Contents

List of Figures

Chapter 1

Introduction

1.1 Background

The compiler mainly consists of two phases – the frontend and the backend. The lexical analysis and the parsing of the source code is done by the frontend of the compiler. Lexical analysis phase identifies the tokens and builds the symbol table. The 'parser' conforms the source code with the grammar of the language. It also performs the semantic analysis. Generally, the Intermediate Representation (IR) is generated by the parser of the frontend.

The backend mainly consists of the 'code generator'. It takes the intermediate representation as input and maps it into the target language. The target language is the sequence of machine instructions that perform the same task as the input. Reordering of instructions or instruction scheduling [1] [3] [11] [7] and judicious assignment of registers to hold variable values or register allocation [1] [3] [11] [5] [4] are the important phases of code generation. Both instruction scheduling and register allocation tries to reduce the space-time complexity. Backend mainly consists of the 'instruction scheduler' and the

'register allocator'. It also consists of optimizations that tries to reduce the compile time and execution time. The intermediate representation is handled by the various passes of the backend which finally converts it into the 'object' code. Our project concerns with the optimizations in instruction scheduling to reduce execution time.

Figure 1.1 shows a very broad overview of compilation stages. Our implementation is at 'Basic-block and scheduling 1' of block D.

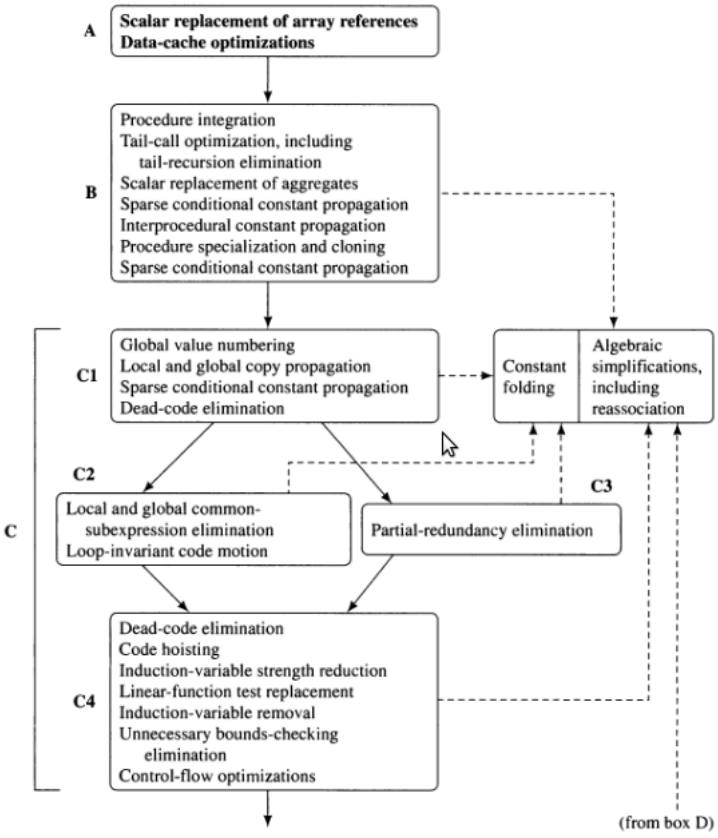

Figure 1.1: Compiler stages 1

1.2 Conventional Scheduling

A traditional scheduler takes execution latencies of all the instructions in the data dependency graph into account because the main objective is to reduce the length or execution time of the schedule. The list scheduler topologically orders the instructions. Using the

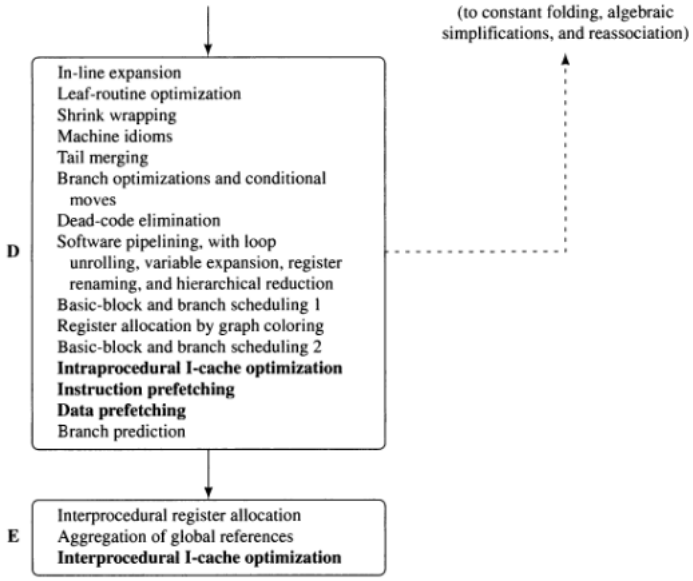

Figure 1.2: Compiler stages 2

order, the instruction is scheduled whose dependencies are satisfied. In case of out-of-order processors, scheduling can be done statically at compile time or dynamically by register renaming. Dynamic scheduling can uncover parallelism obscured by false dependencies. The next section gives a clear overview of scheduling.

1.3 Register reduction sequencing

Decreasing the register pressure decreases spill code [5] and ultimately decreases the execution time as cited in the next section. In this project, we try to *sequence* the instructions such that they use minimum number of registers and thus decreasing execution time. Our improvement over the conventional scheduling may be stated in the following definition.

Definition 1. *:* **Problem Statement** *: Given a data dependence graph G, derive an instruction sequence S for G that is optimal in the sense that its register requirement is minimum.*

1.4 Applications

- The Minimum Register Instruction Sequence can be used as an alternative to traditional scheduling in the backend of the compiler. Thus the MRIS can be implemented in all commercial compilers.

- The MRIS uses less registers during execution reducing the spill code when compared to traditional scheduling. Reduced spilling decreases memory transfers. Thus, an application compiled using the MRIS is likely to consume less power.

Chapter 2

Scheduling concepts and sequencing

Generally, the instructions are re-ordered by analysing the data dependency graph (DDG) to reduce execution time. But, the number of load and store instructions increases the execution time due to increase in 'spill' code. The number of registers are limited in any architecture processor. When the required number of registers are not present to accommodate all the live variables, then some variables are represented in memory and are known as spilled variables. Register pressure is the maximum usage of registers of the processor at any point in the execution sequence of the program. Spilling has more probability with increase in register pressure. Thus if register pressure is reduced, then spilling reduces which ultimately reduces the execution time.

On a processor with instruction-level parallelism, the execution time highly depends on potential parallelism [12] in the program. This program might execute faster if we can *extract* parallelism by reordering instructions from original sequence. Most of the modern processors are out-of-order multiple instruction issue. The processors whose parallelism is managed statically by software are known as Very-Long-Instruction-Word processors and those managed dynamically by hardware are superscalar processors. False dependencies

are handled in out-of-order processors by register renaming [12]. It uses the reservation stations to schedule the instructions whose dependencies are satisfied. In this project we are interested in generating an instruction sequence S, statically at compile time, that uses minimum registers and therefore defined as Minimum Register Instruction Sequence (MRIS). In out-of-order processors, spilling decreases parallelism as each load and store reduces the number of dependants and it is a non-trivial operation. Thus reducing spill codes is important:

- from a performance point of view in architectures that either have a small cache or a large cache miss penalty;
- from a memory bandwidth usage viewpoint;
- from an instruction-level parallelism viewpoint as the elimination of some of the spill instructions frees instruction slots to issue other useful instructions;
- from a power dissipation viewpoint, as load and store instructions contribute to a significant portion of the power consumed;
- in multi-threaded architectures, minimizing the number of registers in a thread reduces the cost of a thread context switch.

Scheduling is a form of program optimization that applies to the machine code produced by code generator. The three constraints for the scheduling are:

- *Control-dependence constraints*: All the operations executed in the original program must be executed in the optimized one.
- *Data-dependence constraints*: The operations in the optimized program must produce the same results as the corresponding ones in the original program.

- *Resource constraints*: The schedule must not oversubscribe the resources on the machine.

Scheduling changes the order of instructions and improves execution time. But, the state of the memory at any one point may not match any of the memory states in a sequential execution. This causes debugging problems when the program's execution is interrupted.

Data Dependence: If the execution order of two or more instructions are changed, whose variables are distinct or the instructions read the same variable, then the final result remains the same. But if an instruction writes to a variable read or written by another will result in different values after their order of execution is changed. The relative execution of these instructions must be preserved and such pairs of instructions are *data dependant* [1]. The data dependency may be true, anti or output dependant. True dependence exists if a write is followed by a read of the same location, the read depends on the value written. If a read is followed by a write to the same location, then anti-dependence exists. The write does not depend on read but write happens before read, then the read operation picks up a wrong value. Output dependence exists when a write is followed by another write. The memory location will have the value written by the last instruction instead of the exact instruction writing the value. Using different locations to store different values eliminates the dependencies. Figure 2.1 shows the data dependencies between the nodes for the code sequence:

c = 1;

a = c;

An edge, known as the dependency edge, from node A to node B indicates that node B is dependant on node A. The graph is known as a data dependency graph, G = (N,

E), having a set of N nodes representing the machine instructions' operations and the set of edges E representing the data-dependence constraints among the operations of the instructions. The dashed line indicates the control dependency among the instructions. FrameIndex is a slot used to store the variables. The graph is constructed for each basic block take function unit resource constraints on each operation. In case of traditional scheduling, an edge may be labeled by the latency indicating that the dependant node is to be scheduled after the latency period. The traditional list scheduler topologically orders the nodes of the data-dependency graph. Following the order the instructions are scheduled whose dependencies are satisfied.

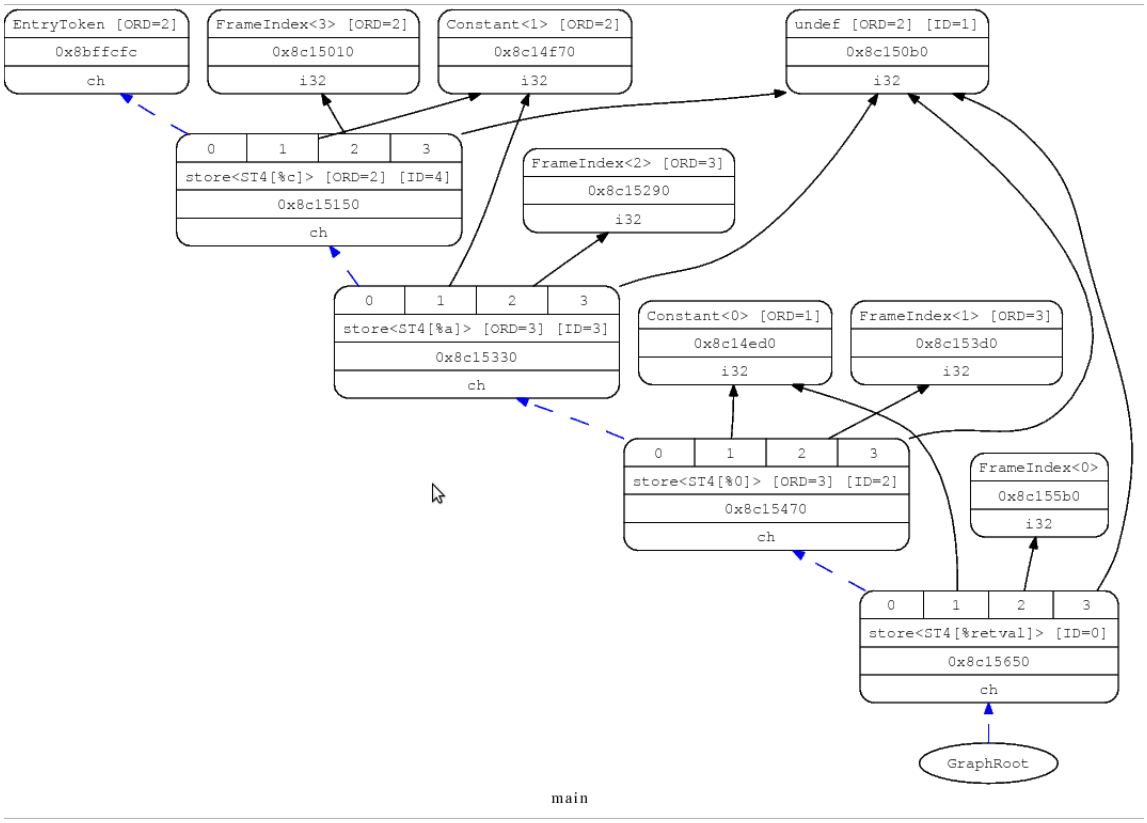

Figure 2.1: Data-Dependency Graph (DDG)

The list scheduling algorithm visits the nodes in the chosen prioritized topological order. The instructions are placed in the schedule as early as possible and thus the nodes may get scheduled in approximately the topological order. That is, the list scheduling

algorithm computes the earliest time slot in which each node can be executed, according to its data-dependence constraints with the previously scheduled nodes. When all the dependencies are satisfied for a node, it is scheduled. The list scheduling algorithm is shown. It takes the machine resources, R = [r_1, r_2, ...], where r_i is number of units available of the ith kind of resource and the data-dependency graph, G = (N, E). Each operation n in N is labeled with its resource-reservation table RT_n; each edge e = $n_1 \rightarrow n_2$ in E is labeled with d_e indicating that n_2 must execute no earlier than d_e clocks after n_1. The output is a schedule S that maps the operations in N into time slots in which the operations can be initiated satisfying all the data and resources constraints.

Algorithm 1 List_Scheduling(R, G)

```
1:  RT = an empty reservation table;
2:  for each n in N in prioritized topological order do
3:     s = max_{e=p→n in E} (S(p) + d_e);
4:     /* Find the earliest time this instruction could begin, given when its predecessors
       started. */
5:     while there exists i such that RT[s + i] + RT_n[i] > R do
6:        s = s + 1;
7:        /* Delay the instruction further until the needed resources are available */
8:     end while
9:     S(n) = s;
10:    for all i do
11:       RT[s + i] = RT[s + i] + RT[i];
12:    end for
13: end for
```

Sequencing differs by a lot from scheduling. Instruction sequence refers to the arrangement of instructions considering only to reduce the number of registers used or a particular heuristic. The MRIS approach tries to do the same. The input to MRIS problem is a partially ordered small sized data-dependency graph. The partial ordering enables the generation of an instruction sequence that requires less registers. But an instruction scheduler takes into account the latency factors and the availability of function unit resources to arrange the instructions. It also considers true, anti and output

dependency constraints whereas the MRIS problem considers only the true dependency constraints. An optimal code generator tries to reduce the schedule length for a fixed number of registers, while the MRIS problem minimizes the number of registers used.

Placement of the sequencer: If registers are allocated before scheduling, the resulting code tends to have many storage dependences that limit code scheduling. On the other hand, if code is scheduled before register allocation, the schedule created may require so many registers that register spilling will negate the advantages of instruction-level parallelism. Thus the ordering between scheduling and register allocation determines the resultant code. Generally, the scheduler is run both before and after the register allocation. The minimum register instruction sequencer tries to reduce the number of registers. Thus, it would be better to sequence the instructions first and pass them to the register allocator because it is more likely for the register allocator to allocate minimum registers for the sequence. Finally, the post-register allocation is performed.

The proposed MRIS method tries to build instruction lineages which are chain of instructions that use the same register. The lineages are fused, that is, two or more lineages which can use the same register are combined. This reduces parallelism and it is a trade off with usage of less registers to improve execution time. When two or more lineages need to run in parallel, they are represented by an interference edge in the lineage interference graph. This graph is colored by any of the graph coloring algorithms and a virtual register is allocated to each instruction of the entire lineage. Logically this method decreases the number of loads and stores and should improve the execution time.

Chapter 3

MRIS- The approach

For an Out-Of-Order issue processor which has many execution units, the ability to execute more than one instructions at the same time increases the yield for the program. But the problem of finding those instructions which can execute parallelly is difficult. Instructions have data dependencies on other instructions. That means, an Instruction makes use of the result produced by some other Instruction. An Instruction cannot start execution until all its Data Dependencies are satisfied. To make use of the ability of the OOO processor [12], Instructions which have all there Data Dependencies satisfied at that particular point need to be executed parallelly. The Scheduling [11] [7] phase re-orders the Instructions such that number of cycles the program has to wait for dependencies to satisfy is reduced thus reducing the execution time. It takes in account the Latency of each of the Instruction before finding a schedule. The variables that are represented in each instructions are to given register to store their values. Since the target processor has limited number of registers, the compiler will encompass a separate phase of Register allocation where each of the variables described in the aforementioned instructions will be mapped to a physical register. If two variables are needed at a same point, those values

need to represented in two different registers. If the program has many such variables, the need for more registers also increases. But since the number of registers are limited in any architecture processor, some variable of the program will have to be represented in memory rather than registers. This is called spilling [5]. Spilling leads to the introduction of Loads and Stores for the variables which are spilled. This make the program to have more Storage dependencies (Anti-dependencies). For an Out-of-order(OOO) processor this means that is takes longer time resolving the dependencies because Load and store instructions takes longer time to execute. This hits hard on the performance of OOO processors.

The Minimum Register Instruction Sequencing (MRIS) [8] tries to reduce the Register pressure of the program by rearranging the instructions. The solution tries to find a set of instructions that can definitely share a register. This set of Instructions is defined as a *Lineage*. A single register is allocated to each of the lineages and all the instructions in a lineage will use the same register to produce their result. The lineages are constructed to have as many instructions in them as possible. Thus, allocating a single register to many instructions reduces the Register Pressure on the program.

3.1 Motivating Example

The live range of a variable starts at the point of definition of that variable and ends at the last use. If two variables are simultaneously live at any point in the program, then those two variable need separate registers to store there values. These live ranges interfere with one another. The interfering live variables increases the register pressure since each of the live variables have to be represented by a separate register. The Non-interfering live variables can use a same register, thereby decreasing the register pressure

of the program.

The Data Dependency Graph (DDG) represents the dependencies of instructions on one another. The nodes represents the Instructions while an edge tells about the data flow from one instruction to another. DDG can be called as the Data flow of the program. Any sequence of instructions that will be generated has to conform to the DDG. An example DDG is shown in the figure 3.1.

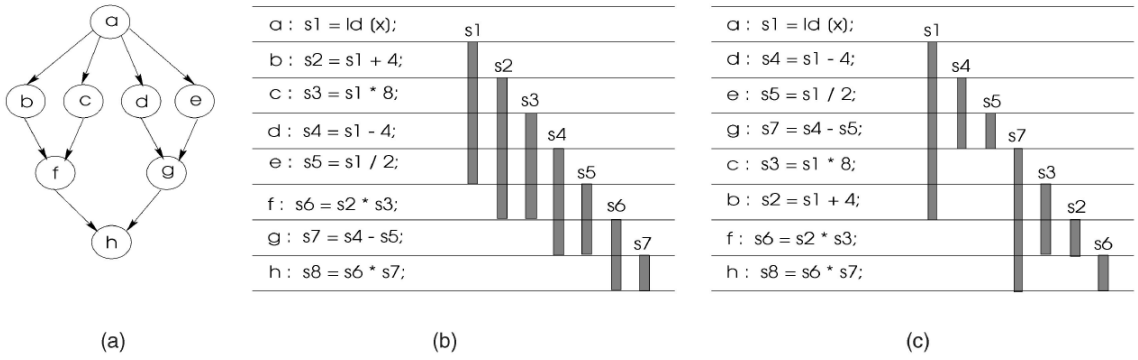

Figure 3.1: An example. (a)DDG. (b) Instruction Sequence 1. (c) Instruction Sequence 2

The figure 3.1 gives a Data dependency graph and two of the possible instruction sequence. Both of the Instruction sequences conform to the given DDG. Also the live ranges of each of the variables produced by the instruction is given. The number of overlapping live ranges in the instruction sequence 1 is 4 and its 3 for the instruction sequence 2. Comparing both the instruction sequence, the sequence 2 has minimum register pressure than that of sequence 1. In the DDG, the node b passes the value that it produced to node f and since there is no other nodes that uses the value of b, we can definitely say that in any instruction sequence, the register that is assigned to b can be reused by f. Similarly the nodes e and g can share the same register. Now, can f and g share the same register? No. Because both the instructions produce values that is used

by the node h. Hence nodes f and g interfere in all the possible instruction sequence and need separate registers for each one of them. Now considering the nodes c and d, although both are interfering in the instruction sequence 1, the sequence 2 places c and d such that they are not live simultaneously. This makes the nodes c and d share a same register. The DDG tells about the nodes that definitely overlap and those that may not overlap for all the instruction sequence. In order a generate a minimal register pressure instruction sequence, the nodes that can definitely share a register has to be identified.

The MRIS approach is bounded by the concept of *Lineages*. Lineage is a set of Instructions that can definitely share a register. The instructions for a Lineage are selected based on the DDG. A node will pass on its value to its descendant. If there are multiple descendants then one of them is chosen to be the heir. Hence the register that is used by the parent to store its value will be passed on to its heir. Such sequence of instructions I_1, I_2, I_3... I_n where I_2 is a heir of I_1, I_3 as heir of I_2 and so on, will form an Instruction Lineage. All the instructions in a lineage will share a same register, as the register is passed onto from one instruction to another.

Since the Instruction sequence that will be generated must conform to the DDG, the instructions in a lineage has a implicit order of execution. The heir of a node cannot execute until its parent finishes its execution. If a parent has many descendants, the heir of the parent node must not start execution until all other descendants of the parent node finishes use of the value produced by the parent. Hence the order of execution should have parent node and all the descendants of the parent node other than the heir and then only the heir of the parent node starts execution. This sequencing constraint is represented by having an *Sequencing edges* from other descendants of the parent node to its heir. This results in the updating of the DDG called the *Augmented DDG*. The

original DDG and its corresponding Augmented DDG is represented in the Figure 3.2.

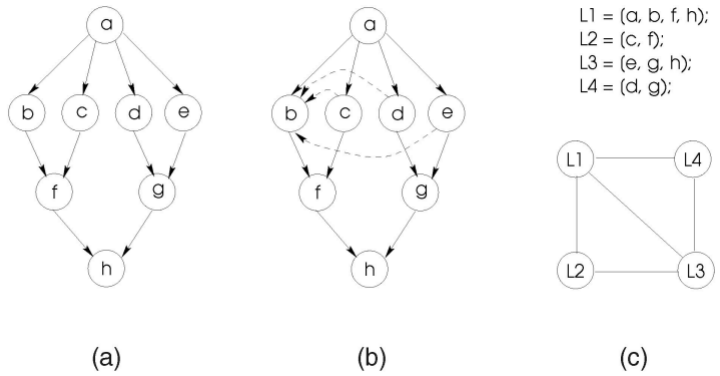

Figure 3.2: Data dependence graph for the motivating example. (a) Original DDG. (b) Augmented DDG. (c) Lineage interference graph.

For the given DDG, the lineages are

$L_1 = \{a, b, f, h\}$

$L_2 = \{c, f\}$

$L_3 = \{e, g, h\}$

$L_4 = \{d, g\}$

The lineage start from the point where the first instruction produces the value in a register and ends at the last use of that register for that lineage. Hence the lineages are represented as the def-last_use relation of instructions wherein the last instruction uses the value in the register. The last instruction may be part of another lineage. Thus we represent this relation starting with a closed interval and ending the lineage by a open interval. This representation for the lineages generated above will be,

$L_1 = [a, b, f, h)$

$L_2 = [c, f)$

$L_3 = [e, g, h)$

$L_4 = [d, g)$

After having defined the set the instructions that can definitely share a register as lineages, the process of Register Allocation of instruction in the program is now to allocate registers for the lineages. The live range of a lineage is the concatenation of the live ranges of the instructions that the lineage contain. To determine the interference between two lineages, we have to see whether the live range of the two lineages overlap in all legal sequence of instructions. If the live range of two lineages overlap in all the instruction sequences then they cannot share a register. This is represented by having an edge between the two lineages in the Lineage Interference Graph(LIG). If two lineages does not overlap at least in some legal instruction sequences, then the two lineages can share a same register given that there is some ordering of instructions between the lineages. The Lineage Interference Graph is constructed by the data of the lineages interfering with each other. The traditional coloring approach [5] [4] is used to color the LIR. The minimum number of color required to color the LIR give the minimum number of register needed for executing the program. This number is called the Heuristic Register Bound(HRB).

The sequencing of the instructions is done using the coloring of lineages and the DDG of the program so that the sequence generated has minimum register pressure.

3.2 Lineage Formation

A node may have many immediate descendants in a DDG, but there can only be one node among them that can be selected as the heir. An heir inherits the register form its parent node and hence becomes a part of the lineage. This imposes a restriction that the heir must be the last one among the descendants of the node to be executed as it redefines the register shared by the parent and the value is lost after redefinition.

The lineage formation algorithm attempt to form as long lineages as possible. The heir in any lineage is the last one to be executed, hence the live ranges of the instructions do not interfere with each other and all the instructions in the lineage can share the same register. In order to ensure that the heir is the last one to be executed, the algorithm introduces *sequencing edges*. A sequencing edge from node v_i to v_j imposes the constraint that v_i must be listed before v_j. This implies that all the nodes that reach v_i must be listed before any node from v_j is listed. The algorithm makes sure that the introduction of sequencing edges does not make the graph cyclic, which would make it impossible to obtain a sequence for the instructions in the DDG, by using the *height* property to select the heir. The *height* of a node is computed as per the equation

$$ht(u) = \begin{cases} 1 & \text{if } u \text{ has no descendants} \\ 1 + max_{v \epsilon D(u)}(ht(v)) & \text{otherwise,} \end{cases}$$

During the lineage formation, if a node v_i has multiple descendants, then the algorithm chooses the node which is having the lowest height among the descendants to be the heir of v_i. If two or more nodes have the same height then the algorithm arbitrarily breaks the tie and it recomputes the height of each node after the introduction of sequencing edges to avoid the formation of cycles. Lineages are formed by the arcs between the nodes and the chosen heirs. Each arc that is a part of a lineage is the def-last-use arc in DDG.

The Lineage Formation algorithm is essentially a greedy depth first graph traversal algorithm which identifies the heir of each node by the height property. The algorithm introduces sequencing edges after the heir of a node is selected if the node has multiple descendants and the heights of all the nodes in the DDG are recomputed. The algorithm distinguishes between the sequencing edges and the flow edges. If there is a sequencing edge between node v_i to node v_j, then v_j is not considered as the descendant of v_i for the purpose of lineage formation. Only those arcs that represent data dependences are considered for lineage formation.

Algorithm 2 Lineage_Formation(V, E)

```
1: mark all nodes in DDG as not in any lineage
2: compute the height of every node in DDG
3: while there is a node not in any lineage do
4:    recompute height ← false
5:    v_i ← highest node not in lineage
6:    start a new lineage containing v_i
7:    mark v_i as in a lineage
8:    while v_i has a descendant do
9:       v_j ← lowest descendant of v_i
10:      if v_i has multiple descendants then
11:         recompute height ← true
12:         for each descendant v_k ≠ v_j of v_i do
13:            add sequencing edge from v_k to v_j
14:         end for
15:      end if
16:      add v_j to lineage
17:      if v_j is already marked as in a lineage then
18:         end lineage with v_j as the last node
19:         break ;
20:      end if
21:      mark v_j as in lineage
22:      v_i ← v_j
23:   end while
24:   if recompute height = true then
25:      recompute the height of every node in the DDG
26:   end if
27: end while
```

3.3 Lineage Interference Graph

By determining the live ranges of two lineages and verifying that the two lineages do not overlap, it is possible to make the two lineages share the same register. The live range of a register is given by the definition-

Definition 2. *If v_1, the first instruction of a lineage $L = [v_1, v_2, ..., v_n)$ is in position t_i, and the last instruction v_n of L is in the position t_j in an instruction sequence , then the* ***live range*** *of the lineage L starts at t_{i+1} and ends at t_j.*

The live range of a lineage is always contiguous, irrespective of interleaving of instructions from multiple lineages. Thus the live range is active from the listing of the first instruction in the lineage until the listing of the last instruction.

In order to determine whether the live ranges of two lineages must necessarily overlap, a condition is defined based on the existence of paths between the lineages. Two sets of nodes are defined. S is the set of nodes that start lineages and E is the set of nodes that end lineages. Then a *reach* relation R is defined as

Definition 3. *The reach relation R: $S \rightarrow E$ maps S to E. For all $v_a \in S$ and $v_b \in E$, node v_a reaches node v_b , $R(v_a, v_b) = 1$, if there is a path in the augmented DDG from v_a to v_b. Otherwise, $R(v_a, v_b) = 0$.*

The reach relation is used to determine whether the live ranges of two lineages must necessarily overlap in all legal instruction sequences for the augmented DDG.

Definition 4. *Two lineages,* $L_u = [u_1, u_2, ..., u_m)$ *and* $L_v = [v_1, v_2, ..., v_n)$, ***definitely overlap*** *if they overlap in all possible instruction sequences.*

The Lineage Interference Graph (LIG) is constructed as an undirected graph whose nodes represent lineages. The graph contains a arc between two nodes if and only if the live ranges of lineages represented by the nodes *definitely overlap.* Essentially there is a edge in LIG between two lineages $L_u = [u_1, u_2, ..., u_m)$ and $L_v = [v_1, v_2, ..., v_n)$ if and only if $R(u_1,v_n) = 1$ and $R(v_1,u_m) = 1$.

3.4 Lineage Fusion

Let there be two lineages $L_u = [u_1, u_2, ..., u_m)$ and $L_v = [v_1, v_2, ..., v_n)$. The two lineages do not necessarily overlap if $R(u_1,v_n) = 1$ and $R(v_1,u_m) = 0$, i.e u_1 reaches v_n , but v_1 does not reach u_m. Therefore there is always a possibility to find a legal sequence where the two live ranges do not overlap, and both lineages can share the same register. Conversely if the nodes of L_v are listed in the instruction sequence before all the nodes in L_u are listed; the two lineages interfere and cannot share the same register.

This can be prevented by introducing new sequencing constraint in DDG. The constraint forces all the nodes of L_u to be listed before any node of L_v is listed. This is called *lineage fusion.* As the name suggests, the lineage fusion operation fuses two lineages which may not necessarily interfere into a single lineage and treat the two lineages L_u and L_v as if they are a single lineage. Lineage fusion also reduces the number of nodes in the lineage interference graph, there by reducing the cost of coloring the graph. Formally

Lineage Fusion is defined as

Definition 5. *Two lineages L_u = [u_1, u_2,....,u_m) and L_v = [v_1, v_2,...., v_n) can be fused into a single lineage if:*

- u_1 *reaches* v_n, *i.e.,* $R(u_1, v_n) = 1$;
- v_1 *does not reach* u_m, *i.e.,* $R(v_1, u_m) = 0$.

When the lineages L_u and L_v are fused together, a sequencing edge from u_m to v_1 is inserted in the DDG. The lineages L_u and L_v are removed and a new lineage $L_w = [u_1, u_2, ..., u_m) \cup [v_1, v_2, ..., v_n)$ is inserted in the lineage set. The last node of the first lineage, u_m , does not necessarily use the same registers as the other nodes in the new L_w lineage. Thus, it is important to represent the lineage resulting from the fusion as a union of semi-open sequences.

Fusing two lineages causes the corresponding nodes in the interference graph to be combined into a single node, say w. Every edge that was incident on the either of the nodes in the lineages before fusing is now incident on w. As a consequence, the interference graph is updated after each such fusion operation. It is also legal to fuse lineages which are completely independent, i.e. when $R(u_1, v_n) = 0$ and $R(v_1, u_m) = 0$. However such fusions would impose unnecessary constraints in the sequencing of L_u and L_v. The coloring algorithm indicates such situations thus providing freedom for sequencing algorithm.

Lineage fusion is applied after the lineage formation and before the coloring of the lineage interference graph. The reason being that the interference graph to be colored would have less number of nodes and would require less number of colors.

To find the lineages that can be fused, the set of lineages is searched for $L_u = [u_1, u_2,....,u_m)$ and $L_v = [v_1, v_2,...., v_n)$ such that $R(u_1, v_n) = 1$ and $R(v_1, u_m) = 0$. In accordance to the fusion condition a new lineage is formed as $L_w = L_u \cup L_v$, which requires that a new sequencing edge must be added from u_m to v_1. The addition of this new edge necessitates the updating of reach relation. The algorithm updates the reach relation such that all the nodes that could reach u_m prior to the fusion can also reach v_n after the fusion. Now v_1 is no longer the start node of the fused lineage L_w, and hence the row corresponding to node v_1 is discarded from the reach relation. Similarly, if u_m does not terminate any other lineage, its column can also be discarded from the reach relation. Further v_1 is removed from the set start nodes S, and u_m is removed from the set of end nodes E, if u_m does not end any other lineages other than L_w.

If there are multiple candidates for lineage fusion, the algorithm arbitrarily selects two lineages and performs the fusion operation. Each fusion results in the updating of sets S, E and reach relation R. The algorithm continues to search for a suitable pair to fuse until it finds none. It is also possible that a fused lineage $L_u \cup L_v$ may again be fused with some other lineage, say L_x (if they form a legal pair to fuse), thus resulting in a compound lineage $L_u \cup L_v \cup L_x$. The order in which the lineages are fused will determine the size of the interference graph and hence the heuristic register bound values. Different order in the fusion of lineages produces different heuristic register bound values.

3.5 Coloring the Interference Graph

The Lineage Interference Graph can be colored using a heuristic graph coloring algorithm [5] [4]. The number of colors required to color the interference graph is referred to as the Heuristic Register Bound (HRB). Due to the heuristics involved in coloring and due to the sequencing order of descendants nodes in DDG, the HRB computed is near-optimal solution and graph coloring generally is NP-complete [6].

3.6 Instruction Sequence Generation

Once the coloring of the Lineage Interference Graph is completed, each lineage is associated with a register and hence the nodes in the lineage. It is assumed that those registers that are live-in and live-out in the DDG are assigned by the Global Register Allocator. This implementation accounts for the live-in and live-out of registers by the dummy *source* and *sink* nodes.

The sequencing algorithm used here is a modified list scheduling algorithm. It uses the information obtained from the coloring of the interference graph, and *lists* the nodes from a ready list based on the height priority and availability of registers assigned to them.

The sequencing algorithm takes inputs $G^{|}$, the augmented DDG with sequencing edges, L, list of lineages obtained from the lineage formation algorithm and after the

Algorithm 3 Sequencing($G^{|}$, L, A)

```
1: ReadyList ← {(v_i, R_j) such that v_i has no predecessor }
2: RegAvailable ← {R_1, R_2,....,R_N}
3: while ReadyList ≠ φ do
4:    for each node v_i in the ReadyList in decreasing height order do
5:       if (v_i ∉ S) or (((v_i, R_j) ∈ A) and (R_j ∈ RegAvailable)) then
6:          // either v_i is not n start node of lineage or
7:          // the register assigned to v_i in A is available
8:          Remove R_j from RegAvailable
9:          Remove (v_i, R_j) from ReadyLIst
10:         List (v_i, R_j)
11:         Add to the ready list all successors of v − i that have a;; its predecessor listed
12:         if (v_i ∈ E) and (v_i, R_j) ∈ A then
13:            // node v − i ends a lineage which
14:            // is assigned register R_j
15:            Return R_j to RegAvailable
16:         end if
17:      end if
18:   end for
19: end while
```

lineage fusion is applied, and A, the register assignment for the nodes from the coloring of the lineage interference graph.

This algorithm may result in a deadlock due to two reasons. First, the order of listing of nodes belonging to two different lineages having the same color assigned may result in deadlock. Such type of deadlocks are referred to as *avoidable deadlocks*. These can be avoided by applying the lineage fusion. The other kind of deadlocks, referred to as *unavoidable deadlocks*, arise due to the underestimation of HRB. This could happen because the condition used to test if two live ranges definitely overlap is sufficient but not necessary.

When the sequencing algorithm is applied to the motivating example, first the node a is listed. Then among the descendants of a, node b is already chosen to be the heir

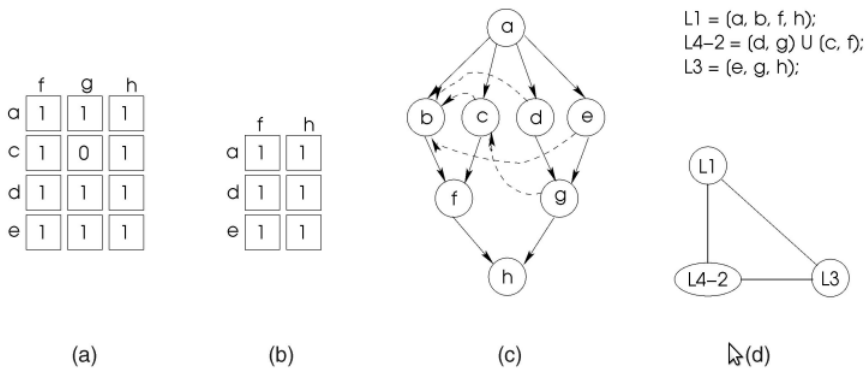

Figure 3.3: Reach relation. DDG and LIG after lineage fusion for motivating example. (a) Reachability before fusion. (b) Reachability after fusion. (c) DDG after fusion. (d) LIG after lineage fusion.

of node a as it is the part of the same lineage. If lineage fusion is not used, listing of a causes nodes *c, d, e* to be added to Ready List. The next node that can be listed is node e as it is under a separate lineage, L_3, and register R_3 is available. Now the sequencing algorithm is left with only one register R_2 and either of node d or c can be listed next. The algorithm breaks the tie arbitrarily and chooses one among the remaining two nodes. If it chooses node c to be listed next, an unnecessary cycle of dependencies is created. Node d cannot be listed until node f is listed, which would then release register R_2. But to list node *f*, node b has to be listed first. Since node b must be the last use of of R_1, it cannot be listed before node *d*. This creates cyclic dependencies and the sequencing algorithm deadlocks. However, this type of deadlock is avoidable deadlock and can be prevented by the process of lineage fusion. When lineage fusion is applied to above case, lineages L_2 and L_4 are fused together and a sequencing edge from node g to node c is added to the graph, which ensures that node d is listed before node *c*. The sequencing algorithm then lists nodes *d, g, c, b, f* and h in order thereby using only three registers as predicted by the coloring algorithm.

Even with the application of lineage fusion, unavoidable deadlocks occur when the HRB computed by the register coloring algorithm is lower than the number of registers actually needed. In such cases there does not exist a legal sequence of instructions that uses HRB or lower number of registers. The simple approach to overcome this is to increase the HRB value computed by one. The algorithm then picks up one node in the Ready List and changes it register assignment to a new register (and for the remaining nodes as well). This strategy overcomes the deadlock by gradually increasing the HRB value and trying to obtain a legal sequence of instructions with as few extra register as possible.

Chapter 4

Implementation

The sequencing by MRIS approach is implemented in the Low-Level Virtual Machine (LLVM) Compiler Infrastructure [16]. LLVM is purely an object oriented infrastructure implemented in C++. LLVM-GCC is a LLVM C front end which converts the source code to object form. The backend of LLVM converts this object code to executable. This backend has the capability to perform various optimizations. The best thing about using LLVM is that every stage of the compiler is separated distinctly with different passes, handled by the pass-manager, for each stage. A pass can be disabled deliberately and the effect of compilation without that pass or that particular compilation stage could be analysed. Analysis can be done any time using the class *AnalysisUsage*. A pass can be replaced with a new one to analyse the effect. In-all, LLVM provides a very sophisticated environment for compilation. In the following section, we discuss briefly the data structures related to scheduling in LLVM.

LLVM maintains the DAG in a data structure known as the SelectionDAG. This class is used to represent a portion of an LLVM function in a low-level Data Dependence DAG representation suitable for instruction selection. This DAG is constructed as the

first step of instruction selection in order to allow implementation of machine specific optimizations and code simplifications. The representation used by the SelectionDAG is a target-independent representation and which is simple, powerful, and is a graph form instead of a linear form. Each of the nodes in the SelectionDAG is represented by the SDNode class. This class maintains the NodeType, NodeId, OperandList – the values used by this node, ValueList – the values that this node defines, UseList – the list of uses of this node. Additionally, the SelectionDAG provides a host representation where a large variety of very-low-level (but target-independent) optimizations may be performed; ones which require extensive information about the instructions efficiently supported by the target.SDNode may be visualised as:

```
class SDNode {
    private:
    int NodeType, NodeId, Height;
    List OperandList, ValueList, UseList, Descendants;
    .
    .
    bool IsInLineage, IsScheduled, VisitedHeight;
    SDNode *SequencingEdge;
};
```

To implement MRIS, the following data members were added to SDNode: Height - contains the height as calculated by the height computation function; Descendants - These are the set of nodes that are descendants of this node. The Boolean values guide the presence of the node - IsInLineage indicates that node is already in the lineage, IsScheduled indicates that the node is already scheduled and VisitedHeight indicates that the

node has been visited while computing height.

The ScheduleDAG class contains the set of Scheduling Units, SUnits, which wrap around the SDNode. Sequence is a member of the ScheduleDAG which represents the schedule. Null SUnits in it represent no-ops. The SUnit class contains the SDNode and the corresponding Machine Instruction. It also contains the list of predecessor and successor dependencies. The isScheduled boolean value indicates if this scheduling unit has been scheduled. It also has the latency component, height and depth of the scheduling unit used in traditional scheduling. This may be represented as:

```
class SUnit {
    private:
    SDNode *Node;
    MachineInstr *Instr;
    DependencyList Preds, Succs;
    unsigned short Latency, Height, Depth;
    bool isScheduled;
    .
    .
};
```

The lineage is represented by the list of SUnit and virtual register pairs. The lineage also contain the start and end nodes of the lineage. The lineage class also contains the accessor functions like to add a node to lineage, set a register to the lineage. The lineage class is as:

```
class Lineage {
    private:
    List LineageNodes(pair¡SUnit*, Register¿);
    SUnit *Start, *End;
    .
    .
};
```

Finally, the entire MRIS is implemented just before list scheduling. The ScheduleDAGRRList class implements the register reduction scheduling. MRIS is also concerned with reducing the registers and thus finds the right place for implementing. Lineage formation, reach matrix, lineage fusion and the sequence generation are implemented.

4.1 Traversal of the DDG

The root of the DDG - GraphRoot is the exit point of the basic block. Entry to the DDG can be made from node EntryToken. The DDG is traversed by moving into the uses of the node - the children of a node. Only true dependency is required for MRIS and therefore, the control dependencies are ignored. In case of loads and stores, FrameIndex slot is used to load and store the values. The children or uses of load or store is got by moving into the FrameIndex and then accessing the uses of FrameIndex whose height is less than the load or store node. A TokenFactor node converts multiple input values into a single output value. If there is a sequencing edge added by the MRIS implementation, then DDG is traversed along it.

4.2 Compilation and usage

LLVM uses C/C++ libraries along with GCC/G++ and other tools like ocaml, gv/dot for its compilation. LLVM-GCC is a LLVM C front-end converts the source program into LLVM bitcode. LLC is a LLVM static compiler. LLI is capable of executing the programs directly from LLVM bitcode. LLVM has other tools like LLVM-AS which is the LLVM assembler; LLVM-DIS, the disassembler; LLVM-LINK, the linker; LLVM-PROF to print execution profile of LLVM program; LLVM-EXTRACT to extract a function from LLVM module.

4.3 List of Functions

These are the list of functions added to the existing LLVM infrastructure to implement Minimum Register Instruction Sequence. Function names and their description follow:

ComputeHeightOfNode(SDNode *N)	Compute the height of node N recursively as height (u) = 1, if 'u' has no descendants = 1 + max (descendants height) of 'u', otherwise.
SUnit *getHighestNodeNotInAnyLineage()	Returns the highest node not in any lineage.
SUnit *getLowestDescendantNode(SUnit *SU)	Returns the lowest descendant of the node.

void MRISLineageFormation (SmallVectorImpl<Lineage> &LineageList)	Forms lineages as required by MRIS
int ModifyLineages (SmallVectorImpl<Lineage> &LineageList)	Build the reach matrix, fuse lineages, build and color the lineage interference graph
bool Reach(SDNode *Va, SDNode *Vb)	This returns true if there exists a path from Va to Vb, also through the sequencing edges in the DDG, otherwise returns false.
int **ReachRelation(int **ReachMatrix, SmallVectorImpl<SUnit*> &StartList, SmallVectorImpl<SUnit*> &EndList)	Calculate the reach relation matrix
void FuseLineages(int UmEnd, int V1Start, SmallVectorImpl<Lineage> &LineageList)	Fusion of two lineages and updating the LineageList
int BuildAndColorLIG(int Row, int Column, int **ReachMatrix, SmallVectorImpl<Lineage> &LineageList)	Build the Lineage Interference Graph using the ReachMatrix and color it to know the minimum number of registers required (HRB). Return HRB.
void AddToReadyList(SDNode *N, std::list<SUnitReg> &ReadyList, SmallVectorImpl<Lineage> &LineageList)	Find SUnit corresponding to SDNode and add to ReadyList

void AddSuccessors(SUnit *Vi, std::list<SUnitReg> &ReadyList, SmallVectorImpl<Lineage> &LineageList)	Add nodes to the ReadyList all successors of Vi that have all its predecessors listed and in decreasing height order.
void LineageInstructionSequence(int HRB, SmallVectorImpl<Lineage> &LineageList)	Sequencing the nodes of the lineages using the registers allocated to each node during coloring of the Lineage Interference Graph

Chapter 5

Test cases

5.1 Example 1 - Simple Arithmetic Operations

Consider the program as shown in figure 5.1. The corresponding DDG extracted during debugging using GDB is shown in figure 5.2.

```
vikram@honneralu-desktop:~/llvm/test$ cat mulDiv.c
int main () {
  int a, b, c, d, e;
  c = a * b;
  e = c / d;
  return 0;
}
```

Figure 5.1: A simple example

The set of lineages formed are as shown in figure 5.3. The corresponding reach matrix is as shown in figure 5.4. After fusions, the reach matrix will be as shown in figure 5.5 and the lineages will be as shown in figure 5.6. Finally, the sequence generated using lineages will be as shown in figure 5.7. This program had an HRB of 2.

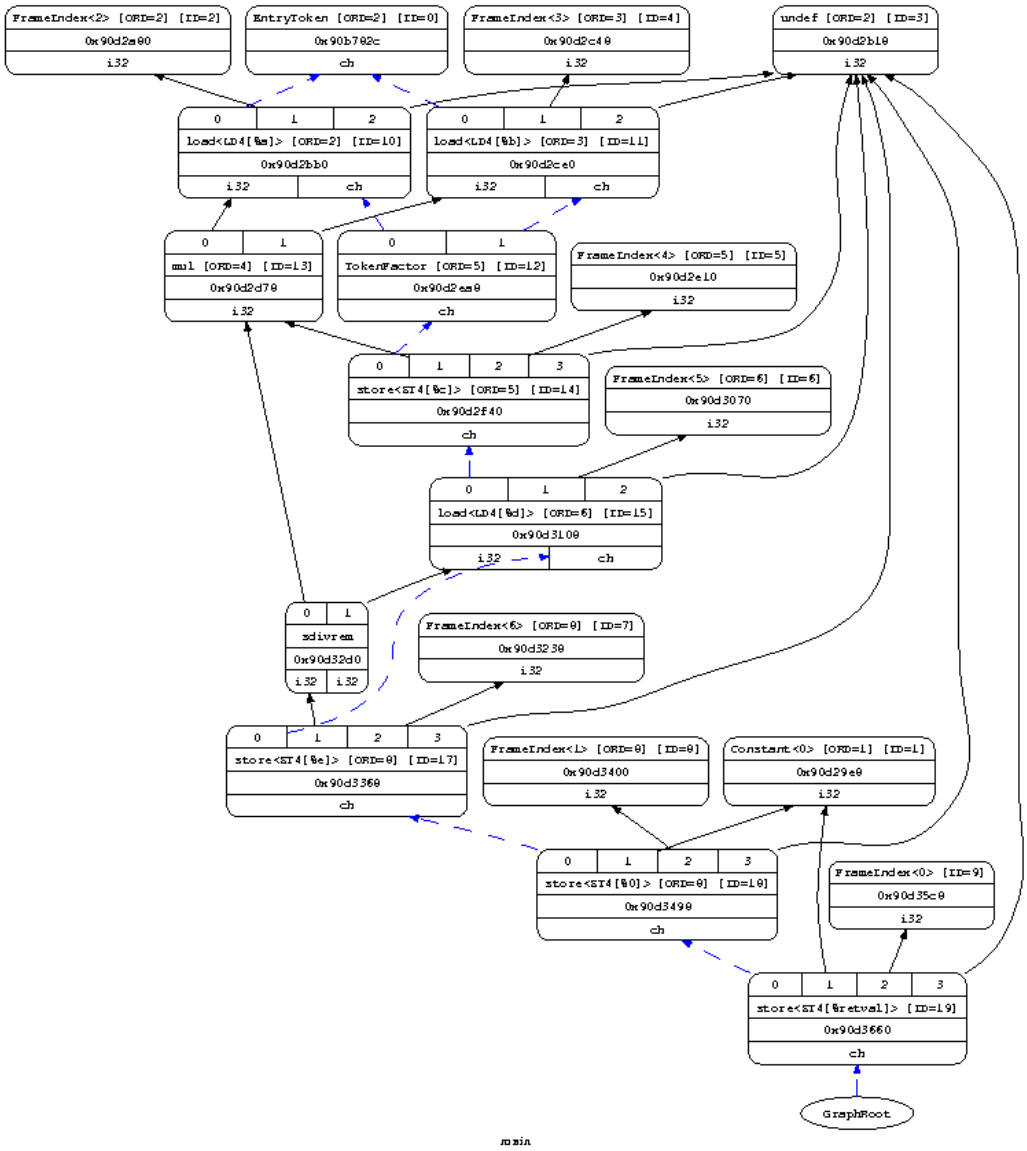

Figure 5.2: MRIS DAG

```
Lineage 0
Print Lineage: size: 4
0x9610d08: i32,ch = load 0x95f584c, 0x9610c60, 0x9610b10<LD4[%b]> [ID=6]
0x9610db0: i32 = mul 0x9610bb8, 0x9610d08 [ID=5]
0x9611398: i32,i32 = sdivrem 0x9610db0, 0x96111a0 [ID=4]
0x9611440: ch = store 0x96111a0:1, 0x9611398, 0x96112f0, 0x9610b10<ST4[%e]> [ID=3]

Lineage 1
Print Lineage: size: 2
0x9610bb8: i32,ch = load 0x95f584c, 0x9610a68, 0x9610b10<LD4[%a]> [ID=7]
0x9610db0: i32 = mul 0x9610bb8, 0x9610d08 [ID=5]

Lineage 2
Print Lineage: size: 2
0x96111a0: i32,ch = load 0x9610fa8, 0x96110f8, 0x9610b10<LD4[%d]> [ID=8]
0x9611398: i32,i32 = sdivrem 0x9610db0, 0x96111a0 [ID=4]

Lineage 3
Print Lineage: size: 1
0x9610fa8: ch = store 0x9610f00, 0x9610db0, 0x9610e58, 0x9610b10<ST4[%c]> [ID=9]
```

Figure 5.3: Lineages formed

```
1111
1111
1010
1011
```

Figure 5.4: Reach Matrix

```
111
111
111
```

Figure 5.5: Reach Matrix after fusions

```
Lineage 0
Print Lineage: size: 4
0x9d7ad08: i32,ch = load 0x9d5f84c, 0x9d7ac60, 0x9d7ab10<LD4[%b]> [ID=6]
0x9d7adb0: i32 = mul 0x9d7abb8, 0x9d7ad08 [ID=5]
0x9d7b398: i32,i32 = sdivrem 0x9d7adb0, 0x9d7b1a0 [ID=4]
0x9d7b440: ch = store 0x9d7b1a0:1, 0x9d7b398, 0x9d7b2f0, 0x9d7ab10<ST4[%e]> [ID=3]
2 2 2 2
Lineage 1
Print Lineage: size: 4
0x9d7abb8: i32,ch = load 0x9d5f84c, 0x9d7aa68, 0x9d7ab10<LD4[%a]> [ID=7]
0x9d7adb0: i32 = mul 0x9d7abb8, 0x9d7ad08 [ID=5]
0x9d7b1a0: i32,ch = load 0x9d7afa8, 0x9d7b0f8, 0x9d7ab10<LD4[%d]> [ID=8]
0x9d7b398: i32,i32 = sdivrem 0x9d7adb0, 0x9d7b1a0 [ID=4]
1 1 1 1
Lineage 2
Print Lineage: size: 1
0x9d7afa8: ch = store 0x9d7af00, 0x9d7adb0, 0x9d7ae58, 0x9d7ab10<ST4[%c]> [ID=9]
```

Figure 5.6: Lineages after fusions

```
Sequence:
0x9d7abb8: i32,ch = load 0x9d5f84c, 0x9d7aa68, 0x9d7ab10<LD4[%a]> [ID=7]
0x9d7ad08: i32,ch = load 0x9d5f84c, 0x9d7ac60, 0x9d7ab10<LD4[%b]> [ID=6]
0x9d7adb0: i32 = mul 0x9d7abb8, 0x9d7ad08 [ID=5]
0x9d7afa8: ch = store 0x9d7af00, 0x9d7adb0, 0x9d7ae58, 0x9d7ab10<ST4[%c]> [ID=9]
0x9d7b1a0: i32,ch = load 0x9d7afa8, 0x9d7b0f8, 0x9d7ab10<LD4[%d]> [ID=8]
0x9d7b398: i32,i32 = sdivrem 0x9d7adb0, 0x9d7b1a0 [ID=4]
0x9d7b440: ch = store 0x9d7b1a0:1, 0x9d7b398, 0x9d7b2f0, 0x9d7ab10<ST4[%e]> [ID=3]
```

Figure 5.7: Final Sequence

Consider another program code as shown in figure 5.8. The sequence generated for this program is shown in figure 5.9 and had an HRB of 3.

```
vikram@honneralu-desktop:~/llvm/test$ cat mris.c
int main () {
  int x, s1, s2, s3, s4, s5, s6, s7, s8;
        s1 = x;
        s2 = s1 + 4;
        s3 = s1 * 8;
        s4 = s1 - 4;
        s5 = s1 / 2;
        s6 = s2 * s3;
        s7 = s4 - s5;
        s8 = s6 * s7;
        return 0;
}
```

Figure 5.8: MRIS example code

```
Sequence:
0x9e051a8: i32,ch = load 0x9de9df4, 0x9e05058, 0x9e05100<LD4[%x]> [ID=23]
0x9e052f8: ch = store 0x9e051a8:1, 0x9e051a8, 0x9e05250, 0x9e05100<ST4[%s1]> [ID=24]
0x9e05cd0: i32,ch = load 0x9e05c28, 0x9e05250, 0x9e05100<LD4[%s1]> [ID=14]
0x9e05a30: i32,ch = load 0x9e05988, 0x9e05250, 0x9e05100<LD4[%s1]> [ID=17]
0x9e056e8: i32,ch = load 0x9e05640, 0x9e05250, 0x9e05100<LD4[%s1]> [ID=20]
0x9e054f0: i32 = add 0x9e051a8, 0x9e05448 [ID=22]
0x9e05e20: i32 = shl 0x9e056e8, 0x9e053a0 [ID=19]
0x9e05640: ch = store 0x9e052f8, 0x9e054f0, 0x9e05598, 0x9e05100<ST4[%s2]> [ID=21]
0x9e05988: ch = store 0x9e056e8:1, 0x9e05e20, 0x9e058e0, 0x9e05100<ST4[%s3]> [ID=18]
0x9e07068: i32,ch = load 0x9e06fc0, 0x9e05598, 0x9e05100<LD4[%s2]> [ID=25]
0x9e07110: i32,ch = load 0x9e06fc0, 0x9e058e0, 0x9e05100<LD4[%s3]> [ID=9]
0x9e071b8: i32 = mul 0x9e07068, 0x9e07110 [ID=8]
0x9e05d78: i32 = srl 0x9e05cd0, 0x9e05790 [ID=13]
0x9e07fc8: i32 = add 0x9e05cd0, 0x9e05d78 [ID=12]
0x9e07d88: i32 = add 0x9e05a30, 0x9e078f0 [ID=16]
0x9e08118: i32 = sra 0x9e07fc8, 0x9e05838 [ID=11]
0x9e05c28: ch = store 0x9e05a30:1, 0x9e07d88, 0x9e05b80, 0x9e05100<ST4[%s4]> [ID=15]
0x9e06fc0: ch = store 0x9e05cd0:1, 0x9e08118, 0x9e05ec8, 0x9e05100<ST4[%s5]> [ID=10]
0x9e07458: i32,ch = load 0x9e073b0, 0x9e05b80, 0x9e05100<LD4[%s4]> [ID=27]
0x9e07500: i32,ch = load 0x9e073b0, 0x9e05ec8, 0x9e05100<LD4[%s5]> [ID=6]
0x9e073b0: ch = store 0x9e07308, 0x9e071b8, 0x9e07260, 0x9e05100<ST4[%s6]> [ID=7]
0x9e075a8: i32 = sub 0x9e07458, 0x9e07500 [ID=5]
0x9e07848: i32,ch = load 0x9e077a0, 0x9e07260, 0x9e05100<LD4[%s6]> [ID=28]
0x9e077a0: ch = store 0x9e076f8, 0x9e075a8, 0x9e07650, 0x9e05100<ST4[%s7]> [ID=29]
0x9e07998: i32 = mul 0x9e07848, 0x9e075a8 [ID=4]
0x9e07b90: ch = store 0x9e07848:1, 0x9e07998, 0x9e07a40, 0x9e05100<ST4[%s8]> [ID=3]
```

Figure 5.9: Sequence for MRIS example code

5.2 Example 2 - if-else Operation

Consider another example involving an if-else construct. The code is shown in the figure 5.10. Figures 5.11 - 5.20 show the data-dependency graphs generated for the basic blocks of the if-else construct, their corresponding lineages after lineage fusions and the final sequence.

```
vikram@honneralu-desktop:~/llvm/test$ cat if-else.c
int main()
{
  int i, j, max;
  if (i > j)
    max = i;
  else
    max = j;
  return max;
}
```

Figure 5.10: if-else Structure

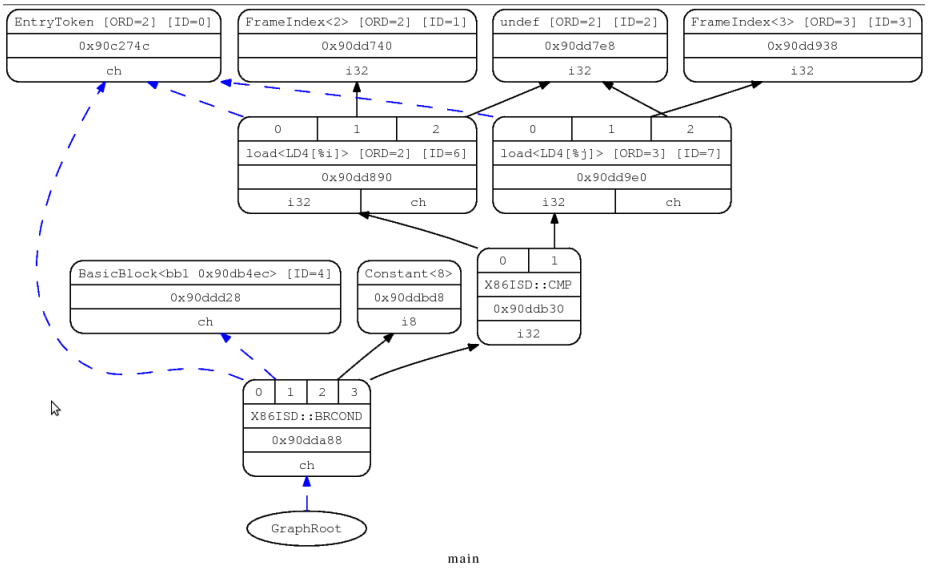

Figure 5.11: BB1 for if-else

```
Lineage 0
Print Lineage: size: 3
0xa45b9d8: i32,ch = load 0xa440734, 0xa45b930, 0xa45b7e0<LD4[%j]> [ID=2]
0xa45bb28: i32 = <<Unknown Node #194>> 0xa45b888, 0xa45b9d8 [ID=1]
0xa45ba80: ch = <<Unknown Node #201>> 0xa440734, 0xa45bd20, 0xa45bbd0, 0xa45bb28 [ID=0]

Lineage 1
Print Lineage: size: 2
0xa45b888: i32,ch = load 0xa440734, 0xa45b738, 0xa45b7e0<LD4[%i]> [ID=4]
0xa45bb28: i32 = <<Unknown Node #194>> 0xa45b888, 0xa45b9d8 [ID=1]

Sequence:
0xa45b888: i32,ch = load 0xa440734, 0xa45b738, 0xa45b7e0<LD4[%i]> [ID=4]
0xa45b9d8: i32,ch = load 0xa440734, 0xa45b930, 0xa45b7e0<LD4[%j]> [ID=2]
0xa45bb28: i32 = <<Unknown Node #194>> 0xa45b888, 0xa45b9d8 [ID=1]
0xa45ba80: ch = <<Unknown Node #201>> 0xa440734, 0xa45bd20, 0xa45bbd0, 0xa45bb28 [ID=0]
```

Figure 5.12: Lineages and final sequence for BB1 of if-else

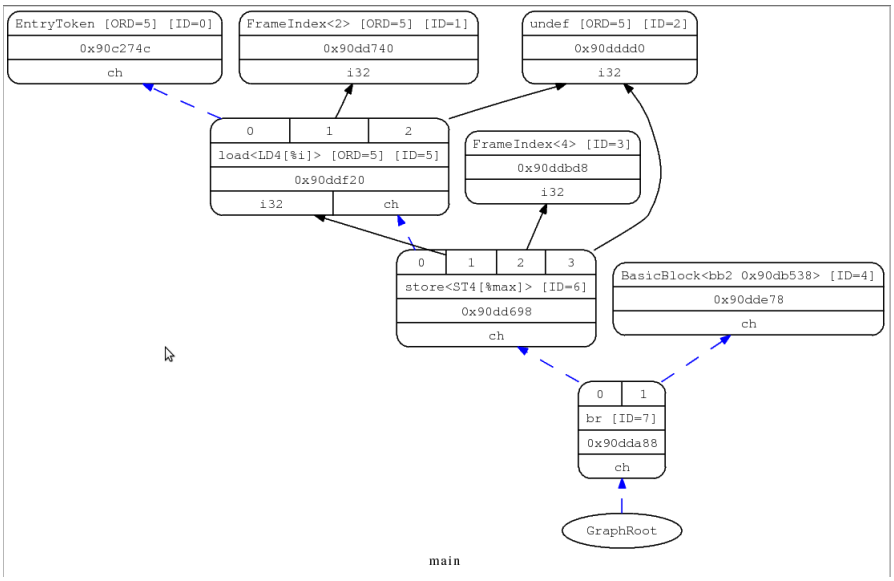

Figure 5.13: BB2 for if-else

```
Lineage 0
Print Lineage: size: 2
0xa4b9f08: i32,ch = load 0xa49e734, 0xa4b9728, 0xa4b9db8<LD4[%i]> [ID=3]
0xa4b9680: ch = store 0xa4b9f08:1, 0xa4b9f08, 0xa4b9bc0, 0xa4b9db8<ST4[%max]> [ID=1]

Sequence:
0xa4b9f08: i32,ch = load 0xa49e734, 0xa4b9728, 0xa4b9db8<LD4[%i]> [ID=3]
0xa4b9680: ch = store 0xa4b9f08:1, 0xa4b9f08, 0xa4b9bc0, 0xa4b9db8<ST4[%max]> [ID=1]
```

Figure 5.14: Lineages and final sequence for BB2 of if-else

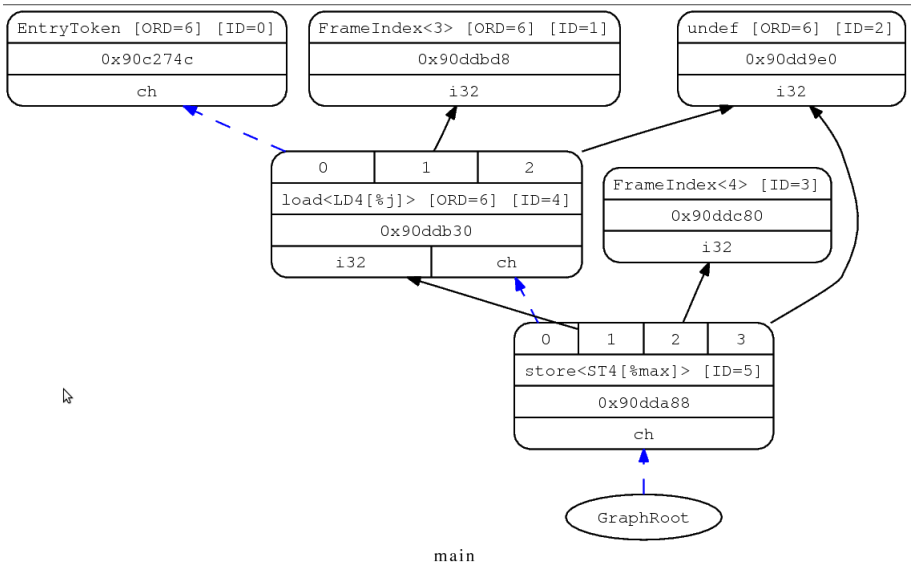

Figure 5.15: BB3 for if-else

```
Lineage 0
Print Lineage: size: 2
0xa8f0b20: i32,ch = load 0xa8d5734, 0xa8f0bc8, 0xa8f09d0<LD4[%j]> [ID=2]
0xa8f0a78: ch = store 0xa8f0b20:1, 0xa8f0b20, 0xa8f0c70, 0xa8f09d0<ST4[%max]> [ID=0]

Sequence:
0xa8f0b20: i32,ch = load 0xa8d5734, 0xa8f0bc8, 0xa8f09d0<LD4[%j]> [ID=2]
0xa8f0a78: ch = store 0xa8f0b20:1, 0xa8f0b20, 0xa8f0c70, 0xa8f09d0<ST4[%max]> [ID=0]
```

Figure 5.16: Lineages and final sequence for BB3 of if-else

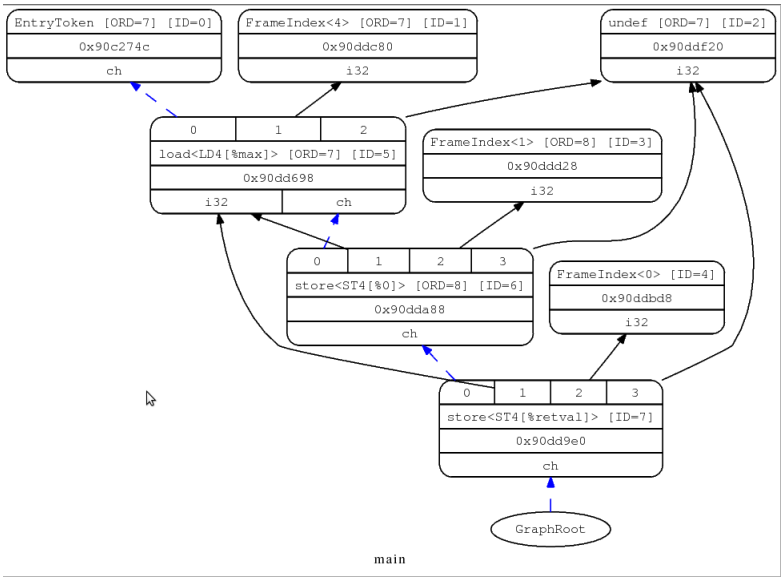

Figure 5.17: BB4 for if-else

```
Lineage 0
Print Lineage: size: 2
0x9d73690: i32,ch = load 0x9d58734, 0x9d73c78, 0x9d73f18<LD4[%max]> [ID=2]
0x9d73a80: ch = store 0x9d73690:1, 0x9d73690, 0x9d73d20, 0x9d73f18<ST4[%0]> [ID=3]

Lineage 1
Print Lineage: size: 1
0x9d739d8: ch = store 0x9d73a80, 0x9d73690, 0x9d73bd0, 0x9d73f18<ST4[%retval]> [ID=0]
  Sequencing Edged nodes:
0x9d739d8: ch = store 0x9d73a80, 0x9d73690, 0x9d73bd0, 0x9d73f18<ST4[%retval]> [ID=0]
0x9d73a80: ch = store 0x9d73690:1, 0x9d73690, 0x9d73d20, 0x9d73f18<ST4[%0]> [ID=3]
Sequence:
0x9d73690: i32,ch = load 0x9d58734, 0x9d73c78, 0x9d73f18<LD4[%max]> [ID=2]
0x9d739d8: ch = store 0x9d73a80, 0x9d73690, 0x9d73bd0, 0x9d73f18<ST4[%retval]> [ID=0]
0x9d73a80: ch = store 0x9d73690:1, 0x9d73690, 0x9d73d20, 0x9d73f18<ST4[%0]> [ID=3]
```

Figure 5.18: Lineages and final sequence for BB4 of if-else

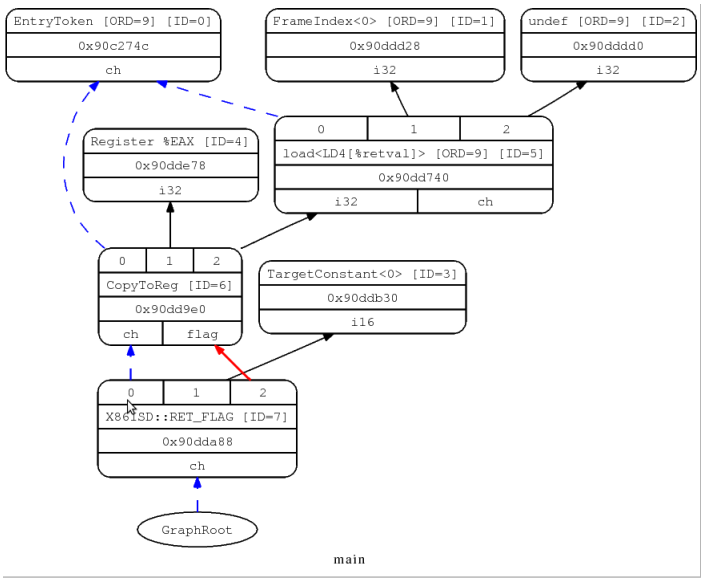

Figure 5.19: BB5 for if-else: *max* gets returned

```
Lineage 0
Print Lineage: size: 1
0xaaa9728: i32,ch = load 0xaa8e734, 0xaaa9d10, 0xaaa9db8<LD4[%retval]> [ID=1]
  Sequencing Edged nodes:
0xaaa9a70: ch = <<Unknown Node #202>> 0xaaa99c8, 0xaaa9b18, 0xaaa99c8:1 [ID=0]
0xaaa99c8: ch,flag = CopyToReg 0xaa8e734, 0xaaa9e60, 0xaaa9728 [ID=0]
Sequence:
0xaaa9728: i32,ch = load 0xaa8e734, 0xaaa9d10, 0xaaa9db8<LD4[%retval]> [ID=1]
```

Figure 5.20: Lineages and final sequence for BB5 of if-else

Chapter 6

Conclusions and Future Work

In this project we try to improve on the traditional instruction scheduling problem by implementing the MRIS approach. In traditional instruction scheduling, the emphasis on reducing the execution time of the program. The scheduler takes into account the execution latencies of different instructions and availability of functional units, and then schedules a sequence which gives a minimum execution time. Here the scheduler does not optimize the number of register used in the execution. The scheduler tries to minimize the code length (schedule length) using a fixed number of registers. In contrast the MRIS approach emphasises on reducing the number of registers used. The MRIS problem is closely related to *optimal code generation* problem.

MRIS approach uses heuristic approach by calculating lineages, interference graph and uses an efficient list scheduling algorithm to find a near optimum solution for the problem. The register allocator then allocates available registers to each of the lineages. The coloring algorithm estimates HRB number of registers and in most of the cases with the number of registers actually used is less than or equal to the HRB value predicted. In some cases where it is impossible to allocate registers using only the HRB number, we increment the number of registers by one until a legal instruction sequence is generated

with as little extra registers used.

The traditional instruction scheduler does not try to minimize the register pressure and hence the register allocator has to spill instructions when the register pressure increases beyond the total number of available register. The spilled instructions will hinder the performance and execution time of the program. MRIS approach minimizes the number of spills thus improving the execution time. MRIS approach might also allow a smaller cache and reduce the traffic of data between the processor and the main memory. We like to evaluate the performance of our heuristic approach of MRIS problem on various benchmarks. The experimental study and evaluating the performance leaves the scope for future work.

Bibliography

[1] A.V. Aho, R. Sethi, and J.D. Ullman, *"Compilers - Principles, Techniques, and Tools"*, corrected ed. Reading, Mass.: Addison-Wesley, 1988.

[2] A.V. Aho, J.E. Hopcroft, J.D. Ullman: The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

[3] Andrew W. Appel, *"Modern Compiler Implementation in C"*, Revised Edition, Cambridge University Press, 2000.

[4] P. Briggs, K.D. Cooper, and L. Torczon, *"Improvements to Graph Coloring Register Allocation"*, *ACM Trans. Programming Languages and Systems*, May 1994.

[5] G.J. Chaitin, *"Register Allocation and Spilling via Graph Coloring"*, Proc. *SIGPLAN '82 Symp. Compiler Construction*, June 1982.

[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: Introduction to Algorithms. 2nd edition, The MIT Press, 2001.

[7] P.B. Gibbons and S.S. Muchnick, *"Efficient Instruction Scheduling for a Pipelined Architecture"*, *Proc. SIGPLAN '86 Symp. Compiler Construction*, June 1986.

[8] R. Govindarajan, H. Yang, J.N. Amaral, C. Zhang, and G.R. Gao, *"Minimum Register Instruction Sequencing to Reduce Register Spills in Out-Of-Order Issue Superscalar Architectures"*, IEEE Transactions on Computers, Vol. 52, No. 1, January 2003.

[9] Leslie Lamport, *LaTeX: A Document Preparation System.* Addison-Wesley, Massachusetts, 2nd Edition, 1994.

[10] S.B. Lippman, and J. Lajoie, *"C++ Primer"*, Third Edition, Addison-Wesley, March 1998.

[11] S.S. Muchnick, *"Advanced Compiler Design and Implementation"*. San Francisco: Morgan Kaufmann, 1997.

[12] D.A. Patterson, and J.L. Hennessy, *"Computer Architecture : A Quantitative Approach"*, Reading, Morgan Kaufmann, 4th ed., 2007

[13] D.A. Patterson, and J.L. Hennessy, *"Computer Organization and Design: The Hardware/Software Interface"*, Second Edition, Harcourt Asia, Morgan Kaufmann, 1998.

[14] M. Poletto and V. Sarkar, *"Linear Scan Register Allocation"*, *ACM Trans. Programming Languages and Systems*, 1998.

[15] H. Schildt, *"C++ : The Complete Reference"*, Third Edition, Osborne McGraw-Hill.

[16] Source code http://llvm.org/