Bab 13

Register Allocation dan Optimization

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 5.3:** Mengoptimasi penggunaan register dan meminimalkan memory access

13.1 Alokasi Register: Strategi dan Kompleksitas

Karena jumlah register fisik dalam CPU sangat terbatas, kompilator harus memutuskan variabel mana yang layak menempati register dan mana yang harus dipindahkan (*spilled*) ke RAM.

13.1.1 Graph Coloring

Masalah alokasi register dapat dimodelkan sebagai pewarnaan graf (*Graph Coloring*).

- Node: Variabel yang aktif (*live variables*).
- Edge: Saling berpotongan (*interfere*), artinya dua variabel hidup di waktu yang sama.
- Warna: Jumlah register fisik yang tersedia.

13.1.2 Linear Scan

Untuk kompilasi yang cepat (*JIT compilers*), algoritma *Linear Scan* digunakan karena lebih sederhana dibanding *graph coloring* namun tetap memberikan hasil yang memadai.

13.2 Interference Graph

13.2.1 Graph Coloring

Register allocation sebagai graph coloring:

- **Nodes**: Variabel/temporaries
- **Edges**: Interference (variables live simultaneously)
- **Colors**: Register assignments
- **Spilling**: Variables yang tidak dapat diwarnai

13.2.2 Interference Graph Construction

```c
typedef struct {
    int var_id;
    char *var_name;
    int start_point;
    int end_point;
} LiveRange;

typedef struct {
    int num_vars;
    bool **adjacency;  // Interference matrix
    LiveRange *ranges;
} InterferenceGraph;

InterferenceGraph* build_interference_graph(BasicBlock *block) {
    // Calculate live ranges
    LiveRange *ranges = calculate_live_ranges(block);

    // Build interference graph
    InterferenceGraph *graph = create_graph(num_vars);

    for (int i = 0; i < num_vars; i++) {
        for (int j = i + 1; j < num_vars; j++) {
            if (ranges_interfere(ranges[i], ranges[j])) {
                add_interference_edge(graph, i, j);
            }
        }
    }

    return graph;
}
```

13.3 Graph Coloring Algorithm

13.3.1 Simplified Graph Coloring

```c
typedef struct {
    int var_id;
    int color;          // -1 = spilled, 0+ = register number
    int degree;         // Number of neighbors
    bool removed;
} GraphNode;

int graph_coloring(InterferenceGraph *graph, int num_registers) {
    GraphNode *nodes = initialize_nodes(graph);
    int spilled_count = 0;

    // Simplify phase
    while (has_uncolored_nodes(nodes)) {
        // Find node with degree < num_registers
        int node = find_low_degree_node(nodes, num_registers);

        if (node != -1) {
            // Remove from graph
            remove_node(nodes, node);
        } else {
            // Spill a node
            int spill_node = select_spill_node(nodes);
            nodes[spill_node].color = -1;   // Mark as spilled
            remove_node(nodes, spill_node);
            spilled_count++;
        }
    }

    // Select phase - assign colors
    assign_colors(nodes, num_registers);

    return spilled_count;
}
```

13.4 Linear Scan Allocation

13.4.1 Linear Scan Algorithm

Alokasi register yang lebih efisien:

```c
typedef struct {
    int var_id;
    int start;
    int end;
    int reg;            // -1 if not allocated
    bool active;
} Interval;

void linear_scan(Interval *intervals, int count, int num_registers) {
    // Sort intervals by start point
    sort_intervals_by_start(intervals, count);

    Interval *active[num_registers];
    int active_count = 0;

    for (int i = 0; i < count; i++) {
        // Expire old intervals
        expire_old_intervals(intervals[i].start, active, &active_count);

        if (active_count < num_registers) {
            // Allocate register
            intervals[i].reg = find_free_register(active, active_count);
            add_to_active(&intervals[i], active, &active_count);
        } else {
            // Spill
            int spill_index = select_spill_candidate(active, active_count
↪ );
            spill_interval(active[spill_index]);
            intervals[i].reg = active[spill_index]->reg;
            active[spill_index] = &intervals[i];
        }
    }
}
```

13.5 Spilling Strategies

13.5.1 Spill Cost Analysis

Menentukan variabel yang akan di-spill:

```c
typedef struct {
    int var_id;
    float spill_cost;  // Lower = better to spill
    int memory_accesses;
    int loop_depth;
    int use_count;
} SpillCandidate;

float calculate_spill_cost(SpillCandidate *candidate) {
    // Consider multiple factors
    float cost = 0.0;

    // More memory accesses = higher cost (don't want to spill)
    cost += candidate->memory_accesses * 10.0;

    // Deeper in loops = higher cost
    cost += candidate->loop_depth * 5.0;

    // More uses = higher cost
    cost += candidate->use_count * 2.0;

    // Normalize by live range length
    int range_length = candidate->end - candidate->start;
    return cost / range_length;
}
```

```

\subsection{Spill Code Generation}

\begin{lstlisting}[language=C]
void generate_spill_code(Instruction *instructions, int *count,
                        SpillCandidate *spilled_vars, int num_spilled) {
    for (int i = 0; i < num_spilled; i++) {
        int var_id = spilled_vars[i].var_id;

        // Insert load before each use
        for (int j = 0; j < *count; j++) {
            if (uses_variable(&instructions[j], var_id)) {
                insert_load_before(&instructions[j], var_id, count);
                j++;  // Skip inserted instruction
            }
        }

        // Insert store after each definition
        for (int j = 0; j < *count; j++) {
            if (defines_variable(&instructions[j], var_id)) {
                insert_store_after(&instructions[j], var_id, count);
                j++;  // Skip inserted instruction
            }
        }
    }
}
```

13.6 Coalescing

13.6.1 Copy Coalescing

Menggabungkan variabel yang di-copy:

```
// Before coalescing:
t1 = x
y = t1
z = t1

// After coalescing (t1 eliminated):
y = x
z = x

bool can_coalesce(InterferenceGraph *graph, int var1, int var2) {
    // Check if variables interfere
    if (have_interference(graph, var1, var2)) {
        return false;
    }

    // Check if coalescing would create too high degree
    int combined_degree = calculate_combined_degree(graph, var1, var2);
    if (combined_degree >= num_registers) {
        return false;
    }

    return true;
}

void coalesce_variables(InterferenceGraph *graph, int var1, int var2) {
    // Merge var2 into var1
    merge_nodes(graph, var1, var2);
    update_interference_edges(graph, var1);
}
```

13.7 Advanced Optimizations

13.7.1 Register Renaming

Menghilangkan false dependencies:

```
// Before renaming:
t1 = a + b
t2 = t1 * c
t1 = d + e    // False dependency on previous t1
t3 = t1 * f

// After renaming:
t1 = a + b
t2 = t1 * c
t3 = d + e    // No false dependency
t4 = t3 * f

void rename_registers(BasicBlock *block) {
    int next_temp = 0;
    int register_map[MAX_VARIABLES];

    for (int i = 0; i < block->instruction_count; i++) {
        Instruction *inst = &block->instructions[i];

        // Rename destination
        if (inst->result) {
            int new_reg = next_temp++;
            register_map[inst->result] = new_reg;
            inst->result = new_reg;
        }

        // Update source operands
        if (inst->arg1 && register_map[inst->arg1]) {
            inst->arg1 = register_map[inst->arg1];
        }
        if (inst->arg2 && register_map[inst->arg2]) {
            inst->arg2 = register_map[inst->arg2];
        }
    }
}

\subsection{Loop Optimization}
```

```c

\begin{lstlisting}[language=C]
void optimize_loop_registers(Loop *loop) {
    // Allocate loop invariants to registers
    identify_loop_invariants(loop);

    // Keep frequently used loop variables in registers
    analyze_loop_variable_usage(loop);

    // Minimize register pressure in loop body
    reduce_loop_register_pressure(loop);

    // Prefer registers for induction variables
    allocate_induction_variables(loop);
}
```

Aktivitas Pembelajaran

1. **Interference Graph**: Implementasikan interference graph construction.
2. **Graph Coloring**: Bangun graph coloring register allocator.
3. **Linear Scan**: Implementasikan linear scan allocation algorithm.
4. **Spilling**: Desain spill cost analysis dan spill code generation.
5. **Coalescing**: Implementasikan copy coalescing optimization.

Latihan dan Refleksi

1. Bangun interference graph untuk potongan kode dengan multiple variables!
2. Implementasikan graph coloring dengan backtracking untuk optimal solution!
3. Analisis spill cost untuk berbagai variabel dalam nested loops!
4. Implementasikan linear scan dengan heuristic improvements!
5. Optimasi register allocation untuk loop-intensive code!
6. **Refleksi**: Bagaimana register allocation mempengaruhi performance generated code?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 5.3

A. Pilihan Ganda

1. Interference graph edge menunjukkan:
   (a) Variables yang sama
   (b) Variables yang hidup bersamaan
   (c) Variables yang di-copy
   (d) Variables yang di-spill
2. Linear scan allocation memiliki complexity:
   (a) O(n)
   (b) O(n log n)
   (c) $O(n^2)$
   (d) $O(n^3)$
3. Spilling dilakukan ketika:
   (a) Register penuh
   (b) Memory penuh
   (c) Graph tidak bisa diwarnai
   (d) Variabel tidak digunakan

B. Essay

1. Jelaskan complete register allocation pipeline dengan interference graph and graph coloring!
2. Implementasikan register allocator dengan linear scan algorithm and spill handling!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat mengoptimasi penggunaan register dan meminimalkan memory access

☐ Saya dapat membangun interference graph untuk register allocation

☐ Saya dapat mengimplementasikan graph coloring algorithm

☐ Saya dapat melakukan linear scan register allocation

☐ Saya dapat mengimplementasikan spill strategies

☐ Saya dapat melakukan register coalescing dan renaming

Rangkuman

Bab ini membahas register allocation dan optimization, termasuk interference graph, graph coloring, linear scan allocation, spilling strategies, dan advanced optimizations. Mahasiswa belajar mengoptimasi penggunaan register untuk performance maksimal.

Poin Kunci:

- Register allocation memetakan variabel unlimited ke register terbatas
- Interference graph modeling conflicts antar variabel
- Graph coloring adalah NP-complete problem
- Linear scan memberikan heuristic yang efisien
- Spilling menangani kasus ketika register tidak cukup
- Coalescing dan renaming mengoptimasi register usage

Kata Kunci: ***Register Allocation***, ***Interference Graph***, ***Graph Coloring***, ***Linear Scan***, ***Spilling***, ***Coalescing***, ***Register Renaming***