Introduction to Optimizing Compilers

EPIC and Compiler Optimization

- EPIC requires dependency free “scheduled code”
- Burden of extracting parallelism falls on compiler
- Success of EPIC architectures depends on efficiency of Compilers!!
- We provide overview of Compiler Optimization techniques (as they apply to EPIC/ILP)
  - enhanced by examples using Trimaran ILP Infrastructure

Hardware-Software Interface

Machine
- Available resources statically fixed
- Designed to support wide variety of programs
- Interested in running many programs fast

Program
- Required resources dynamically varying
- Designed to run well on a variety of machines
- Interested in having itself run fast

Performance = \( t_{opt} \times CPI \times \text{code size} \)

Compiler Tasks

- Code Translation
  - Source language \( \rightarrow \) target language
    - FORTRAN \( \rightarrow \) C
    - C \( \rightarrow \) MIPS, PowerPC or Alpha machine code
    - MIPS binary \( \rightarrow \) Alpha binary

- Code Optimization
  - Code runs faster
  - Match dynamic code behavior to static machine structure
Compiler Structure

Front End

- High-level source code

Optimizer

IR

Back End

Machine independent

Machine dependent

(machine code)

(Intermediate representation)

Structure of Optimizing Compilers

Front End

- Lexical Analysis
  - Misspelling an identifier, keyword, or operator e.g. lex

- Syntax Analysis
  - Grammar errors, such as mismatched parentheses e.g. yacc

- Semantic Analysis
  - Type checking

Front-end

1. Scanner - converts input character stream into stream of lexical tokens

2. Parser - derives syntactic structure (parse tree, abstract syntax tree) from token stream, and reports any syntax errors encountered
### Front-end

3. Semantic Analysis - generates intermediate language representation from input source program and user options/directives, and reports any semantic errors encountered.

### High-level Optimizer

- Global intra-procedural and inter-procedural analysis of source program’s control and data flow.
- Selection of high-level optimizations and transformations.
- Update of high-level intermediate language.

### Intermediate Representation

- **Achieve retargetability**
  - Different source languages
  - Different target machines
- **Example (tree-based IR from CMCC)**

```
int a, b, c, d;
d = a * (b+c);
```

<table>
<thead>
<tr>
<th>Linear form of</th>
<th>Graphical Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0 5 78 &quot;a&quot;</td>
<td>ASGI</td>
</tr>
<tr>
<td>A1 5 78 &quot;b&quot;</td>
<td></td>
</tr>
<tr>
<td>A2 5 78 &quot;c&quot;</td>
<td></td>
</tr>
<tr>
<td>A3 5 78 &quot;d&quot;</td>
<td></td>
</tr>
<tr>
<td>FND1 ADDRL A3</td>
<td>INDRI</td>
</tr>
<tr>
<td>FND2 ADDRL A0</td>
<td>ADDI  INDRI</td>
</tr>
<tr>
<td>FND3 ADDRL FND2</td>
<td></td>
</tr>
<tr>
<td>FND4 ADDRL A1</td>
<td></td>
</tr>
<tr>
<td>FND5 ADDRL FND4</td>
<td></td>
</tr>
<tr>
<td>FND6 ADDRL A2</td>
<td></td>
</tr>
<tr>
<td>FND7 ADDRL FND6</td>
<td></td>
</tr>
<tr>
<td>FND8 ADDI FND5</td>
<td></td>
</tr>
<tr>
<td>FND9 MULI FND3</td>
<td></td>
</tr>
<tr>
<td>FND10 ASGI FND1</td>
<td></td>
</tr>
</tbody>
</table>

### Lowering of Intermediate Language

- Linearized storage mapping of variables.
- Array/structure references → load/store operations.
- High-level control structures → low-level control flow.
Machine-Independent Optimizations

- Dataflow Analysis and Optimizations
  - Constant propagation
  - Copy propagation
  - Value numbering

- Elimination of common subexpression

Machine-Independent Optimizations

- Dead code elimination
- Strength reduction
- Function/Procedure inlining

Code-Optimizing Transformations

- Constant folding
  \[(1 + 2) \Rightarrow 3\]
  \[(100 > 0) \Rightarrow \text{true}\]

- Copy propagation
  \[x = b + c \quad \Rightarrow \quad x = b + c\]
  \[z = y \cdot x \quad \Rightarrow \quad z = y \cdot (b + c)\]

- Common subexpression
  \[x = b \cdot c + 4 \quad \Rightarrow \quad t = b \cdot c\]
  \[z = b \cdot c - 1 \quad \Rightarrow \quad x = t + 4\]
  \[z = t - 1\]

- Dead code elimination
  \[x = 1 \quad \Rightarrow \quad x = b + c\]
  or if \(x\) is not referred to at all

Code Optimization Example

- Constant folding
  \[x = 1\]
  \[y = a \cdot b + 3\]
  \[z = a \cdot b + x + z + 2\]
  \[x = 3\]

- Propagation
  \[x = 1\]
  \[y = a \cdot b + 3\]
  \[z = a \cdot b + 1 + z + 2\]
  \[x = 3\]

- Dead code elimination
  \[y = 1 \quad \Rightarrow \quad y = a \cdot b + 3\]
  \[z = a \cdot b + 3 + z\]
  \[x = 3\]

- Common subexpression
  \[t = a \cdot b + 3\]
  \[y = t\]
  \[z = t + z\]
  \[x = 3\]
**Code Motion**

- Move code between basic blocks
- E.g. move loop invariant computations outside of loops

```
t = x / y
while (i < 100) {
    *p = t + i
    i = i + 1
}
```

**Strength Reduction**

- Replace complex (and costly) expressions with simpler ones
- E.g.
  
  
  ```
a := b*17
  a := (b<<4) + b
  ```

```
while (i < 100) {
    a[i] = i * 100
    t = t + 100
    p = p + 4
    i = i + 1
}
```

**Induction variable elimination**

- Induction variable: loop index.
- Consider loop:
  
  ```
  for (i=0; i<N; i++)
      for (j=0; j<M; j++)
          z[i][j] = b[i][j];
  ```

- Rather than recompute i*M+j for each array in each iteration, share induction variable between arrays, increment at end of loop body.
Loop Optimizations

- Motivation: restructure program so as to enable more effective back-end optimizations and hardware exploitation
- Loop transformations are useful for enhancing
  - register allocation
  - instruction-level parallelism
  - data-cache locality
  - vectorization
  - parallelization

Importance of Loop Optimizations

<table>
<thead>
<tr>
<th>Program</th>
<th>No. of Loops</th>
<th>Static B.B. Count</th>
<th>Dynamic B.B. Count</th>
<th>% of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>nasa7</td>
<td>9</td>
<td>---</td>
<td>322M</td>
<td>64%</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>---</td>
<td>362</td>
<td>72%</td>
</tr>
<tr>
<td></td>
<td>83</td>
<td>---</td>
<td>500</td>
<td>~100%</td>
</tr>
<tr>
<td>matrix300</td>
<td>1</td>
<td>17</td>
<td>217.6M</td>
<td>98%</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>96</td>
<td>221.2M</td>
<td>98%+</td>
</tr>
<tr>
<td>tomtuv</td>
<td>1</td>
<td>7</td>
<td>26.1M</td>
<td>50%</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>22</td>
<td>52.4M</td>
<td>99%+</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>96</td>
<td>54.2M</td>
<td>~100%</td>
</tr>
</tbody>
</table>

Study of loop-intensive benchmarks in the SPEC92 suite [C.J. Newburn, 1991]

Loop Optimizations

- Loops are good targets for optimization.
- Basic loop optimizations:
  - code motion;
  - induction-variable elimination;
  - strength reduction ($x^2 \rightarrow xx$).
- Improve performance by unrolling the loop
  - Note impact when using processors that allow parallel execution of instructions
    - Texas Instruments new DSP processors

Function inlining

- Replace function calls with function body
- Increase compilation scope (increase ILP)
  e.g. constant propagation, common subexpression
- Reduce function call overhead
  e.g. passing arguments, reg. saves and restores

[W.M. Hwu, 1991 (DEC 3100)]

<table>
<thead>
<tr>
<th>Program</th>
<th>In-line Speedup</th>
<th>In-line Code Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccop</td>
<td>1.06</td>
<td>1.25</td>
</tr>
<tr>
<td>compress</td>
<td>1.05</td>
<td>1.04</td>
</tr>
<tr>
<td>equ</td>
<td>1.12</td>
<td>1.21</td>
</tr>
<tr>
<td>expresso</td>
<td>1.07</td>
<td>1.09</td>
</tr>
<tr>
<td>lex</td>
<td>1.02</td>
<td>1.06</td>
</tr>
<tr>
<td>tbl</td>
<td>1.04</td>
<td>1.18</td>
</tr>
<tr>
<td>xllisp</td>
<td>1.46</td>
<td>1.32</td>
</tr>
<tr>
<td>yacc</td>
<td>1.03</td>
<td>1.17</td>
</tr>
</tbody>
</table>
Back End

- IR → Back End → Machine code
  - map virtual registers into architect registers
  - rearrange code
  - target machine specific optimizations
    - delayed branch
    - conditional move
    - instruction combining
    - auto increment addressing mode
    - add carrying (PowerPC)
    - hardware branch (PowerPC)

Machine Dependent Optimizations

- Register Allocation
- Instruction Scheduling
- Peephole Optimizations

Code Selection

- Map IR to machine instructions (e.g. pattern matching)

```c
Inst *match(IR *n) { 
  switch (n->opcode) { 
    case MUL: 
      l = match(n->left()); 
      r = match(n->right()); 
      if (n->type == D || n->type == F) 
        inst = mul_fp((n->type == D), l, r); 
      else 
        inst = mul_int((n->type == I), l, r); 
      break; 
    case ADD: 
      l = match(n->left()); 
      r = match(n->right()); 
      if (n->type == D || n->type == F) 
        inst = add_fp((n->type == D), l, r); 
      else 
        inst = add_int((n->type == I), l, r); 
      break; 
    case .......: 
      ....... 
    } 
  return inst; 
}
```

Machine Dependent Optimizations

- Register Allocation
- Instruction Scheduling
- Peephole Optimizations

Peephole Optimizations

- Replacements of assembly instruction through template matching

- Eg. Replacing one addressing mode with another in a CISC
**Code Scheduling**

- Rearrange code sequence to minimize execution time
  - Hide instruction latency
  - Utilize all available resources

```
Ld f4, 8(r8)  
Fadd f5, f4, f6  
Ld f2, 16(r8)  
Fsub f7, f2, f6  
Fmul f7, f7, f5  
S.d f7, 24(r8)  
Ld f8, 0(r9)  
S.d f8, 8(r9)  
```

1 stall  
1 stall  
3 stalls  
1 stall  

(reorder)

```
Ld f4, 8(r8)  
Fadd f5, f4, f6  
Ld f2, 16(r8)  
Fsub f7, f2, f6  
Fmul f7, f7, f5  
S.d f7, 24(r8)  
Ld f8, 0(r9)  
S.d f8, 8(r9)  
```

3 stalls  
1 stall  
0 stall  
0 stall  
0 stall  

(reorder)

```
Ld f4, 8(r8)  
Fadd f5, f4, f6  
Fmul f7, f7, f5  
L.d f8, 0(r9)  
Fsub f7, f2, f6  
S.d f7, 24(r8)  
```

0 stalls  
0 stall  
0 stall  
0 stall  

(reorder)

**Cost of Instruction Scheduling**

- Given a program segment, the goal is to execute it as quickly as possible

- The completion time is the **objective function** or cost to be minimized

- This is referred to as the **makespan** of the schedule

- It has to be balanced against the running time and space needs of the algorithm for finding the schedule, which translates to **compilation cost**

---

**Instruction Scheduling Example**

```c
main(int argc, char *argv[])
{
    int a, b, c;
    a = argc;
    b = a * 255;
    c = a * 15;
    printf("%d\n", b*b - 4*a*c);
}
```

```
op 10 MPY vr2 
¬ param1, 255
op 12 MPY vr3 
¬ param1, 15
op 14 MPY vr8 
¬ vr2, vr2
op 15 SHL vr9 
¬ param1, 2
op 16 MPY vr10 
¬ vr9, r3
op 17 SUB param2 
¬ vr8, r10
op 18 MOV param4 
¬ addr("%d\n")
op 27 PBRR vb12 
¬ addr(printf)
op 20 BRL ret_addr 
¬ vb12
```

**After Scheduling**
(Prior to Register Allocation)
The General Instruction Scheduling Problem

Feasible Schedule: A specification of a start time for each instruction such that the following constraints are obeyed:

1. Resource: Number of instructions of a given type of any time < corresponding number of FUs
2. Precedence and Latency: For each predecessor $j$ of an instruction $i$ in the DAG, $i$ is started only $\delta$ cycles after $j$ finishes where $\delta$ is the latency labeling the edge $(j,i)$.

Output: A schedule with the minimum overall completion time

Input: A basic block represented as a DAG

- $i_2$ is a load instruction.
- Latency of 1 on $(i_2,i_4)$ means that $i_4$ cannot start for one cycle after $i_2$ completes.

Two schedules for the above DAG with $S_2$ as the desired sequence.
Why Register Allocation?

• Storing and accessing variables from registers is much faster than accessing data from memory.
  – Variables ought to be stored in registers
• It is useful to store variables as long as possible, once they are loaded into registers
• Registers are bounded in number
  – “register-sharing” is needed over time.

Register Allocation

• Map virtual registers into physical registers
  – minimize register usage to reduce memory accesses
  – but introduces false dependencies . . . . 

The Goal

• Primarily to assign registers to variables
• However, the allocator runs out of registers quite often
• Decide which variables to “flush” out of registers to free them up, so that other variables can be bought in
  – Spilling

Cost of Register Allocation

• Therefore, maximizing the duration of operands in registers or minimizing the amount of spilling, is the goal
• Once again, the running time (complexity) and space used, of the algorithm for doing this is the compilation cost
Register Allocation and Assignment

- Allocation: identifying program values (virtual registers, live ranges) and program points at which values should be stored in a physical register.
- Program values that are not allocated to registers are said to be spilled.
- Assignment: identifying which physical register should hold an allocated value at each program point.

Performance analysis

- Elements of program performance (Shaw):
  - execution time = program path + instruction timing
- Path depends on data values. Choose which case you are interested in.
- Instruction timing depends on pipelining, cache behavior.

Programs and performance analysis

- Best results come from analyzing optimized instructions, not high-level language code:
  - non-obvious translations of HLL statements into instructions;
  - code may move;
  - cache effects are hard to predict.
- Importance of compiler:
  - Back-end of compiler

Instruction timing

- Not all instructions take the same amount of time.
  - Hard to get execution time data for instructions.
- Instruction execution times are not independent.
- Execution time may depend on operand values.
Trace-driven performance analysis

- **Trace**: a record of the execution path of a program.
- Trace gives execution path for performance analysis.
- A useful trace:
  - requires proper input values;
  - is large (gigabytes).
- Trace generation in H/W or S/W?

---

Execution Frequencies?

---

What are Execution Frequencies

- Branch probabilities
- Average number of loop iterations
- Average number of procedure calls

---

How are Execution Frequencies Used?

- Focus optimization on most frequently used regions
  - region-based compilation
- Provides quantitative basis for evaluating quality of optimization heuristics
How are Execution Frequencies Obtained?

- Profiling tools:
  - Mechanism: sampling vs. counting
  - Granularity: procedure vs. basic block
- Compile-time estimation:
  - Default values
  - Compiler analysis
  - Goal is to select same set of program regions and optimizations that would be obtained from profiled frequencies

What are Execution Costs?

Cost of intermediate code operation parametrized according to target architecture:
- Number of target instructions
- Resource requirement template
- Number of cycles

How are Execution Costs Used?

In conjunction with execution frequencies:
- Identify most time-consuming regions of program
- Provides quantitative basis for evaluating quality of optimization heuristics

How are Execution Costs Obtained?

- Simplistic translation of intermediate code operation to corresponding instruction template for target machine
Working Set Size and Compilation Cost

- Consider a standard memory hierarchy:
  - Disk ⇔ Main-memory ⇔ Cache ⇔ Registers
- In theory each access to memory takes constant time
- In practice, these constants can vary by orders of magnitude depending on the hierarchy and the size of its working set

An Example

Compilation job:
- Working set size = 40 MB
- Running time $T(n) = c_1 \cdot n$, where $c_1$ is the cost of a memory access
  - $n = 10^6$

Compilation on the first platform:
- Allocated main memory = 64 Mb
- $c_1 = 1$ millisecond
- Compilation time = 1 second

An Example (contd.)

Compilation on the second platform:
- Allocated main memory = 32 Mb
- $c_1 = 1$ millisecond
- Compilation time = 1000 seconds ~ 0.3 hours

Cost Functions

- Effectiveness of the Optimizations: How well can we optimize our objective function?
  Impact on running time of the compiled code determined by the completion time.

- Efficiency of the optimization: How fast can we optimize?
  Impact on the time it takes to compile or cost for gaining the benefit of code with fast running time.
Basic Graphs

- A graph is made up of a set of nodes ($V$) and a set of edges ($E$).

- Each edge has a source and a sink, both of which must be members of the nodes set, i.e. $E = V \times V$.

- Edges may be directed or undirected:
  - A directed graph has only directed edges.
  - A undirected graph has only undirected edges.

### Examples

- **Undirected graph**
- **Directed graph**
**Paths**

- Undirected graph
- Directed graph

**Cycles**

- Undirected graph
- Directed graph
- Acyclic Directed graph

**Connected Graphs**

- Unconnected graph
- Connected directed graph

**Connectivity of Directed Graphs**

- A strongly connected directed graph is one which has a path from each vertex to every other vertex

- Is this graph strongly connected?
The Program Dependence Graph (PDG) is the intermediate (abstract) representation of a program designed for use in optimizations.

It consists of two important graphs:
- Control Dependence Graph captures control flow and control dependence
- Data Dependence Graph captures data dependences

**Control Flow Graph: Definition**

A control flow graph \(CFG = (N_c, E_c, T_c)\) consists of:

- \(N_c\), a set of nodes. A node represents a straight-line sequence of operations with no intervening control flow i.e. a basic block.
- \(E_c \subseteq N_c \times N_c\) \(\times\) Labels, a set of labeled edges.
- \(T_c\), a node type mapping. \(T_c(n)\) identifies the type of node \(n\) as one of: START, STOP, OTHER.

We assume that \(CFG\) contains a unique START node and a unique STOP node, and that for any node \(N\) in \(CFG\), there exist directed paths from START to \(N\) and from \(N\) to STOP.

**Control Flow Graphs**

- Motivation: language-independent and machine-independent representation of control flow in programs used in high-level and low-level code optimizers. The flow graph data structure lends itself to use of several important algorithms from graph theory.
Data and Control Dependences

Motivation: identify only the essential control and data dependences which need to be obeyed by transformations for code optimization.

Program Dependence Graph (PDG) consists of:
1. Set of nodes, as in the CFG
2. Control dependence edges
3. Data dependence edges

Together, the control and data dependence edges dictate whether or not a proposed code transformation is legal.

Data Dependence Analysis

If two operations have potentially interfering data accesses, data dependence analysis is necessary for determining whether or not an interference actually exists. If there is no interference, it may be possible to reorder the operations or execute them concurrently.

The data accesses examined for data dependence analysis may arise from array variables, scalar variables, procedure parameters, pointer dereferences, etc. in the original source program.

Data dependence analysis is conservative, in that it may state that a data dependence exists between two statements, when actually none exists.

Data Dependence: Definition

A data dependence, $S_1 \rightarrow S_2$, exists between CFG nodes $S_1$ and $S_2$ with respect to variable $X$ if and only if:

1. there exists a path $P: S_1 \rightarrow S_2$ in CFG, with no intervening write to $X$, and
2. at least one of the following is true:
   (a) (flow) $X$ is written by $S_1$ and later read by $S_2$, or
   (b) (anti) $X$ is read by $S_1$ and later written by $S_2$, or
   (c) (output) $X$ is written by $S_1$ and later written by $S_2$.

Def/Use chaining for Data Dependence Analysis

A def-use chain links a definition $D$ (i.e. a write access of variable $X$ to each use $U$ (i.e. a read access), such that there is a path from $D$ to $U$ in CFG that does not redefine $X$.

Similarly, a use-def chain links a use $U$ to a definition $D$, and a def-def chain links a definition $D$ to a definition $D'$ (with no intervening write to $X$ in all cases).

Def-use, use-def, and def-def chains can be computed by data flow analysis, and provide a simple but conservative way of enumerating flow, anti, and output data dependences.
Impact of Control Flow

- **Acyclic** control flow is easier to deal with than **cyclic** control flow. Problems in dealing with cyclic flow:
  - A loop implicitly represent a large run-time program space compactly.
  - Not possible to open out the loops fully at compile-time.
  - Loop unrolling provides a partial solution.

*more...*

Impact of Control Flow (Contd.)

- Using the loop to optimize its dynamic behavior is a challenging problem.
- Hard to optimize well without detailed knowledge of the range of the iteration.
- In practice, profiling can offer limited help in estimating loop bounds.

Control Dependence Analysis

We want to capture two related ideas with control dependence analysis of a CFG:

1. Node $Y$ should be control dependent on node $X$ if node $X$ evaluates a predicate (conditional branch) which can control whether node $Y$ will subsequently be executed or not. This idea is useful for determining whether node $Y$ needs to wait for node $X$ to complete, even though they have no data dependences.

Control Dependence Analysis (contd.)

2. Two nodes, $Y$ and $Z$, should be identified as having identical control conditions if in every run of the program, node $Y$ is executed if and only if node $Z$ is executed. This idea is useful for determining whether nodes $Y$ and $Z$ can be made adjacent and executed concurrently, even though they may be far apart in the CFG.
Instruction Scheduling Algorithms

Acyclic Instruction Scheduling

- We will consider the case of acyclic control flow first.

- The acyclic case itself has two parts:
  - The simpler case that we will consider first has no branching and corresponds to basic block of code, e.g., loop bodies.
  - The more complicated case of scheduling programs with acyclic control flow with branching will be considered next.

The Core Case: Scheduling Basic Blocks

- Why are basic blocks easy?
- All instructions specified as part of the input must be executed.
- Allows deterministic modeling of the input.
- No “branch probabilities” to contend with; makes problem space easy to optimize using classical methods.

Instruction Scheduling

- Input: A basic block represented as a DAG

- \( i_2 \) is a load instruction.
- Latency of 1 on \((i_2, i_4)\) means that \( i_4 \) cannot start for one cycle after \( i_2 \) completes.
Two schedules for the above DAG with $S_2$ as the desired sequence.

The General Instruction Scheduling Problem

- Input: DAG representing each basic block where:
  - 1. Nodes encode unit execution time (single cycle) instructions.
  - 2. Each node requires a definite class of FUs.
  - 3. Additional pipeline delays encoded as latencies on the edges.
  - 4. Number of FUs of each type in the target machine.

- Feasible Schedule: A specification of a start time for each instruction such that the following constraints are obeyed:
  - 1. Resource: Number of instructions of a given type at any time < corresponding number of FUs.
  - 2. Precedence and Latency: For each predecessor $j$ of an instruction $i$ in the DAG, $i$ is the started only cycles after $j$ finishes where $k$ is the latency labeling the edge $(j,i)$.

- Output: A schedule with the minimum overall completion time (makespan).

Drawing on Deterministic Scheduling

- Canonical List Scheduling Algorithm:
  - 1. Assign a Rank (priority) to each instruction (or node).
  - 2. Sort and build a priority list of the instructions in non-decreasing order of Rank.
    - Nodes with smaller ranks occur earlier
**Drawing on Deterministic Scheduling (Contd.)**

3. **Greedily list-schedule.**
   - Scan iteratively and on each scan, choose the largest number of "ready" instructions subject to resource (FU) constraints in list-order.
   - An instruction is ready provided it has not been chosen earlier and all of its predecessors have been chosen and the appropriate latencies have elapsed.

**Code Scheduling**

- **Objectives:** minimize execution latency of the program
  - Start as early as possible instructions on the critical path
  - Help expose more instruction-level parallelism to the hardware
  - Help avoid resource conflicts that increase execution time

- **Constraints**
  - Program Precedences
  - Machine Resources

- **Motivations**
  - Dynamic/Static Interface (DSI): By employing more software (static) optimization techniques at compile time, hardware complexity can be significantly reduced
  - Performance Boost: Even with the same complex hardware, software scheduling can provide additional performance enhancement over that of unscheduled code

**Precedence Constraints**

- Minimum required ordering and latency between definition and use

- **Precedence graph**
  - Nodes: instructions
  - Edges (a → b): a precedes b
  - Edges are annotated with minimum latency

**Precedence Graph**

- Instructions and their dependencies are annotated with minimum latencies.
Resource Constraints

- Bookkeeping
  - Prevent resources from being oversubscribed

Machine model

add r1, r1, 1
add r2, r2, 4
fadd r1, r1, r2
fadd r3, r3, r4

The Value of Greedy List Scheduling

- Example: Consider the DAG shown below:

Using the list = <i1, i2, i3, i4, i5>

Greedy scanning produces the steps of the schedule as follows:

The Value of Greedy List Scheduling (Contd.)

- 1. On the first scan: i1 which is the first step.
- 2. On the second and third scans and out of the list order, respectively i4 and i5 to correspond to steps two and three of the schedule.
- 3. On the fourth and fifth scans, i2 and i3 respectively scheduled in steps four and five.

List Scheduling for Basic Blocks

1. Assign priority to each instruction
2. Initialize ready list that holds all ready instructions
   - Ready = data ready and can be scheduled
3. Greedily choose one ready instruction I from ready list with the highest priority
   - Possibly using tie-breaking heuristics
4. Insert I into schedule
   - Making sure resource constraints are satisfied
5. Add those instructions whose precedence constraints are now satisfied into the ready list
Rank/Priority Functions/Heuristics
- Number of descendants in precedence graph
- Maximum latency from root node of precedence graph
- Length of operation latency
- Ranking of paths based on importance
- Combination of above

Orientation of Scheduling
- Instruction Oriented
  - Initialization (priority and ready list)
  - Choose one ready instruction and find a slot in schedule
    make sure resource constraint is satisfied
  - Insert into schedule
  - Update ready list
- Cycle Oriented
  - Initialization (priority and ready list)
  - Step through schedule cycle by cycle
  - For the current cycle C, choose one ready instruction
    be sure latency and resource constraints are satisfied
  - Insert into schedule (cycle C)
  - Update ready list

List Scheduling Example
\[(a + b) \ast (c - d) + e/f\]
- load: 2 cycles
- add: 1 cycle
- sub: 1 cycle
- mul: 4 cycles
- div: 10 cycles
- orientation: cycle
- direction: backward
- heuristic: maximum latency to root

Scalar Scheduling Example

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Ready list</th>
<th>Schedule</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CS 211
### Scalar Scheduling Example

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Ready list</th>
<th>Schedule</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>Id f</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>Id a</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>Id b</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>Id c</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>Id d</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>Id e</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>ld (a + b)</td>
<td>fdiv (e/f)</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>ld a</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>17</td>
<td>ld b</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>18</td>
<td>ld c</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>19</td>
<td>ld d</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>20</td>
<td>ld e</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>21</td>
<td>ld f</td>
<td></td>
</tr>
</tbody>
</table>

### Superscalar Scheduling Example

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Ready list</th>
<th>Schedule</th>
<th>Resources</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>Id f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>Id a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>Id b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>Id c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>Id d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>ld a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>ld b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>ld c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td>ld d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>ld e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td>ld f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Some Intuition

- Greediness helps in making sure that idle cycles don’t remain if there are available instructions further “down stream.”
- Ranks help prioritize nodes such that choices made early on favor instructions with greater enabling power, so that there is no unforced idle cycle.

### How Good is Greedy?

- Approximation: For any pipeline depth $k \geq 1$ and any number $m$ of pipelines, $S_{\text{greedy}} / S_{\text{opt}} \leq (2 – 1/mk)$.
How good is greedy?

- For example, with one pipeline ($m=1$) and the latencies grow as 2, 3, 4, ..., the approximate schedule is guaranteed to have a completion time no more 66%, 75%, and 80% over the optimal completion time.
- This theoretical guarantee shows that greedy scheduling is not bad, but the bounds are worst-case; practical experience tends to be much better.

How Good is Greedy? (Contd.)

- Running Time of Greedy List Scheduling: Linear in the size of the DAG.

A Critical Choice: The Rank Function for Prioritizing Nodes

Rank Functions (Contd.)


Optimality: 2 and 3 produce optimal schedules for RISC processors such as the IBM 801, Berkeley RISC and so on.

An Example Rank Function

1. Initially label all the nodes by the same value, say

2. Compute new labels from old starting with nodes at level zero (i4) and working towards higher levels:

   (a) All nodes at level zero get a rank of 0

   (b) For a node at level 1, construct a new label which is the concentration of all its successors connected by a latency 1 edge.

   (c) The empty symbol is associated with latency zero edges.

   (d) The result is that i2 and i3 respectively get new labels and hence ranks 'greater than' = 1

   (e) Rank of i1 is the concentration of the ranks of its immediate successors i2 and i3 i.e., it is 'greater than or equal to' = 2

   3. The resulting sorted list is (optimum) i1, i2, i3, i4.
Limitations of List Scheduling

- Cannot move instructions past conditional branch instructions in the program (scheduling limited by basic block boundaries)
- Problem: Many programs have small numbers of instructions (4-5) in each basic block. Hence, not much code motion is possible.
- Solution: Allow code motion across basic block boundaries.
  - Speculative Code Motion: “jumping the gun”
    - Execute instructions before we know whether or not we need to
    - Utilize otherwise idle resources to perform work which we speculation will need to be done
  - Relies on program profiling to make intelligent decisions about speculation

Getting around basic block limitations

- Basic block size limits amount of parallelism available for extraction
  - Need to consider more “flexible” regions of instructions
- A well known classical approach is to consider traces through the (acyclic) control flow graph.

Traces

- Main Ideas:
  - Choose a program segment that has no cyclic dependences.
  - Choose one of the paths out of each branch that is encountered.
Traces (Contd.)

• Use statistical knowledge based on (estimated) program behavior to bias the choices to favor the more frequently taken branches.

• This information is gained through profiling the program or via static analysis.

• The resulting sequence of basic blocks including the branch instructions is referred to as a trace.

Detecting Speculation and Replication Structurally

• Need tests that can be performed quickly to determine which of the side-effects have to be addressed after code-motion.

• Preferably based on structured information that can be derived from previously computed (and explained) program analysis.

• Decisions that are based on the Control (sub) Component of the Program Dependence Graph (PDG).

Scheduling Control Flow Graphs with Loops (Cycles)

Register Allocation
Revisiting A Typical Optimizing Compiler

Rationale for Separating Register Allocation from Scheduling

- Each of Scheduling and Register Allocation are hard to solve individually, let alone solve globally as a combined optimization.
- So, solve each optimization locally and heuristically “patch up” the two stages.

Why Register Allocation?

- Storing and accessing variables from registers is much faster than accessing data from memory. The way operations are performed in load/store (RISC) processors.
- Therefore, in the interests of performance—if not by necessity — variables ought to be stored in registers.
- For performance reasons, it is useful to store variables as long as possible, once they are loaded into registers.
- Registers are bounded in number (say 32.)
- Therefore, “register-sharing” is needed over time.

The Goal

- Primarily to assign registers to variables.
- However, the allocator runs out of registers quite often.
- Decide which variables to “flush” out of registers to free them up, so that other variables can be bought in.
This important indirect consequence of allocation is referred to as spilling.
Register Allocation and Assignment

- **Allocation**: identifying program values (virtual registers, live ranges) and program points at which values should be stored in a physical register.
- Program values that are not allocated to registers are said to be **spilled**.
- **Assignment**: identifying which physical register should hold an allocated value at each program point.

Live Ranges

Live range of virtual register \( a = (BB1, BB2, BB3, BB4, BB5, BB6, BB7) \).

Def-Use chain of virtual register \( a = (BB1, BB3, BB5, BB7) \).

Computing Live Ranges

Using data flow analysis, we compute for each basic block:

- In the forward direction, the **reaching** attribute.
  - A variable is reaching block \( i \) if a definition or use of the variable reaches the basic block along the edges of the CFG.
- In the backward direction, the **liveness** attribute.
  - A variable is live at block \( i \) if there is a direct reference to the variable at block \( i \) or at some block \( j \) that succeeds \( i \) in the CFG, provided the variable in question is *not redefined* in the interval between \( i \) and \( j \).

Computing Live Ranges (Contd.)

The live range of a variable is the intersection of basic-blocks in CFG nodes in which the variable is live, and the set which it can reach.
Global Register Allocation

- Local register allocation does not store data in registers across basic blocks.
  Local allocation has poor register utilization global register allocation is essential.
- Simple global register allocation: allocate most "active" values in each inner loop.
- Full global register allocation: identify live ranges in control flow graph, allocate live ranges, and split ranges as needed.

**Goal**: select allocation so as to minimize number of load/store instructions performed by optimized program.

Simple Example of Global Register Allocation

- Live range of \(a\) = \{B1, B3\}
- Live range of \(b\) = \{B2, B4\}
- No interference! \(a\) and \(b\) can be assigned to the same register

Another Example of Global Register Allocation

- Live range of \(a\) = \{B1, B2, B3, B4\}
- Live range of \(b\) = \{B2, B4\}
- Live range of \(c\) = \{B3\}
- In this example, \(a\) and \(c\) interfere, and \(c\) should be given priority because it has a higher usage count.

Cost and Savings

- Compilation Cost: running time and space of the global allocation algorithm.
- Execution Savings: cycles saved due to register residence of variables in optimized program execution.
- Contrast with memory-residence which leads to longer execution times.
Interference Graph

• Definition: An interference graph $G$ is an undirected graph with the following properties:

  • (a) each node $x$ denotes exactly one distinct live range $X_x$, and

  • (b) an edge exists between nodes $x$ and $y$ iff $X_x$ and $Y_y$ interfere (overlap), where $X$ and $Y$ are the live ranges corresponding to nodes $x$ and $y$.

Interference Graph Example

Live Ranges

```
a := ...
b := ...
c := ...
d := ...
```

Interference Graph

```
\(a\) Live ranges overlap
\(b\) and hence interfere
```

Node model live ranges

The Classical Approach


• more...
The Classical Approach (Cont’d.)

- These works introduced the key notion of an interference graph for encoding conflicts between the live ranges.
- This notion was defined for the global control flow graph.
- It also introduced the notion of graph coloring to model the idea of register allocation.

Execution Time and Spill-cost

- Spilling: Moving a variable that is currently register resident to memory when no more registers are available, and a new live range needs to be allocated one spill.
- Minimizing Execution Cost: Given an optimistic assignment—i.e., one where all the variables are register-resident, minimizing spilling.

Graph Coloring

- Given an undirected graph G and a set of k distinct colors, compute a coloring of the nodes of the graph i.e., assign a color to each node such that no two adjacent nodes get the same color.
- Recall that two nodes are adjacent iff they have an edge between them.
- A given graph might not be k-colorable.
- In general, it is a computationally hard problem to color a given graph using a given number k of colors.
- The register allocation problem uses good heuristics for coloring.

Register Interference & Allocation

- Interference Graph: G = <E,V>  
  - Nodes (V) = variables, (more specifically, their live ranges)
  - Edges (E) = interference between variable live ranges
- Graph Coloring (vertex coloring)  
  - Given a graph, G=<E,V>, assign colors to nodes (V) so that no two adjacent (connected by an edge) nodes have the same color
  - A graph can be "k-colored" if no more than k colors are needed to color the graph.
  - The chromatic number of a graph is min(\(n\)) such that it can be k-colored
- k-coloring is an NP-complete problem, therefore optimal solution can take a long time to compute

How is graph coloring related to register allocation?
Register Allocation as Coloring

- Given \( k \) registers, interpret each register as a color.
- The graph \( G \) is the interference graph of the given program.
- The nodes of the interference graph are the executable live ranges on the target platform.
- A coloring of the interference graph is an assignment of registers (colors) to live ranges (nodes).
- Running out of colors implies not enough registers and hence a need to spill in the above model.

Interference Graph

```
ld r4, 16(r3)
sub r6, r2, r4
add r7, r7, 1
blt   r7, 100
ld r5, 24(r3)
beq  r2, $0
add r2, r1, r5
sw r6, 8(r3)
```

```
r1
r2
r3
r4
r5
r6
r7
```

“Live variable analysis”

1st & 3rd are live-in
1st & 3rd are live-out

Chaitin’s Graph Coloring Theorem

- Key observation: If a graph \( G \) has a node \( X \) with degree less than \( n \) (i.e. having less than \( n \) edges connected to it), then \( G \) is \( n \)-colorable IFF the reduced graph \( G' \) obtained from \( G \) by deleting \( X \) and all its edges is \( n \)-colorable.

Proof:

```
G
n-1
```

Graph Coloring Algorithm (Not Optimal)

- Assume the register interference graph is \( n \)-colorable
- How do you choose \( n \)?
- Simplification
  - Remove all nodes with degree less than \( n \)
  - Repeat until the graph has \( n \) nodes left
- Assign each node a different color
- Add removed nodes back one-by-one and pick a legal color as each one is added (2 nodes connected by an edge get different colors)
- Must be possible with less than \( n \) colors

Complications: simplification can block if there are no nodes with less than \( n \) edges
Register Spilling

When simplification is blocked, pick a node to delete from the graph in order to unblock

- Deleting a node implies the variable it represents will not be kept in register (i.e. spilled into memory)
  - When constructing the interference graph, each node is assigned a value indicating the estimated cost to spill it.
  - The estimated cost can be a function of the total number of definitions and uses of that variable weighted by its estimated execution frequency.
  - When the coloring procedure is blocked, the node with the least spilling cost is picked for spilling.
- When a node is spilled, spill code is added into the original code to store a spilled variable at its definition and to reload it at each of its use
- After spill code is added, a new interference graph is rebuilt from the modified code, and n-coloring of this graph is again attempted

The Alternate Approach: more common

- an alternate approach used widely in most compilers
  - also uses the Graph Coloring Formulation
  - Hennessy, Founder of MIPS, President of Stanford Univ!
Important Modeling Difference

• The first difference from the classical approach is that now, we assume that the “home location” of a live range is in memory.
  – Conceptually, values are always in memory unless promoted to a register; this is also referred to as the pessimistic approach.
  – In the classical approach, the dual of this model is used where values are always in registers except when spilled; recall that this is referred to as the optimistic approach.

Important Modeling Difference (Contd.)

• A second major difference is the granularity at which code is modeled.
  – In the classical approach, individual instructions are modeled whereas now, basic blocks are the primitive units modeled as nodes in live ranges and the interference graph.
• The final major difference is the place of the register allocation in the overall compilation process.
  – In the present approach, the interference graph is considered earlier in the compilation process using intermediate level statements; compiler generated temporaries are known.
  – In contrast, in the previous work the allocation is done at the level of the machine code.

Rationale for Separating Register Allocation from Scheduling

• Each of Scheduling and Register Allocation are hard to solve individually, let alone solve globally as a combined optimization.
• So, solve each optimization locally and heuristically “patch up” the two stages.

The Approach Discussed Here

Important Modeling Difference

• The first difference from the classical approach is that now, we assume that the "home location" of a live range is in memory.
  – Conceptually, values are always in memory unless promoted to a register; this is also referred to as the pessimistic approach.
  – In the classical approach, the dual of this model is used where values are always in registers except when spilled; recall that this is referred to as the optimistic approach.

more...

The Main Information to be Used by the Register Allocator

• For each live range, we have a bit vector LIVE of the basic blocks in it.
• Also we have INTERFERE which gives for the live range, the set of all other live ranges that interfere with it.
• Recall that two live ranges interfere if they intersect in at least one (basic-block).
• If |INTERFERE| is smaller than the number of available of registers for a node \( i \), then \( i \) is unconstrained; it is constrained otherwise.

more...
Prioritizing Live Ranges

In the memory bound approach, given live ranges with a choice of assigning registers, we do the following:

• Choose a live range that is “likely” to yield greater savings in execution time.
• This means that we need to estimate the savings of each basic block in a live range.

Estimate the Savings

Given a live range $X$ for variable $x$, the estimated savings in a basic block $i$ is determined as follows:

1. First compute $CyclesSaved$ which is the number of loads and stores of $x$ in $i$ scaled by the number of cycles taken for each load/store.

2. Compensate the single load and/or store that might be needed to bring the variable in and/or store the variable at the end and denote it by $Setup$.

Note that $Setup$ is derived from a single load or store or a load plus a store.

Estimate the Savings (Contd.)

3. $Savings(X,i) = \{CyclesSaved - Setup\}$

These indicate the actual savings in cycles after accounting for the possible loads/stores needed to move $x$ at the beginning/end of $i$.

4. $TotalSavings(X) = \sum_{i \in X} Savings(X,i) \times W(i)$.
   (a) $x$ is the set of all basic blocks in the live range of $X$.
   (b) $W(i)$ is the execution frequency of variable $x$ in block $i$.

5. Note however that live regions might span a few blocks but yield a large savings due to frequent use of the variable while others might yield the same cumulative gain over a larger number of basic blocks. We prioritize the former case and define:

$$\{Priority(X) = TotalSavings(X) / Span(X)\}$$

where $Span(X)$ is the number of basic blocks in $X$. 

more...
The Algorithm

For all constrained live ranges, execute the following steps:

1. Compute \( \text{Priority}(X) \) if it has not already been computed.
2. For the live range \( X \) with the highest priority:
   (a) If its priority is negative or if no basic block \( i \) in \( X \) can be assigned a register—because every color has been assigned to a basic block that interferes with \( i \)—then delete \( X \) from the list and modify the interference graph.
   (b) Else, assign it a color that is not in its forbidden set.
   (c) Update the forbidden sets of the members of \( \text{INTERFERE} \) for \( X' \).

3. For each live range \( X' \) that is in \( \text{INTERFERE} \) for \( X \) do:
   (a) If the FORBIDDEN of \( X' \) is the set of all colors i.e., if no colors are available, SPLIT (\( X' \)).
   Procedure SPLIT breaks a live range into smaller live ranges with the intent of reducing the interference of \( X' \) it will be described next.

4. Repeat the above steps till all constrained live ranges are colored or till there is no color left to color any basic block.

The Algorithm (Contd.)

3. For each live range \( X' \) that is in \( \text{INTERFERE} \) for \( X \) do:
   (a) If the FORBIDDEN of \( X' \) is the set of all colors i.e., if no colors are available, SPLIT (\( X' \)).
   Procedure SPLIT breaks a live range into smaller live ranges with the intent of reducing the interference of \( X' \) it will be described next.

4. Repeat the above steps till all constrained live ranges are colored or till there is no color left to color any basic block.

The Idea Behind Splitting

- Splitting ensures that we break a live range up into increasingly smaller live ranges.
- The limit is of course when we are down to the size of a single basic block.
- The intuition is that we start out with coarse-grained interference graphs with few nodes.
- This makes the interference node degree possibly high.
- We increase the problem size via splitting on a need-to basis.
- This strategy lowers the interference.

The Splitting Strategy

A sketch of an algorithm for splitting:

1. Choose a split point.
   Note that we are guaranteed that \( X \) has at least one basic block \( i \) which can be assigned a color i.e., its forbidden set does not include all the colors. The earliest such in the order of control flow can be the split point.
2. Separate the live range \( X \) into \( X_1 \) and \( X_2 \) around the split point.
3. Update the sets \( \text{INTERFERE} \) for \( X_1 \) and \( X_2 \) and those for the live ranges that interfered with \( X \)
4. Recompute priorities and reprioritize the list.

Other bookkeeping activities to realize a safe implementation are also executed.

### Live Range Splitting Example

**Live Ranges:**
- a: BB1, BB2, BB3, BB4, BB5
- b: BB1, BB2, BB3, BB4, BB5, BB6
- c: BB2, BB3, BB4, BB5

Assume the number of physical registers = 2

```
= a
= b
= c
```

**New Live Ranges:**
- a: BB1, BB2, BB3, BB4, BB5
- b: BB6
- c: BB2, BB3, BB4, BB5

b and b2 are logically the same program variable
b2 is a renamed equivalent of b.
All nodes are now unconstrained.

### Interaction Between Allocation and Scheduling

- The allocator and the scheduler are typically patched together heuristically.
- Leads to the "phase ordering problem: Should allocation be done before scheduling or vice-versa?"
- Saving on spilling or "good allocation" is only indirectly connected to the actual execution time.
  Contrast with instruction scheduling.
- Factoring in register allocation into scheduling and solving the problem "globally" is a research issue.
• Basic block does not expose enough parallelism due to small num of inst.
• Need to look at more flexible regions
  – Trace scheduling, Superblock,…
• Scheduling more flexible regions implies using features such as speculation, code duplication, predication