors have already been scheduled. There are four situations that can occur:

- If an operation u has neither a predecessor nor a successor in the partial schedule the scheduler scans the partial schedule in *forward* direction from $Early\_start_u$ to $Early\_start_u + II - 1$ in order to find a free time slot. Where $Early\_start_u = ASAP_u$.

  We observed that this step also produces valid schedule if the search is carried out in *backward* direction, i.e. from $Early\_start_u + II - 1$ to $Early\_start_u$. This version normally works better if the modulo scheduling is done as a preconditioning for the acyclic scheduler. Otherwise it does not degrade the result.

- If an operation u has only predecessors in the partial schedule it is scheduled as soon as possible in *forward* direction from $Early\_start_u$ to $Early\_start_u + II - 1$.

  $Early\_start_u = max_{v \in PSP(u)}(t_v + \lambda_v - \delta_{v,u} \cdot II)$ where $PSP(u)$ is the set of predecessors of u that are already in the partial schedule, t_v is the time slot where v is scheduled, λ_v is the latency of v and $\delta_{v,u}$ is the iteration distance from v to u.

- Analogously if an operation u has only successors in the partial schedule it is scheduled as late as possible in *backward* direction from $Late\_start_u$ until $Late\_start_u - II + 1$.

$Late\_start_u = min_{v \in PSS(u)}(t_v - \lambda_u + \delta_{u,v} \cdot II)$ where $PSS(u)$ is the set of all successors of u that are already in the partial schedule.

- If an operation u has both predecessors and successors in the partial schedule the scheduler searches for a free time slot in *forward* direction from $Eartly\_start_u$ until $min(Late\_start_u, Early\_start_u + II - 1)$. This can only happen for one node in each recurrence.

  In our implementation we use a slight modification where the search direction is the same as the search direction of the last node that was inserted.

The absolute values for the time slot t_u that are used in the algorithm like $ASAP$, $Early\_start$ and $Late\_start$ can be expressed in terms of stages and cycles as follows:

$$cycle_u = t_u \; mod \; II$$

$$stage_u = \left\lfloor \frac{t_u}{II} \right\rfloor$$

It is easily possible that an operation is scheduled in a stage with a negative number. This is no problem. To normalize the schedule, i.e. make the earliest stage have number 0, we simply add the absolute value of the earliest negative stage number to all stage numbers. The semantics will remain the same only the stage numbers are different. Often it is easier to work with the schedule if it is guaranteed that the earliest stage has number 0.

The branch has to be treated specially. All resources for the branch operation have to be reserved in advance for the cycle the branch operation is going to be scheduled. If the architecture has a branch delay of n cycles the operation has to by scheduled in cycle $II - 1 - n$. Of course the II has to be greater than the branch delay slot. When all other operations are scheduled the branch operation is finally inserted in the desired cycle. The stage for the branch must be the very last stage.

We know the cycle for the branch operation in advance but not the stage. If the branch operation depends on an operation that is scheduled in the last stage and a cycle greater or equal the one intended for the branch, a new stage has to be created only for the branch operation. This may not be optimal for the stage count but does not affect the steady state. It increases the code size of prolog and epilog, however.

4.7 Creation of prolog, kernel and epilog

The schedule found in the previous step corresponds to the kernel, the steady state. This schedule can also be used to create the prolog and the epilog by copying operations adequately. As described earlier, the prolog has to fill the pipeline,

and the epilog finishes iterations that have not been finished by the kernel. Both prolog and epilog consist of $SC-1$ partial copies of the kernel that are scheduled consecutively. The construction is like unrolling the kernel schedule $SC-1$ times and selecting only the appropriate operations.

For the prolog the first partial copy contains only operation from kernel stage 0. The next partial copy contains operations from the kernel stages 0 and 1, and so on. The last partial copy of the kernel in the prolog contains operations from the kernel stages $0 \ldots SC-2$.

The epilog is formed in a similar way. The first partial copy of the kernel in the epilog contains operations from the kernel stages $1 \ldots SC-1$. The next partial copy contains operations from the kernel stages $2 \ldots SC-1$, and so on. The last partial copy of the kernel in the epilog contains only operations from kernel stage $SC-1$.

Only the kernel consists of all the operations of the original loop. The prolog consists of possibly multiple copies of all but the last stage, the epilog contains operations of all stages but stage 0. The branch operation is not included in the epilog.

4.7.1 Insertion of buffer registers

As depicted in figure 6 it may be necessary to insert copy operations into the already finished schedule to prevent that values are prematurely overwritten. This can be done in the kernel before the construction of prolog and epilog. For the construction of the prolog and epilog copy operations are not regarded different to normal operations. Normally copy operations are simple operations that only take one clock cycle to finish, hence it should be possible to find a place where the scheduler can put them, except for very dense kernel schedules. Also thanks to the reduction of live ranges in SMS it should not be necessary to insert a large number of copy operations. Since the copy operations are inserted after the actual scheduling the quality of the schedule is not degraded, however, if the scheduler does not find a place to put the copy operation the schedule has to be discarded and the scheduler has to retry with an incremented II. Nevertheless the register pressure is increased.

The algorithm is straight forward. All live ranges inside the kernel are analyzed. Especially taking into account live ranges that reach across stage boundaries. Then copy operations are inserted such that no live range exceeds the II and the operands are updated appropriately. That is the reading operand of the reading operation is set to the register defined by the copy operation.

When a register is found that is live from operation d to operation r and the live range exceeds the II, then the scheduler scans for a free time slot for the copy

operation c from $t_d + II$ backwards to $t_d + latency(d)$. The reading operand of r has to be set the register defined by c.

For very long live ranges possibly more that one copy operation has to be inserted, but this only rarely occurs.

4.7.2 Updating the branch condition

Since the number of kernel iterations is smaller than the number of iterations in the original loop the branch condition in the kernel has to be adapted such that it takes the branch to the epilog correctly. The last $SC - 1$ iterations are finished in the epilog. LLVM runs a pass that converts counted loops such that the induction variable is always incremented by one. If the iteration count is a compile time constant the corresponding immediate operand in the branch operation has to be reduced by $SC-1$. If a register contains the iteration count its value also has to be decremented by $SC - 1$ before entering the modulo scheduled loop, by inserting a subtraction operation. In this case modulo scheduling is only possible if the register holding the iteration count is not updated inside the loop, thus it is loop invariant. Finally the jump target is set to the top of the kernel.

4.8 Regaining SSA form

For the modulo scheduling pass that runs before register allocation it is necessary to reconstruct SSA form, that was destroyed by scheduling. Many passes that run after the modulo scheduling pass depend on the program to be in SSA form, especially the register allocator. Modulo scheduling destroys SSA form in several ways. First of all the phi nodes are substituted by copy operations such that the same algorithm can be used for basic blocks that are in SSA form and basic blocks that are not. In LLVM phi nodes have to be at the beginning of a basic block. Secondly in the prolog and the epilog one register is defined multiple times because multiple copies of the same operation are inserted. The difficulty is to place the phi nodes right and to update the operands in the operations.

Every virtual register has to be defined exactly once before its first use. This is also true for the prolog and the epilog where multiple copies of the operations and therefore also multiple definitions of the same register exist. To solve this we define a new virtual register every time we find a definition except for registers defined in the very last iteration such that the final results are written to the original registers. This is why multiple *virtual representatives* of the same register exist.

To establish SSA form in the kernel additional phi nodes have to be inserted since the kernel has two predecessors, the kernel itself and the prolog. Thus every new phi node has two operands that are read, one for the back edge and one for the edge coming from the prolog.

There are SC new names and the original name for every register defined inside the loop, i.e. $SC+1$ representatives of one original virtual register. We simply scan prolog, kernel, and epilog top down and replace every definition of a virtual register by a new virtual register and save the new name in the map of representatives. The key is the original register, the value is the representative. If the register definition occurs in the last iteration we simply insert the original name as the representative.

If a register is defined by a phi node of the original loop the map of representatives is initialized with the phi node's register operand corresponding to the back edge. I.e. the first representative of the back edge register is the register itself. This needs to be done even though the working copy of the loop does not incorporate phi nodes since they have been replaced by copy operations. The reason for this initialization is the way we determine the kernel's phi nodes.

4.8.1 Renaming and how to find the right operands

We scan the instructions of prolog, kernel, and epilog top-down. First all uses in the instruction are handled then the definitions. When we encounter a use of register r we replace it by r's current representative. That is always the representative of r's last definition. This works because no live range exceeds the II. After all uses in the instruction are handled all register definitions are replaced by new virtual registers. If register r is replaced by the new virtual register v we make v the current representative of r.

4.8.2 Introducing new phi nodes

After all renamings in the prolog have been made the phi nodes for the kernel are inserted. We put a phi node to the top of the kernel for every register that has a representative at this point, i.e. after the renamings of the prolog but before the renamings of the kernel have been made. This is the reason why we need to initialize the map of representatives according to the original loop's phi nodes, to not forget registers that are defined in the last stage. The reading phi operands corresponding to the prolog are the current representatives, the ones defined in the prolog. Every phi node defines a new virtual register, which then becomes the new representative. While the renamings in the kernel take place the operands corresponding to the kernel's back edge are left blank.

After all renamings in the kernel have been made the remaining phi node operands corresponding to the kernel's back edge are filled in. If the phi node defines register r's representative the phi node's back edge operand is r's current representative after the renamings of the kernel, but before the renaming of the epilog.

4.8.3 Final steps

The last step in making the program conform to LLVM again we have to remove the representation of parallelism. This is done by putting the individual operations an instruction contains in a sequential order leaving out the Nops.

Then we put the branch operation to the end of the kernel's basic block.

To update the control flow graph we make all predecessors of the original loop predecessors of the prolog and all successors of the original loop successors of the epilog. We put the original loop's phi nodes to the top of the prolog. The phi nodes are adapted such that they define a new virtual register and the register operands for the loop's back edge are deleted. Then all uses of these back edge operands in the prolog and the kernel's phi nodes are replaced by the newly defined registers.

Then it is save to delete the original loop.

```
renamings ← empty map;
forall phi nodes h of the original loop do
    u ← register corresponding to the back edge in h;
    renamings[u] ← u;
end
for block ∈ {prolog, kernel, epilog} do
    if block = kernel then
        // In the kernel we have to insert new phi nodes.
        // So far they have only the
        // operands that are defined in the prolog.
        forall registers r ∈ renamings do
            n ← new virtual register ;
            insert new phi node defining n and reading renamings[r] at the
            beginning of the kernel;
            renamings[r] ← n;
        end
    end
    foreach instruction i ∈ block do
        Replace old register names with new ones: Algorithm 5;
    end
    if block = kernel then
        // Now we fill in the back edge operands of the phi nodes.
        forall phi nodes h in the kernel do
            r ← register defined by h;
            o ← register that r represents;
            back edge operand of h ← renamings[o];
        end
    end
end
```

Algorithm 4: Rename registers to regain SSA form.

```
forall operations o ∈ i do
    forall register uses u in o do
        replace u renamings[u];
    end
end
forall operations o ∈ i do
    forall definitions d in o do
        if o is in the last iteration then
            // We use the original name.
            newname ← d;
        else
            newname ← new virtual register;
        end
        replace u with newname;
        renamings[d] ← newname;
    end
end
```

Algorithm 5: Replace the register definitions and uses in instruction i.

4.9 Example

The exact way swing modulo scheduling works is best illustrated by using a simple program containing a loop that is feasible for modulo scheduling as an example. The C program of listing 5 in fact contains two loops that can be modulo scheduled, however in the example we concentrate only on the second one. The first loop is only to initialize the arrays with some values.

The VLIW target architecture to which the program is compiled has the following characteristics [3]:

- 4 identical pipelines in parallel, every resource is available four times.
- Regular operations take 1 clock cycle and use one ALU.
- Multiplications take 3 clock cycles. They use one ALU for one cycle and one Multiplier for 3 cycles.
- Loads take at least 4 cycles, if more, the pipeline stalls.
- Branches take 5 clock cycles, i.e. they have a branch delay slot.
- The conditional branch needs two ALUs for one clock cycle.

[3] These are characteristics of the CHILI processor.

```
 s1  #define N 1000
 s2  int main() {
 s3    int a[N];
 s4    int b[N];
 s5    int i = 0;
 s6    int di = 0;
 s7    int *d = &di;
 s8    int result = 0;
 s9    /* Fill the arrays with
s10     * arbitrary values. */
s11    for (i = 0; i < N; ++i) {
s12      a[i] = i;
s13      b[i] = i + 20;
s14    }
s15    i = 0;
s16    for (i = 0; i < N; ++i) {
s17      *d += a[i] * b[i];
s18      result += *d;
s19    }
s20    return result;
s21  }
```

Listing 5: C program used in the example.

LLVM's front end translates the C program into its intermediate representation and does some transformations. Then instruction selection maps the machine independent operations from the intermediate representation to machine dependent operations that are basically the operations found in the target machine's instruction set. The program is still in SSA form and there is still no notion of parallelism, i.e. it is still a sequential program. This is the point where modulo scheduling takes place. Since register allocation has not been done yet the program contains virtual registers. The loop we are going to modulo schedule this in LLVM's machine dependent representation is depicted in listing 6:

```
 s1  %reg1026<def> = PHI %reg1037, mbb<bb.bb15_crit_edge,0x1586080>, %reg1031,
         mbb<bb15,0x159c590>
 s2  %reg1027<def> = PHI %reg1037, mbb<bb.bb15_crit_edge,0x1586080>, %reg1030,
         mbb<bb15,0x159c590>
 s3  %reg1028<def> = PHI %reg1037, mbb<bb.bb15_crit_edge,0x1586080>, %reg1029,
         mbb<bb15,0x159c590>
 s4  %reg1038<def> = SHL_REG_IMM %reg1026, 2
 s5  %reg1039<def> = ADD_REG_IMM %r61, <fi#0>
 s6  %reg1040<def> = ADD_REG_IMM %r61, <fi#1>
 s7  %reg1041<def> = LOAD_REG_REG %reg1040, %reg1038
 s8  %reg1042<def> = LOAD_REG_REG %reg1039, %reg1038
 s9  %reg1031<def> = ADD_REG_IMM %reg1026, 1
s10  %reg1043<def> = MUL_REG_REG %reg1042, %reg1041
s11  %reg1029<def> = ADD_REG_REG %reg1043, %reg1028
s12  %reg1030<def> = ADD_REG_REG %reg1029, %reg1027
s13  JUMP_LAB_CRI %reg1031, 1, 1000, mbb<bb15,0x159c590>
```

Listing 6: The original loop

The loop is in SSA form and contains phi nodes $s1$, $s2$, and $s3$. The labels for the

basic blocks signify which register is read according to the control flow. Basic block `mbb<bb.bb15_crit_edge,0x1586080>` denotes the loop's predecessor and basic block `mbb<bb15,0x159c590>` is the loop itself. $s4$, $s5$, and $s6$ are address computations, $s9$ updates the induction variable. The jump operation $s13$ takes the jump as long as register `%reg1031`—the induction variable—is not equal to 1000. The parameter 1 signifies $\neq$.

Since we do not know whether modulo scheduling will be possible we make a working copy of the basic block and replace the phi nodes with copy operations. The registers read by the copy operations are the ones corresponding to the back edge. After that, the program is not in SSA form anymore, however every register is still defined only once inside the basic block.

```
 s1  %reg1026<def> = MOV_REG %reg1031
 s2  %reg1027<def> = MOV_REG %reg1030
 s3  %reg1028<def> = MOV_REG %reg1029
 s4  %reg1038<def> = SHL_REG_IMM %reg1026, 2
 s5  %reg1039<def> = ADD_REG_IMM %r61, <fi#0>
 s6  %reg1040<def> = ADD_REG_IMM %r61, <fi#1>
 s7  %reg1041<def> = LOAD_REG_REG %reg1040, %reg1038
 s8  %reg1042<def> = LOAD_REG_REG %reg1039, %reg1038
 s9  %reg1031<def> = ADD_REG_IMM %reg1026, 1
s10  %reg1043<def> = MUL_REG_REG %reg1042, %reg1041
s11  %reg1029<def> = ADD_REG_REG %reg1043, %reg1028
s12  %reg1030<def> = ADD_REG_REG %reg1029, %reg1027
s13  JUMP_LAB_CRI %reg1031, 1, 1000, mbb<bb15,0x159c590>
```

Listing 7: The working copy that will be scheduled

Construction of the DDG

The next step is to construct the data dependence graph with algorithm 1. The resulting DDG is shown in figure 10. The edge labels denote the iteration distance and the type of the dependence. On edges with a delay is greater than 1, the delay is also displayed. Since there are only reading memory accesses there are only true dependences.

RecMII

There are 3 elementary circuits in the DDG:

- $s1 \xrightarrow{0} s9 \xrightarrow{1} (s1)$
- $s3 \xrightarrow{0} s11 \xrightarrow{1} (s3)$
- $s2 \xrightarrow{0} s12 \xrightarrow{1} (s2)$

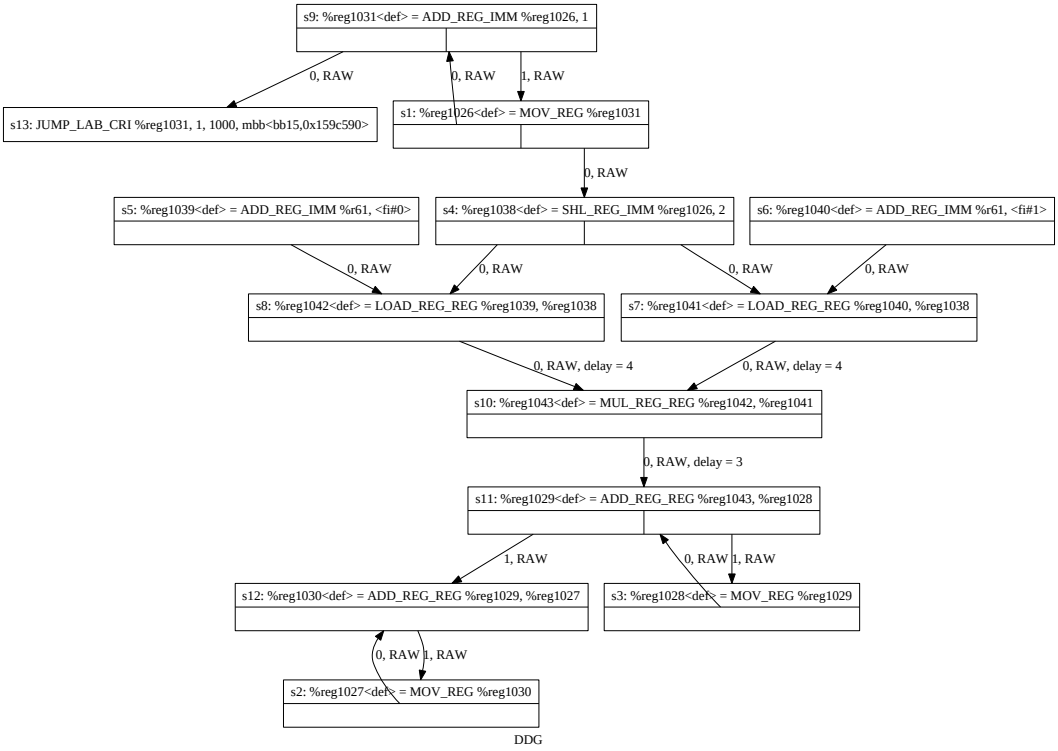

Figure 10: DDG

The recurrences in this example are rather simple. They all contain only 2 operations and every recurrence c has a $delay(c)$ of 2 and a $distance(c)$ of 1. To make the inequation

$$delay(c) - II \cdot distance(c) \leq 0$$

true the II must be at least 2, hence the RecMII is 2.

ResMII

Our resource model has 8 different resources: four ALUs and four multipliers. All regular operations need one ALU for 1 cycle. The multiply operations need additionally one multiplier for 3 clock cycles, which makes a total of 4 resource claims. The jump operation needs 2 ALUs. Over all the resource needs of the whole loop used for the estimation of the ResMII is $11 \cdot 1 + 1 \cdot 4 + 1 \cdot 2 = 17$.

$$ResMII = \left\lceil \frac{17}{8} \right\rceil = 3$$

Table 1: Node properties form the example

	$ASAP$	$ALAP$	MOV	D	H
$s1$	0	0	0	0	16
$s2$	0	9	9	0	1
$s3$	0	8	8	0	2
$s4$	1	1	0	1	9
$s5$	0	1	1	0	9
$s6$	0	1	1	0	9
$s7$	2	2	0	2	8
$s8$	2	2	0	2	8
$s9$	1	9	8	1	1
$s10$	6	6	0	6	4
$s11$	9	9	0	9	1
$s12$	10	10	0	10	0
$s13$	2	10	8	2	0

However since the branch takes 5 clock cycles the MII must be set to 5 since an iteration of the loop cannot be shorter than the time the branch needs.

$$MII = 5$$

Loop unrolling would be a method to increase the MII while simultaneously increasing the available parallelism. This way a MII that is smaller than the jump latency can be avoided without losing efficiency. The kernel's code size is increased, however.

Node properties

To compute the node properties we ignore one back edge in every recurrence to gain an acyclic dependence graph. In our case we ignore every back edge.

The properties are computed as described in section 4.4. Table 1 shows the values for the example loop.

Partial ordering of the nodes

The partial order starts with all nodes in the most critical recurrence. Since all recurrences in the example have the the same latency we can start with any of them. We start with recurrence $s1 \xrightarrow{0} s9 \xrightarrow{1} (s1)$. Hence the first set in the partial order is $\{s1, s9\}$. The next set in the partial order contains the next

critical recurrence and all nodes that lie between any node of this recurrence and any node already in the partial order according to the DDG. We chose recurrence $s3 \xrightarrow{0} s11 \xrightarrow{1} (s3)$. The additional nodes are $s7$, $s8$, $s10$, and $s4$. Thus the second set is $\{s3, s4, s7, s8, s10, s11\}$. The next set consists only of the third recurrence as there are no connecting nodes that lie between the new recurrence and the nodes already in the partial order. The third set is $\{s2, s12\}$. As there are no recurrences left all other sets are the remaining connected components. The order in which they are inserted does not matter. All other connected components only consist of one operation.

We gained the partial order:

$$po = \langle \{s1, s9\}, \{s3, s4, s7, s8, s10, s11\}, \{s2, s12\}, \{s13\}, \{s6\}, \{s5\} \rangle$$

Ordering of the nodes

Once we have the partial order, one set after the other will be examined to deduce the final order in which the operations will be put into the schedule. Again we ignore one back edge in every recurrence to gain an acyclic graph.

We start with the node with the highest $ASAP$ value in the first set: $s9$. The next one is the second node in the first set since there are only two nodes. The order so far is $o = \langle s9, s1 \rangle$.

Now we examine the second set $\{s3, s4, s7, s8, s10, s11\}$. There is one node in the set adjacent to the nodes already in the order: $s4$. It is a successor of $s1$ thus the direction is *top-down*. The next operations in top-down direction are $s7$ and $s8$. Because they have the same attributes it does not matter which one we take first. We take $s8$. The order so far is $o = \langle s9, s1, s4, s8 \rangle$. The next node in top-down direction is $s10$ which has a lower H value than $s7$. Thus we take $s7$ first then $s10$. We continue in top-down direction and append $s11$. The order so far is $o = \langle s9, s1, s4, s8, s10, s7, s11 \rangle$. Since we ignore the back edge there are no successors in the set left thus we have to switch direction to *bottom-up*. In bottom-up direction we find only one more node namely $s3$ and the second set is complete. The order so far is $o = \langle s9, s1, s4, s8, s10, s7, s11, s3 \rangle$

The remaining sets are handled in the same way and we get the final order in which the operations are put into the schedule

$$o = \langle s9, s1, s4, s8, s10, s7, s11, s3, s12, s2, s13, s6, s5 \rangle$$

Scheduling

Scheduling the operations is carried out in the order determined by the previous step. Since the MII is 5 the first II we try for the kernel is also 5. First we reserve all necessary resources for the branch operations, because the cycle where it will be

scheduled is known in advance. The stage however is not know until the schedule is complete. The cycle for the branch b is $II - delay(b) = 0$.

We use the same notation for the program as in the loop unrolling example of section 2.2. For the scheduling phase we use use an unfolded notation, that is we write the individual stages separately. In the final program they will be combined to a dense schedule for the kernel.

The first operation to be scheduled is $s9$. Since the partial schedule is empty there are neither predecessors nor successors in the partial schedule. In pre pass version where we only use the modulo scheduler as a preconditioning step for the list scheduler we chose the *backwards* direction in this case. We look for a suitable time slot from $Early\_start+II-1$ to $Early\_start$ where $Early\_start = ASAP = 1$. All resources except the ones reserved for the branch are free so we can put $s9$ into time slot 5 which corresponds to cycle 0 of stage 1. Note that we put the operation not to the beginning of the instruction since these resources are already occupied by the branch instruction which will also be scheduled to cycle 0.

```
// *** stage 0 ***
{ ; ; ; ;} // 0
{ ; ; ; ;} // 1
{ ; ; ; ;} // 2
{ ; ; ; ;} // 3
{ ; ; ; ;} // 4

// *** stage 1 ***
{ ; ; %reg1031<def> = ADD_REG_IMM %reg1026, 1 ; ;} // 5
{ ; ; ; ;} // 6
{ ; ; ; ;} // 7
{ ; ; ; ;} // 8
{ ; ; ; ;} // 9
```

The next node is $s1$. $s9$ is both predecessor and successor of $s1$ in the partial schedule. The last direction was backwards thus we look for a free time slot between $min(Late\_start, Early\_start+II-1) = min(4, 5) = 4$ and $Early\_start = 1$ and put the operation in time slot 4 which is cycle 4 of stage 0.

```
// *** stage 0 ***
{ ; ; ; ;} // 0
{ ; ; ; ;} // 1
{ ; ; ; ;} // 2
{ ; ; ; ;} // 3
{ %reg1026<def> = MOV_REG %reg1031 ; ; ; ;} // 4

// stage 1
{ ; ; %reg1031<def> = ADD_REG_IMM %reg1026, 1 ; ;} // 5
{ ; ; ; ;} // 6
{ ; ; ; ;} // 7
{ ; ; ; ;} // 8
{ ; ; ; ;} // 9
```

We continue to schedule the remaining nodes in the list based on the predecessors and successors in the partial schedule as described in section 4.6 without inserting the branch operation.

After all operations are placed we have a schedule with a SC of 4. Finally we put the branch operation to the cycle we determined in advance, in our case 0, of the last stage, stage 3. So far no unresolvable resource conflict has occurred. The next step is to analyze the live ranges. We find that the branch operation $s13$ scheduled in cycle 0 of stage 3 reads register `%reg1031` which is defined by operation $s9$ scheduled in cycle 0 of stage 1. The distance between the two operations is $2 * II$ hence we need at least one copy operation to split this live range into two intervals of length II. This copy operation would be scheduled into cycle 0 of stage 2. Unfortunately all ALUs of cycle 0 are already occupied by operations of the various stages thus we have to use two copy operations to split the live range. This also means that we need two more virtual registers which increases the register pressure. We insert the first copy operation `%reg1065<def> = MOV_REG %reg1031` into cycle 3 of stage 1 and the second `%reg1066<def> = MOV_REG %reg1065` into cycle 3 of stage 2. Finally we update the branch operation to read `%reg1066` instead of `%reg1031`.

Modulo scheduling was successful with $II = 5$, no unresolvable resource conflicts occurred. The unfolded schedule of the individual stages is depicted in listing 4.9:

```
// *** stage 0 ***
{ ; ; ; ;} // 0
{ ; ; ; ;} // 1
{ ; ; ; ;} // 2
{ ; ; ; ;} // 3
{ %reg1026<def> = MOV_REG %reg1031 ;
  ;
  %reg1040<def> = ADD_REG_IMM %r61, <fi#1> ;
  %reg1039<def> = ADD_REG_IMM %r61, <fi#0> ;} // 4

// *** stage 1 ***
{ ;
  ;
  %reg1031<def> = ADD_REG_IMM %reg1026, 1 ;
  %reg1038<def> = SHL_REG_IMM %reg1026, 2 ;} // 5
{ %reg1042<def> = LOAD_REG_REG %reg1039,%reg1038 ;
  %reg1041<def> = LOAD_REG_REG %reg1040, %reg1038 ;
  ;
  ;} // 6
{ ; ; ; ;} // 7
{ ;
  %reg1065<def> = MOV_REG %reg1031 ; // Inserted copy
  ;
  ;} // 8
{ ; ; ; ;} // 9

// *** stage 2 ***
{ ; ; ; ;} // 10
{ ;
  ;
```

```
  %reg1043<def> = MUL_REG_REG %reg1042, %reg1041 ;
  %reg1028<def> = MOV_REG %reg1029 ;} // 11
{ ; ; ; ;} // 12
{ %reg1027<def> = MOV_REG %reg1030 ;
  ;
  %reg1066<def> = MOV_REG %reg1065 ; // Inserted copy
  ;} // 13
{ ;
  %reg1029<def> = ADD_REG_REG %reg1043, %reg1028 ;
  ;
  ;} // 14

// *** stage 3 ***
{ JUMP_LAB_CRI %reg1066, 1, 1000, mbb<bb15,0x159d050> ;
  ;
  ;} // 15
{ ; ; ; ;} // 16
{ %reg1030<def> = ADD_REG_REG %reg1029, %reg1027 ;
  ;
  ;
  ;} // 17
{ ; ; ; ;} // 18
{ ; ; ; ;} // 19
```

The kernel's final schedule is dense and contains very few Nops:

```
{ JUMP_LAB_CRI %reg1066, 1, 1000, mbb<bb15,0x159d050> ;
  %reg1031<def> = ADD_REG_IMM %reg1026, 1 ;
  %reg1038<def> = SHL_REG_IMM %reg1026, 2 ;}
{ %reg1042<def> = LOAD_REG_REG %reg1039, %reg1038 ;
  %reg1041<def> = LOAD_REG_REG %reg1040, %reg1038 ;
  %reg1043<def> = MUL_REG_REG %reg1042, %reg1041 ;
  %reg1028<def> = MOV_REG %reg1029 ;}
{ %reg1030<def> = ADD_REG_REG %reg1029, %reg1027;
  ;
  ;
  ;}
{ %reg1027<def> = MOV_REG %reg1030 ;
  %reg1065<def> = MOV_REG %reg1031 ;
  %reg1066<def> = MOV_REG %reg1065 ;
  ;}
{ %reg1026<def> = MOV_REG %reg1031 ;
  %reg1029<def> = ADD_REG_REG %reg1043, %reg1028 ;
  %reg1040<def> = ADD_REG_IMM %mreg127, <fi#1> ;
  %reg1039<def> = ADD_REG_IMM %mreg127, <fi#0> ;}
```

Creating prolog and epilog

As described in section 4.7 the prolog and epilog consist of consecutive partial copies of the kernel. In the example there will be 3 partial copies of the kernel in the prolog and the epilog since the $SC = 4$.

The prolog is constructed as follows: The first copy of the kernel contains only operations of stage 0, the next of stage 0 and stage 1 the last copy of the kernel in the prolog contains operations from the stages 0, 1, and 2.

The epilog is constructed analogously: The first copy of the kernel contains operations of the kernel stages 1, 2, and 3, the next copy contains operations

of the stages 2 and 3. The last copy contains only operations of stage 3 but without the branch instruction. Then we update the operand of the kernel's branch operation that determines the trip count. We have to decrement the constant by 3 as the last 3 iterations are finished in the epilog. We change operation `JUMP_LAB_CRI %reg1066, 1, 1000, mbb<bb15,0x159d050>` to `JUMP_LAB_CRI %reg1066, 1, 997, mbb<bb15,0x159d050>`

Reconstructing SSA form

To reconstruct SSA form prolog, kernel, epilog are traversed top-down and all register definitions are replaced by new virtual registers such that no register is defined twice. These new names are saved in a lookup table corresponding to the register replaced. The lookup table is initialized by the back edge operands of the original loop's phi nodes.

In the same pass register uses are updated such that they read the right virtual register. The name of the register read is always the last representative in the list. The initialization of the renaming map based on the original loop's back edge operands is:

original	representative
%reg1029	%reg1029
%reg1030	%reg1030
%reg1031	%reg1031

Starting with the top of the prolog we encounter the first operation `%reg1026<def> = MOV_REG %reg1031`. We find the reading operand's representative `%reg1031` and replace it. In fact this is not a replacement but it does not make a difference in the algorithm. Uses for which we do not find a representative in the map are left as they are. This happens for the next two operations we encounter `%reg1040<def> = ADD_REG_IMM %r61, <fi#1>` and `%reg1039<def> = ADD_REG_IMM %r61, <fi#0>`. All uses in the instruction are handled thus we can replace the definitions with new virtual registers.

```
{ ; ; ; ;} // 0
{ ; ; ; ;} // 1
{ ; ; ; ;} // 2
{ ; ; ; ;} // 3
{ %reg1067<def> = MOV_REG %reg1031 ;
  ;
  %reg1066<def> = ADD_REG_IMM %r61, <fi#1> ;
  %reg1065<def> = ADD_REG_IMM %r61, <fi#0> ;} // 4
```

The renaming map is updated:

original	representative
%reg1026	%reg1067
%reg1029	%reg1029
%reg1030	%reg1030
%reg1031	%reg1031
%reg1039	%reg1065
%reg1040	%reg1066

We continue in the prolog: The next two operations we encounter are `%reg1031<def> = ADD_REG_IMM %reg1026, 1` and `%reg1038<def> = SHL_REG_IMM %reg1026, 2` . Both read `%reg1026` thus we have to replace the operand according to the lookup table by `%reg1067`. After that, we exchange the definitions and enter them in the map.

```
{ ; ; ; ;} // 0
{ ; ; ; ;} // 1
{ ; ; ; ;} // 2
{ ; ; ; ;} // 3
{ %reg1067<def> = MOV_REG %reg1031 ;
  ;
  %reg1066<def> = ADD_REG_IMM %r61, <fi#1> ;
  %reg1065<def> = ADD_REG_IMM %r61, <fi#0> ;} // 4
{ ;
  ;
  %reg1069<def> = ADD_REG_IMM %reg1067, 1 ;
  %reg1068<def> = SHL_REG_IMM %reg1067, 2 ;} // 5
```

The updated map of representatives:

original	representative
%reg1026	%reg1067
%reg1029	%reg1029
%reg1030	%reg1030
%reg1031	%reg1069
%reg1038	%reg1068
%reg1039	%reg1065
%reg1040	%reg1066

We continue this procedure until we reach the kernel. The insertion of the phi node is described using as an example the representative of register `%reg1040`. By the time we reach the kernel the representative of this register is `%reg1086`. Thus we create a new virtual register `%reg1097` and the new phi node without a back edge operand `%reg1097<def> = PHI %reg1086, mbb<bb15,0x15aa400>` and insert it to the top of the kernel. Then we make `%reg1097` the new representative of `%reg1040`.

After the renamings of the kernel the current representative of `%reg1040` is `%reg1040`. This register is defined in stage 0. Stage 0 of the kernel is part of the last iteration thus the representative and the register are the same. Finally we can fill in the

phi node's back edge operand and get the complete phi node: `%reg1097<def> = PHI %reg1086, mbb<bb15,0x15aa400>, %reg1040, mbb<bb15,0x15a8c40>`. This procedure is done for all registers that have a representative after all renamings in the prolog are made and before the renamings in the kernel.

Then we continue the renaming procedure in the epilog analogous to the prolog and the kernel.

5 Evaluation

5.1 The CHILI architecture

The CHILI is developed by On Demand Micro Electronics. Its main application area is mobile multimedia processing with a focus on video decoding such as H264 and MPEG. Two CHILI cores are incorporated in On Demand's multimedia chip SVENm which is a system on chip designed for multi-format video decoding and image processing.

The CHILI is a four way VLIW architecture featuring four identical pipelines in parallel. I.e. up to four operations can be issued per clock cycle. It has 64 general purpose registers each with a length of 32 bits. The CHILI offers no floating point capabilities since they are not needed in the area of application. However it features SIMD extensions and special operations for video processing such as multiply accumulate, sum of absolute differences, population count, leading 0s, leading 1s, clip, align etc.

Predicated execution is offered for many operations, unfortunately not for memory load and store. The test of the predicate and the operation are executed in parallel thus conditional operations use two VLIW slots.

Each pipeline has 11 stages. Regular instructions take one clock cycle, multiplication takes 3 cycles. Loads take at least four cycles but can take up to 40.

A distinguishing characteristic of the CHILI are the large branch delays. A branch takes 5 cycles, i.e. it has a delay slot of 4 cycles. This means that up to 16 operations can be issued between the branch operation and the point where the branch is taken. This leads to the phenomenon that for short loops the majority of the operations are actually scheduled after the branch operation. The branch operation must reside in the first VLIW slot. Like other conditional operations the conditional branch uses two VLIW slots.

5.2 Results

All results are gathered using On Demand's cycle accurate CHILI simulator. As a micro benchmark suite we used the sample loops found in [Ogr04]. The loops are short and suited for software pipelining as they are single basic block loops. They contain memory accesses and integer arithmetics but no floating point arithmetics. The loops of the example are the only ones that are software pipelined. The numbers also include initialization code for the arrays that is not software pipelined even though it could be. The differences in code size and execution time are only caused by the software pipelined loop.

We evaluated both versions of the algorithm. The pre pass runs before register allocation and post pass after register allocation. Even though the pre pass mod-

ulo scheduler is in fact only a preconditioning run for the list scheduler it leads to improvements in execution time.

Table 2 shows the code size of the two software pipelined versions compared to each other and compared to the version without software pipelining. As expected the code size is increased by software pipelining. The number of instructions is increased significantly by software pipelining also the number of nops these instructions contain. The reason for this is the additional code needed for the prolog and epilog.

We also observe that the pre pass software pipeliner generates shorter programs than the post pass software pipeliner. Since the pre pass scheduler in our implementation is only a preconditioning step for the list scheduler, it is possible for the list scheduler to optimize the straight line code of prolog and epilog and achieve denser schedules.

Even though the absolute number of nops is increased by software pipelining the average number of nops per instruction is decreased in general. This value can be used as an estimation for the utilization of parallelism. Only test10 shows a higher number of nops per instruction for both software pipelined versions compared to the schedule achieved by the list scheduler. For this test case the code size is increased the most by the post pass modulo scheduler.

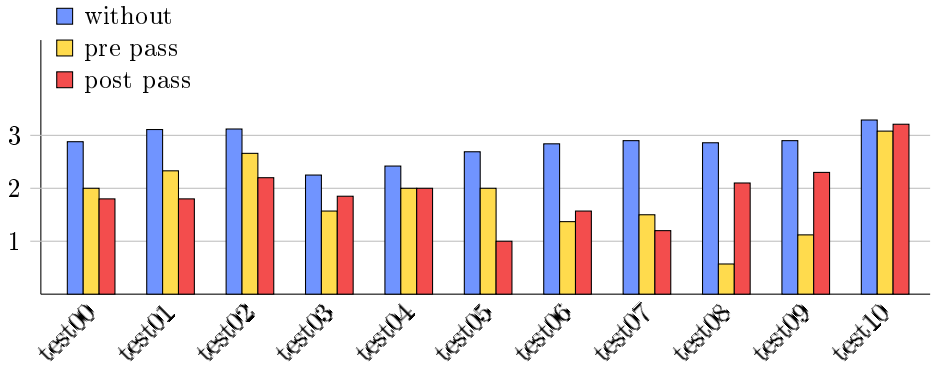

Figure 11: Nops/instruction

Table 3 depicts information about the code size regarding only the block that is executed iteratively. I.e. the loop body of the list scheduled loop respectively the kernel of the software pipelined loop. The number of instructions of the loop body is equal to the number of clock cycles a loop iteration takes. Since modulo scheduling mainly focuses on generating dense kernel schedules the number of nops inside the kernel is of interest. Figure 11 shows a chart representing the average

Table 2: Code size

		instructions	nops	nops/instruction
test00	without	50	157	3.14
	pre pass	55	160	2.90
	post pass	66	202	3.06
test01	without	45	141	3.13
	pre pass	50	148	2.96
	post pass	71	215	3.02
test02	without	47	149	3.17
	pre pass	50	157	3.14
	post pass	64	203	3.17
test03	without	47	135	2.87
	pre pass	56	144	2.57
	post pass	74	216	2.91
test04	without	74	231	3.12
	pre pass	82	242	2.95
	post pass	97	302	3.11
test05	without	63	189	3.00
	pre pass	76	210	2.76
	post pass	85	235	2.76
test06	without	63	193	3.06
	pre pass	76	202	2.65
	post pass	85	249	2.92
test07	without	52	160	3.07
	pre pass	61	167	2.73
	post pass	77	221	2.87
test08	without	67	205	3.05
	pre pass	81	210	2.59
	post pass	102	309	3.02
test09	without	68	214	3.14
	pre pass	80	217	2.71
	post pass	104	326	3.13
test10	without	75	238	3.17
	pre pass	120	384	3.20
	post pass	166	569	3.42

Table 3: Structure of the loop

		instructions	nops	nops/instruction
test00	without	9	26	2.88
	pre pass	6	12	2.00
	post pass	5	9	1.80
test01	without	9	28	3.11
	pre pass	6	14	2.33
	post pass	5	9	1.80
test02	without	8	25	3.12
	pre pass	6	16	2.66
	post pass	5	11	2.20
test03	without	8	18	2.25
	pre pass	7	11	1.57
	post pass	7	13	1.85
test04	without	7	17	2.42
	pre pass	7	14	2.00
	post pass	6	12	2.00
test05	without	13	35	2.69
	pre pass	10	20	2.00
	post pass	7	7	1.00
test06	without	13	37	2.84
	pre pass	8	11	1.37
	post pass	7	11	1.57
test07	without	10	29	2.90
	pre pass	6	9	1.50
	post pass	5	6	1.20
test08	without	15	43	2.86
	pre pass	7	4	0.57
	post pass	10	21	2.10
test09	without	14	41	2.92
	pre pass	8	9	1.12
	post pass	10	23	2.30
test10	without	24	79	3.29
	pre pass	23	71	3.08
	post pass	23	74	3.21

number of nops per instruction. For all test cases both versions of the software pipeliner achieve shorter loop bodies than the list scheduler. In many cases the post pass version creates denser schedules than the pre pass version, however the lowest number of nops per instruction is achieved by the pre pass version. In test08 7 instruction contain only 4 nops. Since the pre pass modulo scheduler does not create the final schedule the length of the kernel is not equal to the II. In general it is greater as the list scheduler does not account for loop carried dependences.

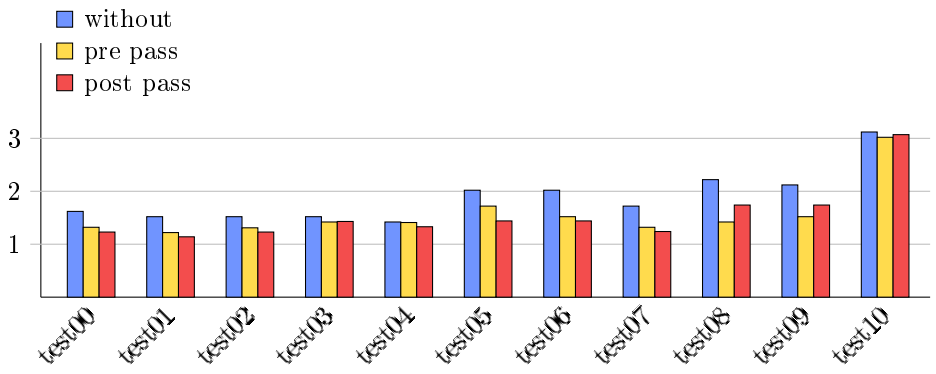

Figure 12: Execution time in 1000 cycles with iteration count 100.

To observe the execution time we ran the same program with 3 different iteration counts: 1000, 100, 30. The execution time in clock cycles is depicted in table 4. In all cases but one the software pipelined programs need fewer clock cycles than the programs scheduled by the list scheduler. Figure 12 shows the execution times with iteration count 100. When running the post pass version of test10 with an iteration count 30 the overhead caused by prolog and epilog outweigh the gain made in the kernel. With increased iteration count the overhead is compensated. As expected performance gains increase with higher iteration counts.

Values that are of importance for the modulo scheduling algorithm are depicted in table 5. In 14 of 22 cases it was was possible to construct a kernel with the minimum initiation interval. All kernels have a stage count of 3 or 4. We observe that the recMII and the resMII are often smaller than 5, the latency of a branch operation. The kernel can not be shorter than this latency. Loop unrolling would increase the recMII and the resMII while offering more ILP hence it can be used to avoid a MII that is smaller than the latency of the branch operation. Nevertheless a dense kernel schedule is likely to be found at the cost of an increased code size.

The column “copies” shows the number of additional copy operations that have to be inserted, i.e. additional registers needed compared to the conventional schedule.

Table 4: Execution time

		1000		100		30	
		cycles	%	cycles	%	cycles	%
test00	without	16026		1626		506	
	pre pass	13022	81.25	1322	81.30	412	81.42
	post pass	12036	75.10	1236	76.01	396	78.26
test01	without	15026		1526		476	
	pre pass	12022	80.00	1222	80.07	382	80.25
	post pass	11041	73.47	1141	74.77	371	77.94
test02	without	15025		1525		475	
	pre pass	13018	86.64	1318	86.42	408	85.89
	post pass	12035	80.09	1235	80.98	395	83.15
test03	without	15025		1525		475	
	pre pass	14021	93.31	1421	93.18	441	92.84
	post pass	14039	93.43	1439	94.36	459	96.63
test04	without	14025		1425		445	
	pre pass	14019	99.95	1419	99.57	439	98.65
	post pass	13037	92.95	1337	93.82	427	95.95
test05	without	20026		2026		626	
	pre pass	17022	84.99	1722	84.99	532	84.98
	post pass	14040	70.10	1440	71.07	460	73.48
test06	without	20026		2026		626	
	pre pass	15020	75.00	1520	75.02	470	75.07
	post pass	14040	70.10	1440	71.07	460	73.48
test07	without	17026		1726		536	
	pre pass	13021	76.47	1321	76.53	411	76.67
	post pass	12041	70.72	1241	71.90	401	74.81
test08	without	22026		2226		686	
	pre pass	14027	63.68	1427	64.10	447	65.16
	post pass	17046	77.39	1746	78.43	556	81.04
test09	without	21026		2126		656	
	pre pass	15020	71.43	1520	71.49	470	71.64
	post pass	17046	81.07	1746	82.12	556	84.75
test10	without	31025		3125		955	
	pre pass	30025	96.77	3025	96.80	925	96.85
	post pass	30071	96.92	3071	98.27	971	101.67

Table 5: Modulo scheduling characteristics

		SC	II	MII	recMII	resMII	copies	circuits
test00	pre pass	3	5	5	2	2	0	2
	post pass	3	5	5	1	2	1	13
test01	pre pass	3	5	5	2	5	0	2
	post pass	4	5	5	1	2	3	8
test02	pre pass	3	5	5	2	2	0	1
	post pass	3	5	5	1	2	0	11
test03	pre pass	3	7	7	7	2	0	7
	post pass	3	7	7	7	2	1	95
test04	pre pass	3	6	6	6	2	0	5
	post pass	3	6	6	6	2	1	40
test05	pre pass	3	14	6	6	4	0	8
	post pass	3	7	6	6	3	4	435
test06	pre pass	4	6	5	3	3	5	3
	post pass	3	7	5	5	3	2	43
test07	pre pass	4	5	5	2	3	2	3
	post pass	4	5	5	1	2	3	15
test08	pre pass	4	7	5	4	4	8	3
	post pass	3	10	9	9	3	2	70
test09	pre pass	4	6	5	4	4	6	3
	post pass	3	10	9	9	3	2	68
test10	pre pass	3	23	23	23	4	0	297
	post pass	3	23	23	23	3	1	23440

The post pass version needs in general more additional registers. This is because after register allocation more dependences exist between registers that have to be resolved by renaming. Thanks to SSA form we can be sure that before register allocation every register is defined exactly once. Thus there cannot be WAW and WAR dependences between registers. After register allocation these dependences have to be considered. This fact is also revealed in column "circuits" that shows the number of elementary circuits in the DDG. The number of circuits is significantly larger after register allocation. Since we need to compute every elementary circuit this can lead to problems during compilation. Even a short loop like test10 can have 23440 circuits.

This results from a DDG containing 16 nodes and 75 edges. To estimate the recMII only the circuit with the longest latency would be necessary however the ordering procedure needs all elementary circuits. In the current version we do not try to simplify the DDG. This may cause that redundant edges exist in the DDG which increases the number of circuits. I.e. there can be more than one edge between two nodes in the DDG. For example a WAW edge and a RAW edge. All dependences between these two nodes are satisfied if the most crucial dependence is satisfied. This is an opportunity to simplify the DDG and reduce the number of circuits.

6 Related Works

Joseph A. Fisher introduced the concept of VLIW machines in [Fis83]. The architecture tries to achieve high computing power by encoding more than one RISC like operations in one instruction word and execute them in parallel. All scheduling has to be done by the compiler without the help of additional hardware, which leads to a lower price for the hardware. To utilize the available parallel hardware resources efficiently and to achieve speedups compared to sequential execution the quality of the scheduler is of great importance. The problem is that the scope of a single basic block is too limited to offer enough parallelism to the scheduler such that the hardware is utilized badly. Fisher had already developed a scheduling technique called trace scheduling for micro-code compaction [Fis81]. This scheduling technique considers a sequence of basic blocks called a trace to enlarge the scope and to extract more parallelism. How the traces are formed depends on profiling information since the algorithm tries to favor the common case. Paths outside the trace on the other hand are degraded since compensation code has to be included to remain the semantics.

Many other scheduling algorithms have been developed to enlarge the region the scheduler operates on. Those are for example linear regions like superblocks [HMC+95], which are similar to traces but with a single entry point or non linear regions like treegions [HBC98].

All of these scheduling techniques work on acyclic structures, but programs spend most of their time in loops, thus it seems promising to optimize their execution. Monica Lam stated in [Lam88] that software pipelining seemed to be an effective method to schedule loops for VLIW architectures.

Rau introduced a heuristic called *iterative modulo scheduling* [Rau94] which was the basis for other modulo scheduling algorithms. It became evident that register pressure is significantly increased by software pipelining, thus algorithms were developed that tried to reduce the register requirements. Examples for those algorithms are *slack modulo scheduling* [Huf93] and *iterative register-sensitive software pipelining* [DJG98] which is based on iterative modulo scheduling.

The algorithm implemented for this work also falls into the category of software pipelining algorithms reducing register pressure. *Swing modulo scheduling* was presented in [Llo96] by Josep Llosa. It achieves good scheduling results while keeping register pressure low by a sophisticated ordering algorithm for the operations. Since it does not incorporate backtracking compilation time is low compared to other software pipelining algorithms.

A comparative study [CLG] showed that swing modulo scheduling performs in many cases better in terms of compilation time and register requirements than other software pipelining techniques while achieving near optimal parallelism.

The compiler platform LLVM we implemented the modulo scheduler for was

presented by Chris Lattner in [Lat02]. LLVM is a state of the art compiler platform used in both industry and research.

Tanya Lattner implemented a swing modulo scheduler for LLVM's SPARC back end [Lat05] that was a good starting point for our implementation. Also Julia Ogris's' implementation of a slack modulo scheduler [Ogr04] provided valuable insight into the development of a modulo scheduler.

7 Future work

We implemented two versions of the modulo scheduler, one that runs before register allocation working on a program in SSA form and a version that runs after register allocation. Both show improvements, however we feel that an integrated solution that combines the advantages of both versions would perform better. The advantage of the pre pass version is that the DDG contains less dependences, thanks to the SSA form, however the disadvantage is that in LLVM there is only limited support for VLIW architectures. For example, the branch operation has to reside at the end of a basic block and it is not possible to fill branch delay slots before register allocation. So we had to make the software pipelined schedule conform to LLVM again and in some cases degrade the quality of the schedule.

7.1 Incorporating VLIW hardware description into tablegen

Tablegen is a utility used in LLVM to generate C++ code from architecture descriptions. Compared to the generated code representing the architecture description, tablegen's input is succinct and tablegen does a lot of work for the programmer. Unfortunately tablegen's architecture description language does not provide constructs to describe VLIW architectures. Important aspects like the number of parallel resources cannot be expressed. In our implementation we wrote the C++ code describing the CHILI architecture by hand.

It is possible to write back ends for tablegen and introduce new concepts like parallel resources. This would make specifying the hardware less error prone and changing architecture details easier.

The target's instruction set is described using tablegen, too. Unfortunately it is not possible to represent instructions that contain more than one operation. I.e. the possibility of parallel execution cannot be expressed.

7.2 Analysis of array accesses

Our implementation does not inspect memory accesses very thoroughly. The dependence analysis does not try to find out if two memory accesses actually write or read the same memory cell. It conservatively assumes that they do and introduces a dependence. This means that memory accesses can be considered dependent from each other even though they do not access the same memory cell thus introducing unnecessary dependences.

For intra iteration dependences alias analysis would provide the information whether it can be guaranteed that two memory accesses are independent from each other. The analysis for inter iteration dependences must also take into account that the address of the accessed memory cell depends on the iteration in which the

access takes place. A typical case is when a loop accesses every entry in an array, the pointer to the array is updated every iteration.

Array accesses are the only dependences that can have an iteration difference greater than 1, since registers are updated every iteration. We conservatively set all iteration differences to 1. Further analysis could find out for some cases greater iteration differences and thus reducing the RecMII.

7.3 Optimize circuit finding algorithm

Especially the post pass version of the modulo scheduler is affected by the possibly large number of elementary circuits in the DDG. The DDG for the pre pass version normally has fewer dependences. Since we need every elementary circuit to calculate the RecMII and for the ordering of the operations, the algorithm that finds the elementary circuits, should be efficient to reduce compilation time.

Additionally the DDG has to be analyzed to eliminate redundant edges.

7.4 Variable trip count

The current version only handles loops with a trip count, that is know at compile time, i.e. a constant trip counts. Modulo scheduling can also handle trip counts that are not known at compile time as long as the trip count is loop invariant, i.e. it is known before the loop is entered. If it cannot be guaranteed that the trip count is greater or equal to the SC, a copy of the original loop has to be kept, since a modulo scheduled loop executes at least SC iterations.

7.5 Unrolling and modulo variable expansion

To solve the problem of overlapping live ranges depicted in figure 5, the current version introduces copy operations. Another way of avoiding overlapping live ranges is called *modulo variable expansion*. This technique uses loop unrolling to enlarge the II such that no live range exceeds the II. I.e. first loop unrolling is done and then the unrolled loop is modulo scheduled. The advantage is that no copy operations have to be inserted and additional ILP can be extracted. Another advantage is that the II can be increased such that it is greater or equal to the jump operation's latency if necessary, since this is the minimum length of the kernel. Experiments showed that for simple loops the MII is often less than the jump operation's latency. Unrolling would solve this problem without sacrificing efficiency. The disadvantage is the increased code size in the prolog, the epilog, and especially the kernel since the working set of operations is increased compared to the original loop which can lead to instruction cache misses.

8 Summary

Software pipelining is a scheduling technique for VLIW architectures that tries to optimize execution time of basic block loops by overlapping consecutive iterations. Modulo scheduling is a family of software pipelining algorithms that construct a kernel containing all operations of the original loop but from different iterations. The length of the kernel is called the initiation interval. Every kernel iteration starts a new iteration of the original loop. A prolog has to run before the kernel to reach a state where repeated execution of the kernel is equivalent to the original loop. The epilog finishes iterations that have not been finished by the kernel. The number of different iterations that are overlapped in the kernel is called the stage count. The length of the prolog and the epilog depend on the stage count. Prolog and epilog increase the code size. Software pipelining also increases register pressure since values have to be buffered between kernel iterations. The minimum initiation interval is estimated as a starting value for the initiation interval. The minimum initiation interval depends on the structure of the data dependence graph, especially the recurrences and the resources available. After every unsuccessful scheduling attempt the initiation interval is increased until modulo scheduling is possible or an upper bound is reached.

Swing modulo scheduling is a modulo scheduling heuristic that produces near optimal schedules with low register requirements compared to other modulo scheduling techniques. It achieves this by inserting the operations to the kernel in a sophisticated order. The order depends on the recurrences found in the data dependence graph. This order assure that dependent operations are scheduled closely together thus reducing live ranges of registers and reducing register pressure.

We implemented the swing modulo scheduling algorithm for the LLVM compiler framework. The algorithm can run after register allocation which has the advantage that the actual modulo schedule is generated, but additional dependences in the data dependence graph can degrade the result. We implemented also a version that runs before register allocation with the advantage of fewer dependences. Unfortunately LLVM does not support VLIW architectures such that we have to change a modulo schedule to confirm to LLVM's intermediate representation. The actual scheduling is done after register allocation by the conventional acyclic scheduler.

Both versions improve execution time for appropriate loops at the expense of increased code size and sightly increased register pressure.

References

[BE08] Florian Brandner and Dietmar Ebner. Compilation Techniques for VLIW Architectures, 2008.

[CLG] Josep M. Codina, Josep Llosa, and Antonio Gonzalez. A Comparative Study of Modulo Scheduling Techniques.

[DJG98] Amod K. Dani, V. Janaki, and Ramanan R. Govindarajan. Register-sensitive software pipelining. In *In Procs. of the Merged 12th International Parallel Processing and 9th International Symposium on Parallel and Distributed Systems*, 1998.

[FFY05] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.

[Fis81] J.A. Fisher. Trace scheduling: A technique for global microcode compaction. *Computers, IEEE Transactions on*, C-30(7):478–490, July 1981.

[Fis83] Joseph A. Fisher. Very long instruction word architectures and the eli-512. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pages 140–150, Los Alamitos, CA, USA, 1983. IEEE Computer Society Press.

[HBC98] W.A. Havanki, S. Banerjia, and T.M. Conte. Treegion scheduling for wide issue processors. *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 266–276, 1-4 Feb 1998.

[HMC+95] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Quellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: an effective technique for vliw and superscalar compilation. pages 234–253, 1995.

[Huf93] Richard A. Huff. Lifetime-sensitive modulo scheduling. In *In Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 258–267, 1993.

[Lam88] Monica Lam. Software pipelinig: An Effective Scheduling Technique for VLIW Machines. *In Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation*, 1988.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[Lat05] Tanya 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`.

[Llo96] Josep Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, page 80, Washington, DC, USA, 1996. IEEE Computer Society.

[MLC+92] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann. Effective compiler support for predicated execution using the hyperblock. *Microarchitecture, 1992. MICRO 25., Proceedings of the 25th Annual International Symposium on*, pages 45–54, 1-4 Dec 1992.

[Ogr04] Julia Ogris. Software Pipelining in a DSP C-Compiler. Master's thesis, Institut für Computersprachen, Abteilung für Programmiersprachen und Übersetzerbau der Technischen Universität Wien, August 2004.

[Rau94] B. Ramakrishna Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, New York, NY, USA, 1994. ACM.

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Wien, am 14. Oktober 2008

Benedikt Huber