Bab 1

Optimasi Kompilator Dasar

1.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep optimasi kompilator dan tujuannya
2. Menjelaskan dan mengidentifikasi basic blocks dalam intermediate code
3. Mengimplementasikan optimasi lokal: constant folding dan constant propagation
4. Mengimplementasikan dead code elimination
5. Memahami dasar-dasar data-flow analysis untuk optimasi global
6. Mengevaluasi efektivitas optimasi dengan membandingkan before/after
7. Membedakan machine-independent dan machine-specific optimizations

1.2 Pendahuluan

Optimasi kompilator adalah proses transformasi kode intermediate untuk meningkatkan kualitas kode yang dihasilkan tanpa mengubah semantik program. Menurut sumber dari Scribd OBE CSE Document:

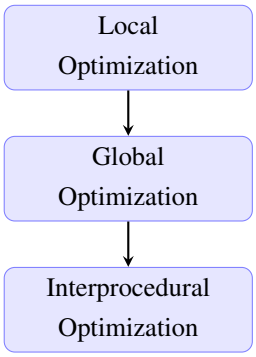
> "Perform machine-independent optimizations (basic block optimizations, data-flow analysis). Local and global optimization; data-flow analysis."**?**

Tujuan optimasi kompilator meliputi:

- **Meningkatkan Performa**: Mengurangi waktu eksekusi program
- **Mengurangi Ukuran Kode**: Menghasilkan executable yang lebih kecil
- **Mengurangi Konsumsi Memory**: Mengoptimasi penggunaan memory

- **Meningkatkan Efisiensi Energy**: Mengurangi konsumsi daya (penting untuk embedded systems)

Gambar 1.1 menunjukkan level-level optimasi.



Gambar 1.1: Level-level optimasi kompilator

Namun, optimasi harus dilakukan dengan hati-hati karena:

- Optimasi yang terlalu agresif dapat meningkatkan waktu kompilasi
- Beberapa optimasi dapat membuat kode lebih sulit di-debug
- Optimasi yang salah dapat mengubah semantik program (bug)

1.2.1 Prinsip Optimasi

Menurut Dragon Book**?**, optimasi harus mematuhi prinsip-prinsip berikut:

1. **Correctness**: Optimasi tidak boleh mengubah semantik program
2. **Benefit**: Optimasi harus memberikan manfaat yang signifikan
3. **Speed**: Proses optimasi tidak boleh terlalu lambat
4. **Simplicity**: Optimasi harus mudah diimplementasikan dan di-maintain

1.2.2 Level Optimasi

Optimasi dapat dikategorikan berdasarkan scope-nya:

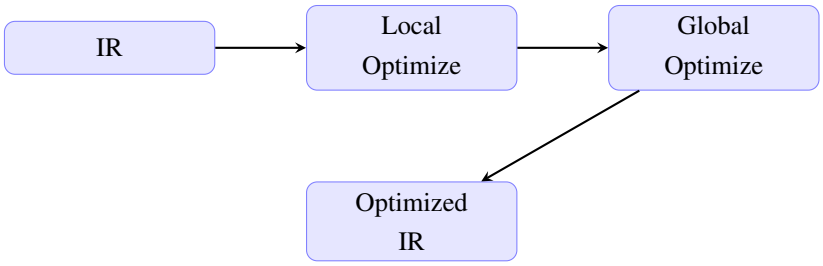
- **Local Optimization**: Optimasi dalam satu basic block
  - Constant folding
  - Constant propagation

  - Algebraic simplification
  - Strength reduction

- **Global Optimization**: Optimasi lintas basic blocks
  - Common subexpression elimination
  - Loop optimization
  - Dead code elimination (global)
  - Constant propagation (global)

- **Interprocedural Optimization**: Optimasi lintas fungsi/prosedur
  - Inlining
  - Interprocedural constant propagation
  - Whole-program optimization

Gambar 1.2 menunjukkan pipeline optimasi dalam kompilator.



Gambar 1.2: Pipeline optimasi kompilator

1.3 Basic Blocks

Basic block adalah fondasi untuk banyak optimasi kompilator. Menurut University of Michigan[1], basic block didefinisikan sebagai:

> "A basic block is a straight-line sequence of code with no jumps in except at the entry, and no jumps out except at the exit. Once execution enters it, all instructions execute sequentially."

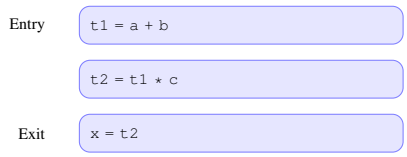
[1] https://web.eecs.umich.edu/~weimerw/2015-4610/ca1/ca1.html

1.3.1 Karakteristik Basic Block

Sebuah basic block memiliki karakteristik berikut:

1. **Single Entry Point**: Hanya ada satu titik masuk (entry point)
2. **Single Exit Point**: Hanya ada satu titik keluar (exit point)
3. **Sequential Execution**: Semua instruksi dieksekusi secara berurutan tanpa branching
4. **No Internal Control Flow**: Tidak ada jump, branch, atau call di tengah-tengah block

Gambar 1.3 menunjukkan contoh basic block.



Gambar 1.3: Contoh basic block

1.3.2 Identifikasi Basic Blocks

Algoritma untuk mengidentifikasi basic blocks dalam intermediate code:

1. **Leader Identification**: Tentukan leader (instruksi pertama dalam basic block)
   - Instruksi pertama dalam program adalah leader
   - Instruksi yang merupakan target dari jump/branch adalah leader
   - Instruksi setelah jump/branch/call adalah leader
2. **Block Construction**: Untuk setiap leader, buat basic block yang berisi:
   - Leader instruction
   - Semua instruksi berikutnya hingga menemukan leader berikutnya atau instruksi control flow

1.3.3 Contoh Identifikasi Basic Block

Perhatikan contoh three-address code berikut:

```
L1: t1 = a + b
    t2 = c * d
    t3 = t1 + t2
    if t3 > 0 goto L2
```

```
    t4 = t1 - t2
    goto L3
L2: t5 = t1 * t2
L3: t6 = t5 + 1
    return t6
```

Basic blocks yang diidentifikasi:
Block 1 (L1):

```
t1 = a + b
t2 = c * d
t3 = t1 + t2
if t3 > 0 goto L2
```

Block 2 (L2):

```
t5 = t1 * t2
```

Block 3 (setelah goto L3):

```
t4 = t1 - t2
goto L3
```

Block 4 (L3):

```
t6 = t5 + 1
return t6
```

1.3.4 Control Flow Graph (CFG)

Control Flow Graph adalah representasi grafis dari alur kontrol program. Menurut Wikipedia[2]:

> "A control-flow graph (CFG) is a representation of a function where each node is a basic block, and edges represent possible flow of control from one block to another."

CFG membantu dalam:

- Memahami struktur program
- Melakukan data-flow analysis
- Mengidentifikasi loop dan struktur kontrol lainnya
- Mengoptimasi lintas basic blocks

1.4 Constant Folding

Constant folding adalah optimasi yang mengganti ekspresi yang hanya melibatkan konstanta dengan hasil komputasinya pada waktu kompilasi. Menurut GeeksforGeeks[3]:

[2] https://en.wikipedia.org/wiki/Control-flow_graph
[3] https://www.geeksforgeeks.org/compiler-design/constant-folding/

> "Constant folding replaces expressions involving only constants (literals) with their computed result at compile time, rather than at runtime. Example: turning '5 + 7 * 2' into '19' in the generated code."

1.4.1 Contoh Constant Folding

Before optimization:

```
t1 = 5 + 7
t2 = t1 * 2
t3 = 10 / 2
x = t2 + t3
```

After constant folding:

```
t1 = 12          // 5 + 7 = 12
t2 = 24          // 12 * 2 = 24
t3 = 5           // 10 / 2 = 5
x = 29           // 24 + 5 = 29
```

Atau bahkan lebih optimal:

```
x = 29           // Semua konstanta di-fold menjadi satu nilai
```

1.4.2 Implementasi Constant Folding

Algoritma constant folding untuk three-address code:

1. Untuk setiap instruksi dalam basic block:
   - Jika kedua operan adalah konstanta, evaluasi ekspresi
   - Ganti instruksi dengan assignment konstanta hasil
2. Ulangi hingga tidak ada lagi perubahan (iterasi mungkin diperlukan jika ada dependensi)

1.4.3 Contoh Implementasi dalam C++

Berikut adalah contoh sederhana implementasi constant folding:

Listing 1.1: Contoh implementasi constant folding

```
1 struct Instruction {
2     string op;      // operator: +, -, *, /, =
3     string result;  // variabel hasil
4     string arg1;    // operand pertama
5     string arg2;    // operand kedua (optional)
6 };
7
```

```cpp
bool isConstant(const string& var,
                const map<string, int>& constants) {
    return constants.find(var) != constants.end();
}

int evaluateConstant(int val1, int val2, const string& op) {
    if (op == "+") return val1 + val2;
    if (op == "-") return val1 - val2;
    if (op == "*") return val1 * val2;
    if (op == "/") return val2 != 0 ? val1 / val2 : 0;
    return 0;
}

void constantFolding(vector<Instruction>& instructions) {
    map<string, int> constants;

    for (auto& inst : instructions) {
        if (inst.op == "=" && isNumeric(inst.arg1)) {
            // Assignment konstanta langsung
            constants[inst.result] = stoi(inst.arg1);
        } else if (inst.op != "=") {
            // Operasi biner
            if (isConstant(inst.arg1, constants) &&
                isConstant(inst.arg2, constants)) {
                int val1 = constants[inst.arg1];
                int val2 = constants[inst.arg2];
                int result = evaluateConstant(val1, val2, inst.op);
                constants[inst.result] = result;
                // Ganti instruksi dengan assignment konstanta
                inst.op = "=";
                inst.arg1 = to_string(result);
                inst.arg2 = "";
            }
        }
    }
}
```

1.5 Constant Propagation

Constant propagation adalah optimasi yang mengganti penggunaan variabel yang diketahui bernilai konstan dengan nilai konstanta tersebut. Menurut GeeksforGeeks[4]:

> "Constant propagation replaces variables known to be constant with their constant values, then combined with constant folding to simplify more complex expressions."

[4] https://www.geeksforgeeks.org/machine-independent-code-optimization-in-compiler-design/

1.5.1 Contoh Constant Propagation

Before optimization:

```
x = 10
y = 20
t1 = x + 5      // x adalah konstanta 10
t2 = y * 2      // y adalah konstanta 20
z = t1 + t2
```

After constant propagation:

```
x = 10
y = 20
t1 = 10 + 5     // x diganti dengan 10
t2 = 20 * 2     // y diganti dengan 20
z = t1 + t2
```

After constant folding (kombinasi):

```
x = 10
y = 20
t1 = 15         // 10 + 5 = 15
t2 = 40         // 20 * 2 = 40
z = 55          // 15 + 40 = 55
```

1.5.2 Local vs Global Constant Propagation

Local Constant Propagation:

- Hanya dalam satu basic block
- Lebih sederhana, tidak memerlukan data-flow analysis
- Dapat dilakukan bersamaan dengan constant folding

Global Constant Propagation:

- Lintas basic blocks
- Memerlukan data-flow analysis (reaching definitions)
- Lebih kompleks tetapi lebih powerful
- Harus mempertimbangkan multiple paths dalam CFG

1.5.3 Implementasi Local Constant Propagation

Algoritma untuk local constant propagation:

1. Scan basic block dari atas ke bawah
2. Maintain map variabel $\rightarrow$ nilai konstanta
3. Untuk setiap instruksi:
    - Jika assignment konstanta: update map
    - Jika penggunaan variabel: ganti dengan konstanta jika tersedia
    - Jika assignment dari variabel non-konstanta: hapus dari map (variabel tidak lagi konstanta)

1.6 Dead Code Elimination

Dead code elimination adalah optimasi yang menghapus kode yang tidak memiliki efek pada perilaku program yang dapat diamati. Menurut GeeksforGeeks[5]:

> "Dead code elimination removes code that has no effect on the program's observable behavior. Two main kinds: unreachable code (code that can never be executed) and assignment to variables never used (where a variable's value is computed but never read before being overwritten)."

1.6.1 Jenis Dead Code

1. Unreachable Code Kode yang tidak pernah dapat dieksekusi karena tidak ada path yang mencapainya.

Contoh:

```
x = 10
return x
y = 20      // Dead code - tidak pernah dieksekusi
z = y + 5   // Dead code
```

2. Dead Assignments Assignment ke variabel yang nilainya tidak pernah digunakan sebelum di-overwrite.

Contoh:

```
x = 10
x = 20      // Assignment pertama adalah dead code
y = x       // Hanya nilai kedua yang digunakan
```

[5] https://www.geeksforgeeks.org/dead-code-elimination/

1.6.2 Unreachable Code Elimination

Algoritma untuk menghapus unreachable code:

1. Bangun Control Flow Graph (CFG)
2. Lakukan traversal dari entry block (misalnya DFS atau BFS)
3. Mark semua basic block yang dapat dicapai
4. Hapus semua basic block yang tidak ter-mark

1.6.3 Dead Assignment Elimination

Algoritma untuk menghapus dead assignments (menggunakan live-variable analysis):

1. Lakukan live-variable analysis untuk menentukan variabel mana yang "live" di setiap titik
2. Variabel dikatakan "live" jika nilainya mungkin digunakan di masa depan
3. Untuk setiap assignment `x = ...`:
    - Jika `x` tidak live setelah assignment, assignment tersebut adalah dead code
    - Hapus assignment tersebut

1.6.4 Contoh Dead Code Elimination

Before optimization:

```
x = 10
y = 20
t1 = x + y
t2 = t1 * 2
x = t2
t3 = 5 + 3      // Dead: t3 tidak pernah digunakan
t4 = t3 - 2    // Dead: t4 tidak pernah digunakan
return x
```

After dead code elimination:

```
x = 10
y = 20
t1 = x + y
t2 = t1 * 2
x = t2
return x
```

1.6.5 Implementasi Dead Code Elimination

Berikut adalah contoh implementasi sederhana untuk dead assignment elimination:

Listing 1.2: Contoh implementasi dead code elimination

```
set<string> computeLiveVariables(const vector<Instruction>& insts) {
    set<string> live;

    // Scan dari bawah ke atas
    for (int i = insts.size() - 1; i >= 0; i--) {
        const auto& inst = insts[i];

        // Variabel yang digunakan adalah live
        if (!inst.arg1.empty() && !isConstant(inst.arg1)) {
            live.insert(inst.arg1);
        }
        if (!inst.arg2.empty() && !isConstant(inst.arg2)) {
            live.insert(inst.arg2);
        }

        // Variabel yang di-assign tidak lagi live setelah assignment
        // (kecuali jika digunakan di sisi kanan)
        if (live.find(inst.result) != live.end()) {
            live.erase(inst.result);
        }
    }

    return live;
}

vector<Instruction> eliminateDeadCode(
    const vector<Instruction>& instructions) {

    set<string> live = computeLiveVariables(instructions);
    vector<Instruction> optimized;

    for (const auto& inst : instructions) {
        // Skip jika assignment ke variabel yang tidak live
        if (inst.op == "=" && live.find(inst.result) == live.end()) {
            continue; // Dead assignment
        }

        optimized.push_back(inst);

        // Update live set
        if (!inst.arg1.empty()) live.insert(inst.arg1);
        if (!inst.arg2.empty()) live.insert(inst.arg2);
        live.erase(inst.result);
    }

    return optimized;
}
```

1.7 Data-Flow Analysis Dasar

Data-flow analysis adalah teknik untuk menghitung informasi tentang kemungkinan perilaku program. Menurut GeeksforGeeks[6]:

> "Data-flow analysis is a technique to compute information about possible program behaviors (how definitions, uses of variables, expressions, etc. propagate through the code). Usually done on a CFG."

Data-flow analysis adalah fondasi untuk optimasi global yang lebih advanced.

1.7.1 Konsep Dasar Data-Flow Analysis

Data-flow analysis bekerja dengan:

1. **Domain**: Himpunan informasi yang ingin dihitung
   - Live variables: set variabel yang live
   - Reaching definitions: set definisi yang mencapai suatu titik
   - Available expressions: set ekspresi yang sudah dihitung
2. **Transfer Function**: Bagaimana informasi berubah setelah eksekusi instruksi
3. **Meet/Join Operation**: Bagaimana menggabungkan informasi dari multiple paths
4. **Fixpoint Iteration**: Iterasi hingga mencapai fixpoint (tidak ada perubahan)

1.7.2 Live Variable Analysis

Live variable analysis menentukan variabel mana yang "live" (nilainya mungkin digunakan) di setiap titik program.

Definisi:

- Variabel `v` dikatakan **live** di titik `p` jika ada path dari `p` ke penggunaan `v` tanpa assignment ke `v` di antara keduanya
- Variabel `v` dikatakan **dead** jika tidak live

Algoritma (Backward Analysis):

1. Inisialisasi: `LIVE[exit] = {}`
2. Untuk setiap basic block `B` (dari exit ke entry):

[6] https://www.geeksforgeeks.org/data-flow-analysis-compiler/

- `LIVE[B] = UNION(LIVE[successors])`
- `LIVE[B] = LIVE[B] - DEF[B] + USE[B]`
- `DEF[B]`: variabel yang didefinisikan di B
- `USE[B]`: variabel yang digunakan di B

3. Ulangi hingga fixpoint

1.7.3 Reaching Definitions

Reaching definitions analysis menentukan definisi variabel mana yang "mencapai" suatu titik program.

Definisi: Definisi `d` dikatakan **reach** titik `p` jika ada path dari `d` ke `p` tanpa definisi lain untuk variabel yang sama.

Kegunaan:

- Constant propagation (global)
- Deteksi penggunaan variabel sebelum inisialisasi
- Optimasi lainnya

1.7.4 Available Expressions

Available expressions analysis menentukan ekspresi mana yang sudah dihitung dan masih valid (operand-nya belum berubah).

Kegunaan:

- Common subexpression elimination
- Optimasi lainnya

1.8 Kombinasi Optimasi

Dalam praktik, optimasi biasanya dilakukan dalam beberapa pass dan saling berinteraksi:

1.8.1 Order of Optimization

Urutan optimasi yang umum:

1. **Constant Folding & Propagation**: Simplifikasi ekspresi konstanta
2. **Dead Code Elimination**: Hapus kode yang tidak digunakan

3. **Common Subexpression Elimination**: Hapus komputasi duplikat
4. **Loop Optimizations**: Optimasi khusus untuk loop
5. **Register Allocation**: Alokasi register yang efisien

1.8.2 Iterative Optimization

Optimizer biasanya menjalankan beberapa pass hingga tidak ada lagi perubahan:

```
do {
    changed = false
    changed |= constantFolding()
    changed |= constantPropagation()
    changed |= deadCodeElimination()
    // ... optimasi lainnya
} while (changed)
```

1.9 Evaluasi Efektivitas Optimasi

Setelah mengimplementasikan optimasi, penting untuk mengevaluasi efektivitasnya.

1.9.1 Metrics untuk Evaluasi

1. Code Size

- Ukuran executable sebelum dan sesudah optimasi
- Jumlah instruksi dalam intermediate code
- Ukuran object files

2. Execution Time

- Waktu eksekusi program dengan benchmark
- Profiling untuk mengidentifikasi bottleneck
- Perbandingan before/after

3. Memory Usage

- Peak memory consumption
- Stack usage
- Heap allocation patterns

4. Compilation Time

- Waktu yang dibutuhkan untuk kompilasi
- Trade-off antara waktu kompilasi dan kualitas optimasi

1.9.2 Benchmarking

Langkah-langkah untuk benchmarking:

1. **Prepare Test Cases**: Siapkan berbagai test case (small, medium, large programs)
2. **Baseline Measurement**: Ukur metrik sebelum optimasi
3. **Optimized Measurement**: Ukur metrik setelah optimasi
4. **Compare Results**: Bandingkan dan hitung improvement percentage
5. **Verify Correctness**: Pastikan program masih menghasilkan output yang benar

1.9.3 Contoh Evaluasi

Berikut adalah contoh format laporan evaluasi:

Metric	Before	After	Improvement
Code Size (bytes)	1024	768	25% reduction
Execution Time (ms)	100	75	25% faster
Instruction Count	150	110	26.7% reduction
Compilation Time (s)	2.5	3.1	24% slower

Tabel 1.1: Contoh hasil evaluasi optimasi

1.10 Implementasi Praktis

Dalam bagian ini, kita akan melihat contoh implementasi optimizer sederhana yang menggabungkan beberapa optimasi dasar.

1.10.1 Struktur Optimizer

Listing 1.3: Struktur dasar optimizer

```
class Optimizer {
private:
    vector<BasicBlock> basicBlocks;
    ControlFlowGraph cfg;

```

```
public:
    // Identifikasi basic blocks
    void identifyBasicBlocks(const vector<Instruction>& insts);

    // Optimasi lokal dalam basic block
    void optimizeBasicBlock(BasicBlock& block);

    // Optimasi global lintas basic blocks
    void optimizeGlobal();

    // Kombinasi semua optimasi
    vector<Instruction> optimize(const vector<Instruction>& insts);
};

vector<Instruction> Optimizer::optimize(
    const vector<Instruction>& instructions) {

    // 1. Identifikasi basic blocks
    identifyBasicBlocks(instructions);

    // 2. Optimasi lokal untuk setiap basic block
    for (auto& block : basicBlocks) {
        optimizeBasicBlock(block);
    }

    // 3. Optimasi global
    optimizeGlobal();

    // 4. Reconstruct instructions dari basic blocks
    return reconstructInstructions();
}

void Optimizer::optimizeBasicBlock(BasicBlock& block) {
    bool changed = true;

    while (changed) {
        changed = false;

        // Constant folding
        changed |= constantFolding(block);

        // Constant propagation
        changed |= constantPropagation(block);

        // Dead code elimination
        changed |= deadCodeElimination(block);
    }
}
```

1.11 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Optimasi kompilator bertujuan meningkatkan kualitas kode tanpa mengubah semantik
2. Basic blocks adalah unit fundamental untuk optimasi lokal
3. Constant folding dan constant propagation adalah optimasi dasar yang efektif
4. Dead code elimination menghapus kode yang tidak berguna
5. Data-flow analysis adalah fondasi untuk optimasi global
6. Evaluasi efektivitas optimasi penting untuk memastikan optimasi memberikan manfaat

Optimasi kompilator adalah bidang yang luas dan kompleks. Bab ini memberikan dasar-dasar optimasi lokal. Untuk optimasi yang lebih advanced seperti loop optimization, interprocedural optimization, dan machine-specific optimization, diperlukan pemahaman yang lebih mendalam tentang data-flow analysis dan teknik optimasi lainnya.

1.12 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang optimasi kompilator, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* **?** - Bab 9: Machine-Independent Optimizations
- **Engineering a Compiler**: Cooper & Torczon (2011) **?** - Bab 8: Introduction to Optimization, Bab 9: Data-Flow Analysis
- **GeeksforGeeks**: Tutorial tentang berbagai optimasi kompilator
  - Constant Folding: https://www.geeksforgeeks.org/compiler-design/constant-folding/
  - Dead Code Elimination: https://www.geeksforgeeks.org/dead-code-elimination/
  - Data-Flow Analysis: https://www.geeksforgeeks.org/data-flow-analysis-compiler/
- **University of Michigan**: Course materials tentang compiler optimization [7]
- **LLVM Documentation**: Advanced optimization techniques [8]

[7] https://web.eecs.umich.edu/~weimerw/2015-4610/ca1/ca1.html
[8] https://llvm.org/docs/Passes.html