Codegeneratoren

Sebastian Geiger
Twitter: @lanoxx, Email: sbastig@gmx.net

January 26, 2015

Abstract

This is a very short summary of the topics that are tought in "Codegeneratoren". Code generators are the backend part of a compiler that deal with the conversion of the machine independent intermediate representation into machine dependent instructions and the various optimizations on the resulting code. The first two Sections deal with prerequisits that are required for the later parts. Section 1 discusses *computer architectures* in order to give a better understanding of how processors work and why different optimizations are possible. Section 2 discusses some of the higher level *optimizations* that are done before the code generator does its work. Section 3 discusses *instruction selection*, which conserns the actual conversion of the intermediate representation into machine dependent instructions. The later chapters all deal with different kinds of optimizations that are possible on the resulting code.

I have written this document to help my self studing for this subject. If you find this document useful or if you find errors or mistakes then please send me an email.

Contents

1 Computer architectures **2**
1.1 Microarchitectures . . . 2
1.2 Instruction Set Architectures . . . 3

2 Optimizations **3**
2.1 Static Single Assignment . . . 3
2.2 Graph structures and memory dependencies . . . 4
2.3 Machine independent optimizations . . . 4

3 Instruction selection **5**

4 Register allocation **6**

5 Coallescing **7**

6 Optimal register allocation using integer linear programming **7**

7 Interproceedural register allocation **8**
7.1 Global Register allocation at link time (Annotation based) . . . 8
7.2 Register allocation across procedure and module bounaries (Web) . . . 8

8 Instruction Scheduling **9**
8.1 Phase ordering problem: . . . 9
8.2 Instruction scheduling for the IBM RISC System/6000 Processor (IBM) . . . 9
8.3 Efficient Instruction Scheduling for a Pipelined Architecture (HP) . . . 10

9 Register allocation & instruction scheduling **11**
9.1 Integrated Prepass Scheduling . . . 11
9.2 Dag-driven register allocation . . . 11
9.3 Register Allocation with Scheduling Estimate (RASE) . . . 11
9.4 Which algorithm should be prefered? . . . 12

10 Trace scheduling **12**
10.1 Variable Location Mapping (VLM): . . . 12
10.2 Delayed binding . . . 12

11 Software pipelining **13**
11.1 Modulo Variable Expansion . . . 13
11.2 Iterative modulo scheduling: . . . 13
11.3 Optimizations for processors with SIMD instructions . . . 14
11.3.1 SIMD Instruction Generation . . . 14
11.3.2 Alignment analysis . . . 14

1 Computer architectures

Codegenerators are dependent on the computer architecture they are designed for. Therefore a comprehensive understanding of the different concepts of computer architectures is requried. This capter can be roughly split into two main aspects: **microarchitectures** and **Instruction Set Architectures** sometimes abbreviated ISA. The concept of *microarchitectures* referes to the hardware part of a processor, that is its different parts such as execution units, pipelines, caches and such. The instruction set architecture defines the interface between the processor and its users (usually assembler programmers or compiler writers). It is easy to see that a codegenerator is dependent on the instruction set architecture it was designed for. For example a different codegenerator is needed for CISC and RISC computers. But a codegenerator developer also needs a good understanding of the underlying microarchitecture in order to better optimize the generated code. An example for such optimizations is the use of instruction level parallelism or code scheduling to achieve a higher pipeline utilization. Below the most important concepts of microarchitectures and ISA are explained.

1.1 Microarchitectures

Each processor is composed of many different hardware parts, that are all responsible for different things. The following list gives an overview of the most important parts and their functions:

- **Caches:** Store some of the program code and data from memory to allow faster access. Separate instruction and data caches can allow simultaneous instruction and data access every cycle.
- **In/out of order execution:** Realized by reorder buffers. Instructions are dispatched to the reorder buffer in-order but the executin units can execute the instruction out-of-order.
- **Register renaming:** Register renaming minimizes architectural resource dependencies, namely WAW and WAR. Rename buffers are used to implement register renaming.
- **Reorder buffer:** Saves instructions in-order, is used by the execution unit which can execute instructions out-of-order.
- **Precise interrupts:** An interrupt is precise if the state of the processor includes the data of all instructions before the instruction that caused the interrupt and no data from the instruction causing the interrupt or later instructions. Processors that do not have precise interrupts are quite difficult to program.
- **Completion stage:** Important for precise interrupts. Also separates completion stage and write-back stage to support load- with-update instructions.
- **Branch prediction:** Used to avoid pipleine stalling. CPU predicts the branch before this information is available.
- **Branch target address cache:** Used to predict the target address of a branch or jump instruction. The BTAC is accessed with the fetch address and returns the target address.
- **Branch history table:** Stores information about whether a branch was taken or not, depending on the complexity of the branch history table it is also possible to store patterns, such as, that a branch was only taken every second time.
- **Reservation station:** Used to keep an instruction until its operands become available (The PowerPC 604 has two reservation stations for each execution unit). This way instructions can be dispatched even if their operands are not yet available.
- **Superscalar architecture:** A superscalare processor executes more then one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundand functional units on the processor, this can include different functional units such as branch, integer, multiply, floating-point units, but also redundand units of the same type such as four integer units. Examples for superscalar architectures in common processors are:
  - Hyperthreading (duplicates register space for the thread context)
  - Multiple execution units (often combined with out of order execution)

  - SIMD
  - VLIW

- **Pipelining:** Increases the **frequency** of the processor, but also the **latency**, one instruction requires multiple clock cycles. Instructions handling is split into several stages where each stage can handle one instruction at a time. The instruction passes through several stages before its computation is complete. Typical stages are: (Fetch, Decode, Dispatch, Execute, Write-back, etc.). The latency usually covers only the execution stage. Example: In the first cycle an instruction is fetched by the fetch stage, then in the next cycle an new instruction is fetched by the fetch station and the first instruction moves on to the decode station where it is decoded. **Problems** of pipelining are called **hazards** and are caused by the increased latency of instruction results. Such a hazard often appears if a second instruction depends on the result of a first instruction. It can be solved with different techniques such as out-of-order execution. Another problem appears in combination with branch prediction. If a branch is wrongly predicted, then the pipeline becomes invalid and must be restarted, which leads to a penalty of several cycles.

1.2 Instruction Set Architectures

An instruction set architecture defines the programming interface of the processor. It includes aspects such as instructions, registers, data types and memory access, as well as interrupt handling and exception handling. The instruction set architecture also defines how external I/O is done. Common instruction set architectures are:

- **RISC** Reduced Instruction Set Computer
- **CISC** Complex Instruction Set Computers:
- **VLIW** Very Long Instruction Word: Architectures which combine multiple instructions for parallel execution.

There are two very important differences between CISC and RISC, one is the way memory access works, the second is the instruction width. RISC uses a load-store architecture in which instructions that operate on the memory are only used to load data into registers or store data from registers in memory, while all computation happens on registers. CISC on the otherhand also allows operations to modify data directly inside the memory. The second big difference is the width of instructions. RISC has a fixed instruction width (such as 4 bytes) where as CISC has a variable instruction width.

It is also important to note that pipelining and superscalar architectures can be problematic on CISC processors due to the complexity of CISC instructions. Therefore CISC processors have introduced a concept called micro-operations, where instructions are first converted into one or more micro-operation and are then dispatched an executed by the processor. This fact however does not turn a CISC processor into a RISC processor, it is just an implementation detail of the microarchitecture of CISC processors.

Instruction level parallelism

Instruction level parallelism (ILP) referes to the fact that many instructions are independent of each other and can be executed simulaneously. It is used as a measure for the degree of parallelism in a given program. There are different techniques to exploit this inherent parallelism in programs.

- In **Software** ILP can be exploited through a rescheduling of the instructions by the compiler.
- In **Hardware** the ILP can be exploited by many techniques of the microarchitecture such as *instruction pipelining*, *superscalar execution*, *out-of-order exectuion*, *register renaming* and *branch prediction*.

2 Optimizations

2.1 Static Single Assignment

Static single assignment (SSA) is an optimization required to enable most other optimizations, is a form where each variable has only a single value assignment. Additional every variable must be defined before it is used. Every code can be transformed into SSA form by adding indexes to variables:

$$i = 5 + 7$$
$$z = 8$$
$$i = i + z$$

becomes:

$$i_1 = 5 + 7$$
$$z = 8$$
$$i_2 = i_1 + z$$

When dealing with branches φ-nodes are added to merge variables from separate branches:

```
if(a)
    i1 = 10
else
    i2 = 20
i3 = φ(i1, i2)
```

2.2 Graph structures and memory dependencies

Graph data structures are also required for most optimizations and contain information about data dependencies between instructions, they are also often called **dependency graphs**. In general graphs can be separated into two categories: acyclic graphs and cyclic graphs. **Acyclic graphs** are often used to model code inside basic blocks[1] or extended basic blocks. On the otherhand **cyclic graphs** are usually used to model loops or recursion and usually extend over a whole proceedure and not just a basic block. In such graphs instructions (or operations) are represented as vertices and data dependencies as edges. Depending on where the graph is used vertices and edges can be annotated with additional information such as delayes between instructions.

The data dependencies that form the edges of a graph can be categorized into three kinds of dependencies:

- true dependency or **read after write** (RAW)

$A = B + C$	write A
$D = A + 2$	read A

- anti dependency or **write after read** (WAR)

$A = B + C$	read B
$B = D - 2$	write B

- Output dependency or **write after write** (WAW)

$A = B + C$	write A
$D = A + 2$	
$A = E + F$	write A

Many of the optimizations that are introduced in this and the later chapters require a detailed knowledge of the data dependencies between instructions.

2.3 Machine independent optimizations

Machine independent optimizations are performed on the intermediate representation and are independent of the machine (i.e the instruction set architecture) being used.

- Scalar optimizations:
  - **Constant Evaluation:** resolves expressions of multiple constans such as 5+7 which is resolved to 12

[1] The concept of basic blocks is introduced in the compiler construction lecture and refers to a block of code with only one entry and one exit point. That means in particular that basic blocks cannot contain loops or branches.

- **Constant Propagation:** propagates the constant to places in the code where variables refer to constants
  - **Copy Propagation:** replaces the use of a variable with the use of another variable of equal value
  - **Common subexpression elemination:** Opposite of **rematerialization**
  - **Scalar expansion:** Propmote scalar to vector/array inside loop (used in SIMD and VLIW)
  - **Strength reduction:** replaces expensive instructions by cheaper ones (eg. replace $5 * a$ with $a >> 2 + a$ since shift and add are cheaper than multiplication). Another meaning of strength reductions can concern address optimizations in connection with indiction variables.

- Progam optimizations:
  - Dead code elemination (reachable vs. useless code elimination)
  - Function inlining (Speed) vs. Procedural abstraction (Code size)
  - Tailcall elimination: Resolves recursion into iterative loops (very common in logic programming)
  - Peephole optimization (very small optimization 2-3 instruction, hard to analyze)

- Induction variable elimination: is an optimization that tries to identify and remove the indiction variables inside loops.
  - The improved loop uses pointer arithmetic instead of an index variable
  - Invariant code motion moves code that does not depend on the loop body out of the loop body
  - Use information about induction variable for optimization

- Loop optimizations:
  - Loop interchange (change inner and outer loop), the inverse function is still loop interchange
  - Loop fusion vs. Loop fission (Splitting) (or distribution)
    - Loop peeling is special case of loop splitting that peels of one or more iterations from the beginning or the end of the loop and performs these operations separately before or after the loop.
  - Loop collapsing (collases inner and outer loop) vs. Strip mining
  - Strip mining transforms a single loop into a doubly nested loop where is incremented by a certain block size and the inner loop is incremented by one:

```
1   for(int j=0; j<N; j+=32) {
2       for(int i=j; i<min(j+31, N); i++) {
3       }
4   }
```

  - Loop vectorization (SIMD, VLIW): The compiler analyzes the dependence graph and if no loops are found it can make use of special vector instructions of the target processor.
  - Loop concurrentization: Loops are split into several partitions and each partition is computed on a different processor concurrently.

- Reorder transformation (like code scheduling but higher level, e.g. compiler frontend not backend). Uses sophisticated dependency analysis.

3 Instruction selection

Instruction selection is a process which transforms the intermediate representation of the program into object code, by selecting a machine instruction for each computation. The resulting object code after instruction selection is therefore machine dependent. Instructions may have associated costs and the goal of instruction selection is to find a sequence of instruction which have minimal costs.

There exist several implementations for instruction selection:

- **iBurg** is an improved version of **Burg** which was not discussed in the lecture. Its main advantage is that its much simpler to implement (about 700 lines of code). Burg computes costs at generation time (compile-compile time). In theory iBurg could support dynamic cost computation at instruction selection time (compile time), but does does not do so in practice for backwards compatibility with Burg. Instructions are selected through tree pattern matching and dynamic programming. The goal of tree pattern matching is to find a minimal cost cover for the data flow tree that represents the operations of the intermediate language.

- **BEG**

  BEG stands for Back End Generator. It is not only used for instruction selection but also includes register allocation. The important aspects of BEG are:

- It is more advanced then iBurg
- Has a more powerful rule engine and supports **dynamic costs** and **conditions** for instruction selection.
- Can do **register allocation**, but only locally on basic block level
- Has two types of register allocation (on the fly, general register allocation)

- Dag driven instruction selection using PBQP:

  Instruction selection is modeled as a partitioned boolean quadratic problem, which is an NP-Complete problem. PBQP tries to find a cost minimal assignment h of variables X to their respective domains D:

  $$h : X \to D$$

  At the core of PBQP two functions are used, a local cost function c and a related cost function C, the sum f of both forms the cost of an assignemnt for variables to their respective domains:

  $$f = \sum c_i + \sum C_{ij}$$

  The instruction selection with PBQP works like this. For each node u in the input DAG there exists a variable x_u which is mapped to an operation out of a list of possible operations for this node. The possible operations are defined by a normalized graph grammar which consists of base rules and chain rules. The local cost function is defined as a cost vector c_u that defines the cost of assigning each operation to x_u. Additionally the related cost function is modelled as a matrix C_{uv}. It is used to *"ensure consistency among base rules and to account for the costs of chain rules"* [9]. Its values are either **zero** for matching base rules, **infinite** for incompatible base rules or a value which is the **sum of transitive chain rule costs** in order to make two non-matching base rules compatible. In summary: *"A solution of PBQP determines which base rules and chain rules are to be selected"* [9].

  The PBQP based instruction selection also provides a solution for complex patterns such as "div-mod" or "auto-increment", however this can lead to cycles in the graph:

  - SSA graphs model the data flow through registers, but not the dataflow through memory. That is, SSA graphs do not reflect memory dependencies. This can lead to cylces which must be broken up by finding a topological order among the patterns.
  - Instructions with multiple outputs such as "div-mod" or "auto-increment for pointers" can cause cycles in the graph. This can happen if "a set of operations in the SSA graph is matched by a pattern for which there exists a path in the SSA graph that exits and re-enters the complex pattern within the basic block. To ensure the existence of a topological order among the chosen productions those cycles must be broken up. To break up the cycles the SSA graph is augmented with additional edges representing potential data dependencies [9].

An important difference between the different instruction selectors is when costs are computed. Burg and iBurg do it at compile-compile time, where as BEG and PBQP do it are compile time.

4 Register allocation

There are three important algorithms. Chaitin and Briggs which use a graph coloring approach and Chow-Hennesey which use a priority function.

- **Chaitin:** Builds a dependency graph and then tries to simplify the graph. Simplification happens by removing nodes from the graph that have a *degree* which is less than the amount of available registers. The nodes are placed on a stack for later coloring. If the graph reaches a state where no such nodes can be removed, then Chaitin spills one node. The node to spill is selected through a cost function that assigns each node a weight (cost divided by degree of the node) and then chooses the one where the cost for spilling is as low as possible. If a node is spilled, then the graph is rebuild without that node and the algorithm is restarted. Otherwise when all nodes have been removed from the graph the coloring phase starts. Each node is removed from the stack added back to the graph, then it is assigned a color that is different from all its neighbours. This approach prevents Chaitin from successfully coloring certain graph patterns such as the diamond pattern (cycle of four nodes) with a minimum of 2 registers. Instead Chaitin needs one additional register.

- **Briggs:** Just like Chaitin, but it performs the spilling phase after the coloring phase, this has the effect that Briggs can color diamond patterns in the graph with a minimum of 2 registers instead of 3 like Chaitin does.

- **Chow Hennesy:** Chow Hennesy assume that all variables reside in memory and are loaded into registers as required. This is the reverse concept of Chaitin and Briggs. A priority function is used to calculate the saving if a variables is kept in a register. The variables which have the greatest savings are then selected to be stored in registers. The algorithm reserves 4 registers for the code generator. If no suitable register is found, then live range splitting is applied.

Differences between the algorithms:

- Both Chaitin and Briggs algorithms need multiple iterations where as Chow-Hennesy finishes in one iteration.
- Chaiting and Briggs do the life range analysis on instruction level where as Chow-Hennesy perform life range analysis on basic block level.
- Chaitin and Briggs do register allocation for all registers, where as Chow-Hennesy only do register allocation outside of basic blocks.

Live range splitting: TODO, Live range splitting can be added to Chaitin and Brigg but was not present in the original algorithms.

Calling conventions: Calling conventions are conventions about how registeres are used and who is responsible to save and restore them. There are **caller-saves** and **callee-saves** registers. The conventions are a result of a multi-step proceedure. First the architecture has an impact on the calling conventions, then the operating system and lastly the used programming language. Calling conventions are part of the ABI and if not taken care of they my cause incomparibilities between programm binaries such as if libraries are used or programms written in different languages interact.

Pre-coloring: Precoloring is used to assign correct colors to argument registers to avoid unnecessary copy instructions.

5 Coallescing

Coallescing is a form of copy elemination that finds registers that are copies of each other and removes the copy instruction, thus putting the values into the same register. Coallescing may affect the colorability of an interference graph (both negative and positive results are possible).

- Aggressive coallescing (used by Chaitin): Remove all possible copy instructions, may result in increased spilling.
- Conservative coallescing: Performs coallescing only when graph colorability is not affected. Was intended to be an improvement but later analysis showed that the algorithm actually performed worse than aggressive coallescing.
- Iterative coallescing: Eliminates more copies by interleaving conservative coallescing with simplification phases (like Briggs). Also has worse results than aggressive coallescing.
- Optimistic coallescing: Aggressive coallescing, but the algorithm memorizes what has been coallesced. If coloring is not possbile and nodes need to be spilled, then the coallscing is reversed by live range splitting and then only parts of the nodes are spilled if possible. **Best results**.

6 Optimal register allocation using integer linear programming

Performance overhead of spill code is minimized by using integer linear programming to achieve near optimal results. Since the problem is NP-complete an optimal solution will be instractable for very complex problems. However the Optimal Register Allocator (ORA) solver still produces very good results that are very often within 1% of the optimal solution. However in order to define what optimal means we need to know which optimizations are being done by the register allocator.

- **What does the register allocator do and not do?**
  1. The register allocator does not reorder instructions.
  2. The register allocator adds only spill load instrucitons, spill store instrucitons, and rematerialization instructions.
  3. The register allocator removes only copy instructions and instructions defining symbolic registers that are rematerialized.
  4. All controll-flow paths throguh a function can potentially execute.
  5. At a given location, the cost of a spill instruction is a fixed number of cycles (based on the instrucitons' type) times the location's estimated execution count.
  6. A basic block's estimated execution conut equals the sum of the countes exeiting the block, i.e. estimated execution counts satisfy Kirchhoff's current law over the control flow graph.

- **What means rematerialization?** Sometimes its cheaper to recompute a value than to store it in a register. This is often the case if a value can be recomputed with a single instruction. rematerialization is often used in address calculation.
- **Improvement of the paper:** The authors improved their own paper after 6 years and reduced the complexity of the algorithm from n^3 to $n^{2.5}$. They also improved the speed by a factor of 1000x (10x CPU, 10x better solver, 10x improved model).

TODO: Describe the three modules (steps): Analysis, Solver and Rewrite module

7 Interproceedural register allocation

Interproceedural register allocation (which is also often called global register allocation) performs register allocation accross proceedures or compilation units and not just within a single function as is done in global register allocation. This solves the problem that different functions may assign the same global variable to different registers or the same register to different local variables, which causes unnecessary copy instructions.

7.1 Global Register allocation at link time (Annotation based)

Global here means interproceedural accross all compilation units. The register allocation is split into three parts:

- Compiler
- Register allocator
- Linker

The compiler does as much work as posible in order to maximize the benefits of seperate compilation such that in the linking phase there is not too much work required. The compiler inserts annotations into the object code which describe how variables are used. Annotations provide information to the linker about how to replace variables and how to rewrite the code.

There are the following annotations:

- REMOVE
- STORE
- LOAD
- OP1, OP2, RESULT

The register allocator generates a call graph and uses the information provided by the compiler assign variables to registers. It does this by building groups of variables that are not active at the same time and then assigns are usage frequency to those groups. The same happens for global variables. Finally the groups with the highest freuqencies are allocated to registers. After variables have been assigned to registers the linker can use this information rewrite the code and to apply the actions that have been stored in the annotations. While doing this it must take care to adjust addresses of instructions or data if a previous instruction is being removed.

Additional considerations concern initialization and recursion.

Initialization

Special care must be taken to initialize global variables and proceedure parameters. For global variables an INIT action is placed at the beginning of the program and runs when the program is invoked. The INIT action adds instructions to copy the initial value of global variables from memory to the respective register.

Recursion and indirect calls

Recursion means that the same proceedure my be called repeatedly befor the first invokation has been completed. Therefore the program needs to take care to save the necessary registers of local variables before the next invokation of the proceedure. The question is who should be responsible to save the local varibles of the proceedure. It turns out that it is not enough if each recursive proceedure is responsible to save its local variables because this would not archive local variables of other proceedures in the call chain. The solution is to let the proceedure that makes the recursive call save the local variables of all proceedures in the call chain. The same solution is used for indirect calls.

7.2 Register allocation across procedure and module bounaries (Web)

This has two important parts, the web algorithm itself and clustering (e.g. spill code motion).

Web algorithm:

Performs several steps:

1. Reads in source files and produces intermediate representation files.
2. Creates program database.
3. Uses Program database and transforms intermediate representation files into object files.
4. Linking of the object files.

The web algorithms creates a table with L_REF, P_REF, and C_REF entries from which a web interference graph is generated. This is then used to apply a coloring on the graph that determines which register is being used for each global variable.

Clustering (Splill code motion):

The idea is to move spill code for callee-saved registers inside a cluster from one node to its dominating node. If the node is being called more than once, then the overhead of the spill code is reduced. Same for caller-saved registers which are moved down to lower nodes in the same cluster.

8 Instruction Scheduling

Instruction scheduling is an optimization that deals with rearranging the instructions of a program to better exploit the inherent parallelism of the instructions. The goal is to move instructions that depend on each other further appart and instead insert independent instructions inbetween. This prevents that dependent instructions block the processor pipeline and allows a more instructions to execute simultaneously. The simplest form of instruction selection is called list scheduling because from a list of possible instruction that can be scheduled one with the highest priority is selected based on some kind of heuristic. There exist two papers that deal with list scheduling which both present different heuristics for the problem. The list scheduling algorithms presented in the papers have a worst case complexity of $O(n^2)$ needed to build the dependency graph. However in practice there exists also a linear time algorithm.

The improved **linear time algorithm** works as follows. The algorithms goes through each instruction and each time a register is defined the algorithm saves which instruction defined the register. Each time a register is the corresponding instruction can be looked up and the algorithm adds a dependency to the graph. The only problem with this algorithm is that the initialization of the corresponding data structure may be expensive. However this can be optimized when using virtual registers by storing from which instruction to which instruction a basic block extends. Based on these limits it can be easily determined if an instruction is still independent. Otherwise the datastructures would need to be rebuild for each basic block.

8.1 Phase ordering problem:

A problem between instruction scheduling and register allocation is that both optimizations can have negative effects on the other one, thus it is difficult to decide what should be done first - register allocation or instruction selection. This problem is called the *phase ordering problem.* The list scheduling algorithms below attempt different solution but still perform both optimizations separately. A better solution is discussed in the next section which presents algorithms that perform instruction scheduling and register allocation together at the same time.

The impact of each optimization to the other is as follows:

- Register allocation serializes the code and removes opportunities for scheduling (less instruction level parallelism).
- Instruction scheduling increases the length of the live ranges (register pressure), causing more interferences and thus requires the use of more registers.

8.2 Instruction scheduling for the IBM RISC System/6000 Processor (IBM)

The algorithm which is described in this paper performs the instruction scheduling twice, once before register allocation and once afterwards. The first instruction scheduling has more freedom since register allocation has not yet been done, but register allocation may insert spill code and delete move instruction which make a second run of instruction scheduling necessary. The algorithm itself works as follows:

- It uses a dependency graph which is constructed for each basic block. The dependency graph has one node for each instruction and adds a directed edge between two instructions if one needs to preceed the other.
- Edges are labeled with the delay between instructions.
- After the dependency graph has been build, nodes are labled upwards with the **maximum delay** from the node to the end of the basic block (red digits in figure 1).

- The algorithm keeps a **current time** counter, which is inizially zero. It is increased by the execution time of each scheduled instruction, which is usually one.
- Nodes are labels with **earliest time** value which is calculated from the current time of the last scheduled predecessor plus the delay of that instruction to its successor. For example the node L.B has a delay of one and is scheduled in step 2 where the current time is increased to 2, so the ADD node would be labled with an earlies time of 3 (2+1).

Figure 1 shows the a dependency graph from the paper but adds the values of the maximum delay and a table that shows how the current time is updated and which instruction is selected in each step.

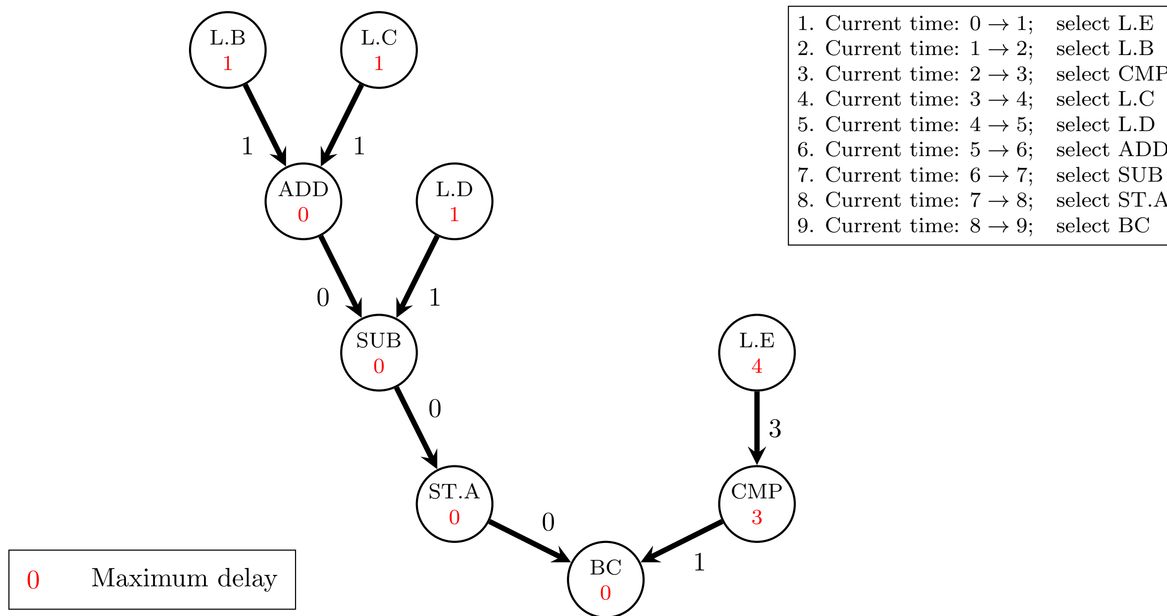

Figure 1: Dependency graph constructed by IBM instruction scheduling algorithm

When deciding which node to select in each step, the algorithm uses an eight step heuristic. Since there is no planning or lookahead involved the algorithm is not guaranteed find the best solution. There are the following eight heuristics:

1. Initialize the set of all those instructions that have not yet been selected, and that have no predecessors in the dependency graph (these are the "legal" ones).
2. Refine the subset to those instructions whose earliest time has arrived or, if none, those with the smallest earliest time.
3. If one or more instructions have been selected, and if the current subset contains one or more instructions of opposite type (fixed/floating) from the last one selected, then refine the current subset to those of this opposite type.
4. Refine the subset to those of maximum total delay along the path of each instruction to the end of the basic block.
5. Refine the subset to those of minimum "liveness weight".
6. Refine the subset to those with greatest uncovering".
7. Refine the subset to the unique instruction that came first in the original ordering.

8.3 Efficient Instruction Scheduling for a Pipelined Architecture (HP)

The general idea is similar to the IBM algorithm. The algorithm works as follows:

1. Build a scheduling dag over the basic block
2. Put the roots of the dag into a candidate set
3. Select the first instruction to be scheduled based on the heuristics below
4. While the candidate set is not empty do the following:

(a) based on the last instruction scheduled and the heuristics select and emit the next instruction that should be scheduled

(b) delete the instruction from the dag and add any newly exposed candiates

The following heuristics use the following criteria to determine the priority with which a candidate should be scheduled:

1. Whether an instruction interlocks with any of its immediate successors in the dag.
2. The Number of the immediate successors of the instruction
3. The length of the longest path from the instructions to the leaves of the dag.

The instruction scheduling in is performed before register allocation. The paper mentions register allocation but claims that "serializing definitions does not unduly restrict [the] code".

9 Register allocation & instruction scheduling

This section introduces three methods which combine instruction scheduling and register allocation in order to solve the phase ordering problem. The first paper presents two algorithms the Integrated Prepass Scheduling (IPS) and the dag-driven register allocation. The second paper introduces a method named Register Allocation with Scheduling Estimate (RASE).

9.1 Integrated Prepass Scheduling

The main issue of the phase ordering problem is that register allocation reduces opportunities for scheduling while instruction scheduling increases the length of live range making register allocation more difficult. The IPS algorith solves this problem by using two strategies that can be selected depending on what strategy is more appropriate. The two strategies are:

- CSP (Code scheduling for pipelined processors) reduces pipeline delays but can increase the lifetime of registers. It is used if enough free registers are available.
- CSR (Code scheduling to minimizes register usage) is used when the number of registers is low to control register usage. It tries to "find the next instruction which will not increase the number of live register or if possible, decrease that number".

IPS uses a simple heuristic to switch between the two strategies, it keeps track of the number of available registers in a variable named AVLREG, if the number of free registers drops below a certain threshold, then it switches to CSR. When AVLREG has increased above the threshold then it switches back to CSP.

9.2 Dag-driven register allocation

Dag-driven register allocation tries is a form of postpass scheduling to keep the graph flat and wide. It also does not insert any additional spilling instructions into the code. The algorithm introduces two measures for the graph the width and the height:

- The **width** is defines as the maximum number of mutually independent nodes. Wider graphs indicate higher parallelism.
- The **height** is the length of the longest path. If the graph is too high then code scheduling becomes less efficient.

The goal of the dag-driven register allocator is to reduce the height of the graph and limit the width of the graph to a number that is less or equal to to the amount of registers available.

TODO: Explain better

- Makes uses redundant dependencies when allocating registers.
- Tries to balance the graph
- Inserts dependencies in such a way that the critical path does not become longer

9.3 Register Allocation with Scheduling Estimate (RASE)

The RASE approach is an integrated scheduling and register allocation method. The idea behind RASE is to perform a pre-scheduling in order to calculate schedule cost estimates that enable the register allocator to make a decision about how many registers it is going to use. Essentially RASE splits the available registers into two parts, one part for the register allocator, and one part for the instruction scheduler. The RASE algorithm has three phases:

- Pre-scheduling
- Global register allocation
- Instruction scheduling with local register allocation

The **first phase** is the pre-scheduling, which calculates the schedule cost estimate. During the pre-scheduling, the order of the instructions is not changed.

The **second phase** performs a *global* register allocation using the schedule cost estimate, but leaves enough free registers for the last phase. The schedule cost estimate allows the register allocator to estimate, how the cost of local register allocation increases, if it uses more registers to allocate *global* pseudo-registers. In the second phase the register allocator is only responsible for allocating global pseudo-registers to physical registers, the allocation of *local* pseudo-registers is defered until the third phase.

The **third phase** performs the instruction scheduling and does the *local* register allocation. This register allocation must be within the limits for *local* registers that have been determined in the second phase.

How does the schedule cost estimate function work? The schedule cost estimate is a quadratic function (of *regressive* form). It takes as input the number of local registers and returns as output the estimated machine cycles:

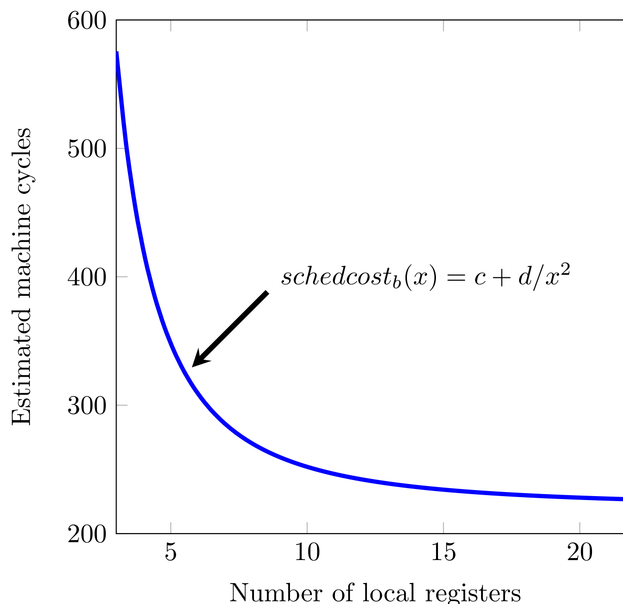

Figure 2: Schedule cost estimate function

The coefficientes c and d are computed as part of the pre-scheduling. Essentially this function allows the register allocator to estimate how much the required machine cycles are incresed, due to loss of parallelism, if the available local registers are reduced. As can be seen in Figure 2, if the register allocator leaves 15 registers to the third phase, then the total execution cost for that basic block it not very high, however if only 4 or 5 registers are left to the third phase, then the cost increases quite significantly.

9.4 Which algorithm should be prefered?

When comparing the three algorithms one can say that IPS is the "best" algorithm, because it is realively easy to implement and produces results which are comparable to the other two solutions. The IPS algorithms can be easily implemented in existing register allocators such as Chaitin or Briggs. In contrast the DAG driven algorithm and RASE are both quite complicated to implement but do not produce significantly better results.

10 Trace scheduling

Trace scheduling attempts to provide another solution for the phase ordering problem. A trace is **a sequence of instructions that can have exit and entry points but that must not have any loops**. The trace scheduler

orders traces by their frequency and schedules the most frequent trace first. Inner most loops usually contain the traces with the highest frequency and are thus scheduled first. Since the instruction scheduler is also responsible for assigning variables to registers, this scheduling approach effectively gives priority to those variables, which are used with the highest frequency. The advantage of this approach is that at the beginning all registers are available for use by the scheduler, giving the scheduler complete choice over registers with the highest frequency.

Since traces are scheduled individually of each other care must be taken if two traces merge with each other or split from each other. This can either happen at split nodes, when one trace leaves from the other, or at join nodes, when one trace joins into another. Since the traces are scheduled and register allocated individually there needs to be a machanism to communicate register allocation decisions between traces. This is done by adding special nodes named *Value Location Mapping* (VLM).

10.1 Variable Location Mapping (VLM):

A VLM is placed at the split or join nodes between two traces and contains the information about already allocated variables. So when a trace is scheduled that contains a variable which has already been allocated, then the scheduler can take this decision into account. It tries to ensure that individually scheduled traces reference their values at the same locations to avoid unnecessary data motions such as register moves, spills and restores. If a VLM appears at the beginning of a trace, then the scheduler must make sure to read values that appear in the trace from the locations specified in the VLM. On the other hand, if a VLM appears at the end of a trace, then the scheduler must make sure to *satisfy* the references in the VLM, that is, it must store values that appear in the trace at the locations specified by the VLM. Sometimes it may not be possible to satisfy the constraints in a VLM. This can happen if there is a VLM both at the top and bottom of a trace and both VLMs have conflicting locations for the same variable. If it is not possible to satisfy the location constraints in a VLM, then the scheduler must insert **compensation code** to move or copy values to the correct locations.

10.2 Delayed binding

Delayed binding is used because some values have live ranges that go through a trace but the value is not actually used inside the trace. *"A delayed binding is a sort of pseudo-location that may be assigned to an unreferenced value by the instruction scheduler."* Delaying the variable binding until there is an actual reference to it helps to preserve registers for more important values. When a delayed binding is bound to an actual location it is determined if there exists an unused register that is free through out the whole live range of the value, otherwise the value is bound to memory.

Paper Evaluation: The evaluation of the trace scheduling paper is not useful, because the results are only compared to themselves.

11 Software pipelining

Software pipelining is a form of instruction scheduling with the goal of scheduling the different instructions of a loop in a way, such that multiple loop iterations can be active at the same time.

11.1 Modulo Variable Expansion

One problem in software pipelining is that a variable is defined and then used two cycles later. If this variable remains in the same register during every loop iteration, then each iteration must take two cycles. However if copies of the variable are kept in different registers for consequtive iterations, then it is possible to start a new operation in each cycle. This concept of using multiple registers for the same variable is called **modulo variable expansion**.

11.2 Iterative modulo scheduling:

Different scheduling strategies exist in software pipelining, the iterative modulo scheduling is a form of the "schedule-then-move" stategy. The goal of iterative modulo scheduling is to find a schedule for the instructions of a loop body, such that the loop can be repeated at regular intervals, without causing any conflicts between the instructions. There are two types of conflicts that can occur, a resource conflict or a dependency conflict:

- Resource conflict: The same processor resource such a bus or a pipeline stage are in use at the same time.
- Dependency conflict: Can be an inter-iteration-dependency if there exists a dependency between instructions in two different iterations of a loop or an intra-iteration-dependency if there is a dependency between two instruction in the same loop iteration.

Before the actual iterativev modulo scheduling is performed several other optimizations and transformations are performed, one of them is IF-conversion. **IF-conversion** removes all branches except the loop closing branch and

converts the code into a single basic block. The different branches of the original code are instead expressed by data dependencies involving predicates.

Iterative modulo scheduling introduces the notion of an **initiation intervals (II)** which is the interval between the start of a new loop iteration. An II of four means that every 4-th instruction a now loop iteration begins. A **minimum initiation interval (MII)** is calculated and used to calculate the minimum length of a schedule. To compute the actual II a candidate II is used that is initialy set to the MII and then step wise incremented until a suitable II is found. The MII is the maximum of the Resource constraint MII and Recurrence constraint MII values:

- The **Resource-constrained MII** used to avoid that two instructions block the same processor resource at the same time. For example if an add instruction requires to use the result bus in the fourth cycle and a multiply instruction requires the result bus in the sixth cycle, then an add cannot be sheduled two cycles after a multiply. The ResMII can be computed by performing a bin- packing of the reservation tables for all the operations.
- The **Recurrence-constrained MII** is used prevent data dependencies between the same operations in different loop iterations.

The iterativ modulo scheduling starts with the initially computed II and tries to find a valid schedule for this II and a given budget. If no schedule can be found the II is increased by one and the proceedure is repeated until a schedule is found. There are some differences between the traditional acyclic list scheduling and iterative modulo scheduling:

- Instructions can also be unscheduled and rescheduled, thus operation scheduling instead of instruction scheduling is performed. Operation scheduling picks an instruction and scheduls it at a time slot that is both legal and most desirable.
- Instructions do not become "ready" after their predecessors have been scheduled. Instead the algorithm keeps track of which instructions have never been scheduled and schedules instruction based on a priority function.
- If an instruction is unscheduled it is immediately rescheduled.

11.3 Optimizations for processors with SIMD instructions

Processors with SIMD instructions allow to execute the same sequence of instructions on multiple instances of data such as an array. For this purpose the data must be naturally aligned on the block size. If the alignment cannot be statically determined at compile time then the compiler needs to insert dynamic runtime checks that verify if the pointers are aligned. If the pointers are aligned then SIMD instructions can be used, otherwise the memory is accessed sequentially. These dynamic checks increase the program size and impact the execution speed. Not only the dynamic checks increase program size, but also the fact that there need to be two versions of a loop, one version with SIMD instruction and another version which accesses memory locations sequentially.

Describes an analyzation technique to statically identify the alignment of C pointers. The alignment information can then be used to reduce dynamic alignment checks and thus reduce the overhead of those checks. About 50% of all pointers can be statically identified. The first part of the paper describes the code generator that is used to generate SIMD instructions.

11.3.1 SIMD Instruction Generation

The SIMD instruction generation consists of two phases:

- Phase 1 deals with the generation of logical and arithmetical SIMD instruction inside loops that process arrays.
- Phase 2 deals with the generation of LOAD and STORE SIMD instructions for data that is used by the logical and arithmetical SIMD instructions which have been generated in phase 1.

In order to generate logical and arithmetical SIMD instructions from loops the lop may be unrolled a few times. For example if a loop processes 16bit elements and the SIMD operands are 32bit then the loop needs to be unrolled two times. If the loop count is not a multiple of two then a pre- or postloop is added which is executed $N \mod K$ times, where K is the unroll count and N the loop count. Depending on alignment information a preloop or a postloop is added.

For the generation of SIMD instructions an acyclic dependency graph is generated from the unrolled loop version. The nodes in the graph are *s-nodes* (statement nodes) and *b-nodes* (groups of basic blocks). The algorithm first schedules s-nodes that are not candidates for SIMD instructions as well as all b-nodes. Then it tries to find a series of s-nodes which are structurally equivalent and combines them into a SIMD instruction. To identify the instructions that can be combined into SIMD nodes, the algorithm builds *"all possible combinations of SIMD candidates and rates them according to the number of resulting SIMD expressions and the number of adjacent subwords in the SIMD expressions"*. Scheduled nodes are removed from the dependency graph and the proceedure is repeated until the graph is empty.

The algorithm als applies scalar expansion and accumulator splitting. Scalar expansion replaces a scalar by an array which contains copies of the scalar, such that it can be used with SIMD instructions. Accumulator splitting is used in reductions such as calculating the sum of an array. The array is split into several smaller arrays for which the sum can be calculated in parallel, the results is then a smaller array from which the final result can be computed.

11.3.2 Alignment analysis

- Intra procedural alignment analysis
- Inter procedural alignment analysis

References

[1] The PowerPC 604 RISC Microprocessor (S.P. Song & M.D. Denman), 1994

[2] Alpha AXP Architecture (R.L. Sites), 1993

[3] Discorvery 6 - Intels Sternenschiff P6 im Detail (G. Schnurer), ct' 4/1995

[4] Advanced Compiler Optimizations For Supercomputers (D.A. Padua & M.J. Wolfe), 1986

[5] The CONVEX FORTRAN 5.0 Compiler (R. Mercer), 1988

[6] Effectiveness of a Machine-Level, Global Optimizer (M. S. Johnson & T.C. Miller), 1986

[7] Engineering a Simple - Efficient Code Generator Generator (C.W. Fraser, D.R. Hanson, T.A. Proebsting), 1992

[8] BEG - a Generator for Efficient Back Ends (H. Emmelmann, F-W. Schröer, R. Landwehr), 1989

[9] Generalized instruction selection using SSA graphs (Andreas Krall et all.), 2006

[10] Coloring Heuristics for Register Allocation (P. Briggs, K.D. Cooper, K. Kennedy, L. Torczon), 1989

[11] Register Allocation & Spilling via Graph Coloring (G.J.Chaitin), 1982

[12] Register Allocation via Coloring (G.J.Chaitin, M.A. Auslander, A.K. chandra, J. Cocke), 1980

[13] The Priority-Based Coloring Approach to Register Allocation (F.C. Chow & J.L. Hennessy), 1990

[14] Optimistic Register Coalescing (J. Park & S-M. Moon), 1998

[15] Optimal and Near-optimal Global Register Allocation Using 0-1 Integer Programming (D.W. Goodwin & K.D. Wilken), 1996

[16] A Faster Optimal Register Allocator (C. Fu & K. Wilken), 2002

[17] Global Register Allocation at Link Time (D.W. Wall), 1986

[18] Register Allocation Across Proceedure and Module Boundaries (V. Santhanam & D. Odnert), 1990

[19] Instruction scheduling for the IBM RISC System/6000 processor (H.S.Warren), 1990

[20] Efficient Instruction Scheduling for a Pipelined Architecture (P.B. Gibbons, S.S. Muchnick), 1986

[21] Code Scheduling and Register Allocation in Large Basic Blocks (J.R. Goodman & W-C. Hsu), 1988

[22] Integrating Register Allocation and Instruction Scheduling for RISCs (D.G. Bradlee, S.J.Eggers, R.R. Henry), 1991

[23] Phase Ordering of Register Allocation and Instruction Scheduling (S.M. Freudenberger & J.C. Ruttenberg), 1991

[24] Software Pipelining: An Effective Scheduling Technique for VLIW Machines (M. Lam), 1988

[25] A Realistic Resource-Constrained Software Pipelining Algorithm (A. Aiken, A. Nicolau), 1990

[26] Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops (B.R Rau), 1994

[27] Compiler optimizations for processors with SIMD instructions (I. Pryanishnikov, A. Krall, N. Horspool), 2006