Bab 16

Performance Evaluation dan Benchmarking

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 6.2:** Mengevaluasi kinerja compiler dan optimasi

16.1 Studi Kasus: Proyek Compiler Subset C

Sebagai bentuk evaluasi kinerja dan integrasi seluruh fase, kita akan meninjau spesifikasi proyek ***Subset C*** yang telah kita bangun secara bertahap.

16.1.1 Spesifikasi Grammar dan AST

Proyek ini mengimplementasikan grammar *top-down* untuk mengakomodasi *recursive descent parser* serta grammar *bottom-up* untuk *Bison*. Token yang didukung meliputi tipe data `int`/`float`, kontrol `if`/`while`, dan fungsi `print`.

16.1.2 Manajemen Runtime dan Memori

Pada fase awal, proyek ini menggunakan alokasi statis untuk variabel sederhana. Saat fungsi ditambahkan, proyek mengadopsi *activation record* standar x86-64 untuk memastikan kompatibilitas dengan *runtime C library* (`libc`).

16.1.3 Analisis Kinerja

Mahasiswa diharapkan melakukan *benchmarking* terhadap kode yang dihasilkan, membandingkan antara kode tanpa optimasi dengan kode yang telah melalui fase *constant folding* dan *dead code elimination*.

16.2 Benchmarking Methodology

16.2.1 Benchmark Design

Prinsip benchmark yang baik:

- **Reproducibility**: Hasil dapat direproduksi
- **Fairness**: Perbandingan yang adil
- **Representative**: Mewakili kasus nyata
- **Statistical significance**: Hasil statistik valid

16.2.2 Test Suite

```
// Benchmark test suite structure
typedef struct {
    char *name;
    char *description;
    char *input_file;
    int expected_time_ms;
    int expected_memory_kb;
} BenchmarkCase;

BenchmarkCase benchmarks[] = {
    {"small_file", "Small C file", "small.c", 100, 1024},
    {"medium_file", "Medium C file", "medium.c", 500, 4096},
    {"large_file", "Large C file", "large.c", 2000, 16384},
    {"complex_template", "Complex template", "template.cpp", 5000, 32768}
};
```

16.3 Measurement Tools

16.3.1 Time Measurement

```
#include <time.h>
#include <sys/time.h>

// High-resolution timer
double get_time_ms() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec * 1000.0 + ts.tv_nsec / 1000000.0;
}

// Measure compilation time
double measure_compilation_time(char *command) {
    double start = get_time_ms();

```

```
    int result = system(command);

    double end = get_time_ms();
    return end - start;
}
```

16.3.2 Memory Measurement

```
#include <sys/resource.h>
#include <unistd.h>

// Measure peak memory usage
size_t measure_peak_memory() {
    struct rusage usage;
    getrusage(RUSAGE_CHILDREN, &usage);
    return usage.ru_maxrss;  // Peak resident set size
}

// Measure current memory usage
size_t measure_current_memory() {
    FILE *status = fopen("/proc/self/status", "r");
    if (!status) return 0;

    char line[256];
    size_t memory = 0;

    while (fgets(line, sizeof(line), status)) {
        if (strncmp(line, "VmRSS:", 6) == 0) {
            sscanf(line + 7, "%zu", &memory);
            memory *= 1024;  // Convert KB to bytes
            break;
        }
    }

    fclose(status);
    return memory;
}
```

16.4 Compiler Comparison

16.4.1 Compiler Matrix

Compiler	Versi	Speed	Size	Memory
GCC	11.2	Fast	Medium	Medium
Clang	14.0	Fastest	Small	Low
MSVC	19.3	Medium	Large	High
ICC	2021.1	Fast	Medium	Medium

Tabel 16.1: Perbandingan Compiler Populer

16.4.2 Optimization Levels

```
# Benchmark different optimization levels
for level in O0 O1 O2 O3 Os; do
    echo "Testing with -$level optimization"
    time gcc -$level test.c -o test_$level
    ./test_$level
    echo "Size: $(stat -c%s test_$level)"
done
```

16.5 Performance Analysis

16.5.1 Profiling Results

```
# Profile with gprof
gcc -pg -O2 test.c -o test_profile
./test_profile
gprof test_profile gmon.out > profile_report.txt

# Profile with perf
perf stat -e cycles,instructions,cache-misses ./test_program

# Profile with valgrind
valgrind --tool=callgrind ./test_program
```

16.5.2 Statistical Analysis

```
import statistics
import numpy as np

def analyze_benchmark_results(times):
    """Analyze benchmark results statistically"""
    mean = np.mean(times)
    std_dev = np.std(times)
    median = np.median(times)

    # Remove outliers (beyond 2 standard deviations)
    filtered = [t for t in times if abs(t - mean) <= 2 * std_dev]

    filtered_mean = np.mean(filtered)
    filtered_std = np.std(filtered)

    return {
        'raw_mean': mean,
        'raw_std': std_dev,
        'filtered_mean': filtered_mean,
        'filtered_std': filtered_std,
        'median': median,
        'sample_size': len(times),
        'outliers_removed': len(times) - len(filtered)
    }
```

16.6 Optimization Impact

16.6.1 Optimization Techniques

Teknik	Speedup	Trade-off
Inline functions	1.2x - 2x	Code size increase
Loop unrolling	1.5x - 3x	Code size increase
Vectorization	4x - 8x	Limited to data parallel
Constant folding	1.1x - 1.3x	Compile time increase
Dead code elimination	1.05x - 1.2x	Compile time increase

Tabel 16.2: Impact Optimasi pada Performance

16.6.2 Case Studies

```
// Case study: Parser optimization
// Before optimization
int parse_slow(char *input) {
    // Multiple function calls
    token_t token1 = get_next_token(input);
    token_t token2 = get_next_token(token1->remaining);
    token_t token3 = get_next_token(token2->remaining);
    return process_tokens(token1, token2, token3);
}

// After optimization
int parse_fast(char *input) {
    // Inlined token processing
    token_t tokens[3];
    int count = 0;

    while (*input && count < 3) {
        tokens[count++] = get_next_token_inline(&input);
    }

    return process_tokens_inline(tokens[0], tokens[1], tokens[2]);
}
```

16.7 Benchmark Automation

16.7.1 Automated Testing

```
#!/usr/bin/env python3
import subprocess
import json
```

```python
import time

def run_benchmark(compiler, source_file, iterations=10):
    """Run automated benchmark"""
    results = []

    for i in range(iterations):
        start_time = time.time()

        # Compile
        compile_cmd = f"{compiler} -O2 {source_file} -o benchmark_test"
        subprocess.run(compile_cmd, shell=True, capture_output=True)

        # Execute
        start_exec = time.time()
        subprocess.run("./benchmark_test", shell=True, capture_output=
↪ True)
        end_exec = time.time()

        # Measure memory
        memory_usage = measure_memory_usage()

        results.append({
            'iteration': i,
            'compile_time': start_exec - start_time,
            'exec_time': end_exec - start_exec,
            'total_time': end_exec - start_time,
            'memory_kb': memory_usage
        })

    return results

def analyze_results(results):
    """Analyze benchmark results"""
    compile_times = [r['compile_time'] for r in results]
    exec_times = [r['exec_time'] for r in results]
    memory_usage = [r['memory_kb'] for r in results]

    return {
        'compile_stats': analyze_benchmark_results(compile_times),
        'exec_stats': analyze_benchmark_results(exec_times),
        'memory_stats': analyze_benchmark_results(memory_usage),
        'raw_data': results
    }
```

16.8 Real-World Performance

16.8.1 Industry Benchmarks

- **SPEC CPU2006**: CPU benchmark suite
- **SPEC CINT2006**: C integer benchmark

- **SPEC CFP2006**: C floating point benchmark
- **Compile-time benchmarks**: Compiler compilation speed

16.8.2 Cloud Compilation

```
# Cloud compilation benchmark
for cloud_provider in "aws" "gcp" "azure"; do
    echo "Testing $cloud_provider cloud compilation"

    # Measure compilation time
    start=$(date +%s%N)
    $cloud_provider compile build_project
    end=$(date +%s%N)

    echo "Compilation time: $((end - start)) seconds"

    # Measure cost
    cost=$($cloud_provider billing get-cost --project build_project)
    echo "Cost: $cost"
done
```

Aktivitas Pembelajaran

1. **Benchmark Setup**: Buat benchmark suite untuk compiler testing.
2. **Performance Analysis**: Analisis hasil benchmark dengan statistik.
3. **Optimization Testing**: Uji impact berbagai optimasi levels.
4. **Tool Comparison**: Bandingkan berbagai compiler tools.
5. **Automation**: Buat automated benchmarking pipeline.

Latihan dan Refleksi

1. Desain benchmark suite untuk compiler dengan multiple test cases!
2. Implementasikan high-resolution timer untuk akurasi pengukuran!
3. Analisis hasil benchmark dengan statistical methods!
4. Bandingkan performance GCC vs Clang untuk berbagai optimasi levels!

5. Identifikasi bottleneck dalam compilation pipeline!

6. **Refleksi**: Bagaimana performance evaluation mempengaruhi pengembangan compiler?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 6.2

A. Pilihan Ganda

1. Metrik yang TIDAK diukur dalam performance evaluation:
   (a) Code quality
   (b) Developer productivity
   (c) User satisfaction
   (d) Compilation speed
2. Tool untuk profiling memory usage:
   (a) GDB
   (b) Valgrind
   (c) /proc filesystem
   (d) System monitor
3. Optimasi yang memberikan speedup terbesar:
   (a) Loop unrolling
   (b) Vectorization
   (c) Inlining
   (d) Constant folding

B. Essay

1. Jelaskan metodologi benchmarking yang komprehensif untuk compiler evaluation!
2. Implementasikan automated benchmarking system dengan statistical analysis!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat mengevaluasi kinerja compiler dan optimasi

☐ Saya dapat merancang benchmark suite yang efektif

☐ Saya dapat menggunakan profiling tools untuk analisis

☐ Saya dapat melakukan statistical analysis pada hasil benchmark

☐ Saya dapat membandingkan performance berbagai compiler

☐ Saya dapat mengidentifikasi optimization opportunities

Rangkuman

Bab ini membahas performance evaluation dan benchmarking, termasuk metodologi, measurement tools, compiler comparison, optimization analysis, dan automated testing. Mahasiswa belajar mengevaluasi dan mengoptimasi kinerja compiler.

Poin Kunci:

- Performance evaluation mengukur efektivitas compiler
- Benchmarking menyediakan pengukuran yang sistematis
- Profiling tools membantu identifikasi bottleneck
- Statistical analysis memastikan validitas hasil
- Optimizations memiliki trade-off yang perlu dipertimbangkan
- Automation mempermudah testing berulang

Kata Kunci: ***Performance Evaluation***, ***Benchmarking***, ***Profiling***, ***Optimization***, ***Statistical Analysis***, ***Compiler Comparison***