3. Coding for Performance

3.1 Introduction

3.2 Defining Performance

3.3 How Code Structure Impacts Performance

3.4 How Data Structure Impacts Performance

3.5 How the Compiler Impacts Performance

3.6 Exercises

3.x Important Performance Analysis Tools

Sources:
[1] Darryl Gove: „Multicore Application Programming“, Chapter 3, Pearson Education, 2011, ISBN 978-0-321-71137-3

3.1 Introduction

- Performance of serial code still remains important even for parallel applications, because:
  - Each thread of a parallel application is still a serial stream of instructions,
  - Normally each parallel application contains some serial code which can not be parallelized

- Two performance optimization strategies:
  - Most often used strategy: Solve the performance problem after the program is functionally correct
  - Definitely the better way: Consider performance even during the initial design phase

- This chapter explains:
  - Where performance can be lost or gained
  - How early performance considerations lead to a better software

3.2 Defining Performance

- Two most important performance metrics are: Throughput and Latency
  - Throughput: (Rate, or items per time unit)
    can be:
    - Transactions per second
    - Jobs per hour
    - completed tasks per time unit
  - Latency: (Response time or time per item)
    - How long does it take to complete a task
- For some applications throughput is important, for others latency, and for some a combination of both

3.3 How Code Structure Impacts Performance

- Structure means:
  - Distribution of source code between source files
  - Distribution of source code into application and supporting libraries

3.3 How Code Structure Impacts Performance

Only One Source File

- Code distribution in one or several source code files?
  - Everything in one source code file:

```
int do_mul(int p1, int p2)
{
    return(p1 * p2);
}

long do_mul_add(long addp, int mul1p, int mul2p)
{
    return(addp + do_mul(mul1p, mul2p));
}

int main()
{
    long sum = 0;

    for(int i = 0; i < 10000000; i++) {
        sum = do_mul_add(sum, i, 2);
    }

    printf("sum = %ld\n", sum);
}
```

Compile&Link instruction:

```
gcc -O2 allinone.c -o allinone
```

3.3 How Code Structure Impacts Performance

Several Source Code Files

– Code distribution in several source code files:

main.c

```
extern long do_mul_add(long addp, int mul1p, int mul2p);
int main()
{
    long sum = 0;
    for(int i = 0; i < 10000000; i++) {
        sum = do_mul_add(sum, i, 2);
    }
    printf("sum = %ld\n", sum);
}
```

libA.c

```
extern int do_mul(int p1, int p2);
long do_mul_add(long addp, int mul1p, int mul2p)
{
    return(addp + do_mul(mul1p, mul2p));
}
```

libB.c

```
int do_mul(int p1, int p2)
{
    return(p1 * p2);
}
```

Compile&Link instruction:

```
gcc -O2 -c libA.c
gcc -O2 -c libB.c
gcc -O2 -c main.c
gcc -O2 main.o libA.o libB.o -o multisource
```

3.3 How Code Structure Impacts Performance

Performance Comparison

Scenario	Execution time in clock cycles	
	No compiler optimization (-O0)	Strong compiler optimization (-O2)
One source code file	180,000,000	
Several source code files	180,000,000	

3.3 How Code Structure Impacts Performance

Using Libraries (1)

- Code can be bundled in ‘Statically’ or ‘Dynamically’ linked libraries
- Statically linked library (also called ‘static library’):
  - A static library combines a number of object files into a single library
  - Linker extracts relevant code from the library at link time
    → Extracted code is part of the executable!
- Dynamically linked library (also called ‘shared library’):
  - A shared library combines a number of object files into a single library
  - Linking is done dynamically: Normally library code is loaded into main memory ‘on request’, i. e. if one of the library functions is called
    → Library code is NOT part of the executable!
  - Shared library can be used by several applications at a time
    → Less main memory consumption
    → Much smaller application code size
  - In case of a bug, a shared library can be updated without recompiling and linking the application
  - Most often shared libraries are compiled to use position independent code which is less efficient

3.3 How Code Structure Impacts Performance

Using Libraries (2)

- No direct call of a library function because the address where the function code is stored in main memory first has to be calculated using a so-called trampoline function and a jump table (PLT: Procedure Link Table)
- Example: printf call

App.

```
.text …
        bl printf@plt    // call trampoline function in
        …                // procedure linkage table section

.plt…
```

Tram-poline func.

```
printf@plt:
        add …            // calculate address of corresponding
        add …            // global offset table entry
        ldr pc,[printf@got.plt]

.text:
```

Shared Lib.

```
printf: push {fp, lr}
        …
        pop {fp, pc}
```

3.3 How Code Structure Impacts Performance

Scenario: Static Library

- Scenario using static libraries:

main.c

```
extern long do_mul_add(long addp, int mul1p, int mul2p);
int main()
{
    long sum = 0;
    for(int i = 0; i < 10000000; i++) {
        sum = do_mul_add(sum, i, 2);
    }
    printf("sum = %ld\n", sum);
}
```

Compile&Link instruction:

```
gcc -O2 -c libA.c
gcc -O2 -c libB.c
ar -r libAB.a libA.o
ar -r libBB.a libB.o
gcc -O2 -c main.c
gcc -O2 main.o libAB.a -o statLibs
```

libA.c

```
extern int do_mul(int p1, int p2);
long do_mul_add(long addp, int mul1p, int mul2p)
{
    return(addp + do_mul(mul1p, mul2p));
}
```

libB.c

```
int do_mul(int p1, int p2)
{
    return(p1 * p2);
}
```

3.3 How Code Structure Impacts Performance

Scenario: Shared Library

- Scenario using shared libraries:

```
extern long do_mul_add(long addp, int mul1p, int mul2p);
```

main.c

```
int main()
{
    long sum = 0;

    for(int i = 0; i < 10000000; i++) {
        sum = do_mul_add(sum, i, 2);
    }

    printf("sum = %ld\n", sum);
}
```

libA.c

```
extern int do_mul(int p1, int p2);

long do_mul_add(long addp, int mul1p, int mul2p)
{
    return(addp + do_mul(mul1p, mul2p));
}
```

libB.c

```
int do_mul(int p1, int p2)
{
    return(p1 * p2);
}
```

Compile&Link instruction:

```
gcc -shared -fpic -O2 libA.c libB.c -o libAB.so
gcc -O2 -c main.c
gcc -O2 main.o ./libAB.so -o sharedLibs
```

3.3 How Code Structure Impacts Performance

Performance Comparison: Static versus Shared Libraries

- Performance comparison of static and shared libs:

Scenario	Execution time in clock cycles Strong compiler optimization (-O2)
One source code file	1,000,000
Several source code files	91,000,000
Static libraries	91,000,000
Shared libraries	176,000,000

3.3 How Code Structure Impacts Performance

Some Guidelines for Using Libraries

- Some guidelines/hints for using libraries:
  - Rarely executed functions should be extracted into libraries
  - Frequently executed functions should better be extracted into static libraries and the so-called ‘cross file optimization’, if available, should be activated
  - Code which is used by several applications should be extracted in a dynamic library

3.4 How Data Structure Impacts Performance

Access Penalties

- Used data structure may have a very strong impact on performance, because a data item may be accessed a lot of times during program execution and because there are great differences in access time depending on where the item is stored

- Remember: Access times (penalties) of a memory hierarchy with two-level cache
  - Penalty of first level cache:
  - Penalty of second level cache:
  - Penalty of main memory:

3.4 How Data Structure Impacts Performance

Optimize Layout of Data in Memory

- Coding styles to optimize the layout of data in memory:
  1. Improving Performance through data density and locality
  2. Selection of appropriate array access patterns

1. Improving performance through data density and locality

- By placing heavily requested variables very close to each other so that they reside in the same cache line (and are loaded in ONE cache miss cycle!)
- Example: (Assumption: Cache line size = 64 bytes)

```
int var1;
int padding1[15];
int var2;
int padding2[15];
```

3.4 How Data Structure Impacts Performance

Optimize Data Access Pattern (1)

2. Selection of appropriate array access patterns

– Which of the following two codes executes faster?

```
#define DIM 10000

double arr[DIM] [DIM];
double sum = 0;

for(int k = 0; k < DIM; k++)
    for(int l = 0; l < DIM; l++)
        sum += arr[l] [k];
```

```
#define DIM 10000

double arr[DIM] [DIM];
double sum = 0;

for(int l = 0; l < DIM; l++)
    for(int k = 0; k < DIM; k++)
        sum += arr[l] [k];
```

3.4 How Data Structure Impacts Performance

Optimize Data Access Pattern (2)

– Memory layout of a two-dimensional array of doubles (m x n matrix with m: number of rows, n: number of columns)

```
arr[0, 0]     arr[0, 1]     … arr[0, n-1]
arr[1, 0]     arr[1, 1]     … arr[1, n-1]
                            …
arr[m-1, 0]   arr[m-1,1]    … arr[m-1, n-1]
```

Two possible layouts:

Memory address	Element
arr + 0	arr[0, 0]
arr + 8	arr[0, 1]
arr + 16	arr[0, 2]
...	...
arr + n · 8	arr[1, 0]

Elements of a row are adjacent!

Memory address	Element
arr + 0	arr[0, 0]
arr + 8	arr[1, 0]
arr + 16	arr[2, 0]
...	...
arr + m · 8	arr[0, 1]

Elements of a column are adjacent!

3.4 How Data Structure Impacts Performance

Optimize Data Access Pattern (3)

– Execution time in s of the code from page 21 (compiled with -O2 flag):

DIM	Row-major code	Column-major code
100	0.00003	0.00003
1000	0.0024	0.003
10000	0.3	2.0
20000	1.2	8.2

3.4 How Data Structure Impacts Performance

Summary

- Summary:
  - Best data structure layout depends on how the used algorithm accesses the data structure
  - If algorithm changes, layout of the data structure also may change
  - Very often there is a tradeoff scenario:
    Most natural and easiest to understand algorithm may not be the one which provides the best performance (because it does not respect computer’s architecture)

3.5 How the Compiler Impacts Performance

Limited Compiler Scope

- Compiler has to translate high-level language code to machine language code.

- Compiler optionally can optimize the code in order to:
  - Speedup execution
  - Reduce the memory footprint

- Some compiler optimization techniques:
  - Elimination of work
  - Restructuring of work
  - Inlining
  - ...

3.5 How the Compiler Impacts Performance

Compiler Optimization (1)

- Compiler may optimize code by <u>elimination of work</u>
  - Example: (Stupid implementation of a delay)

```
for(int i = 0; i < 1000; i++) { }
```

3.5 How the Compiler Impacts Performance
Compiler Optimization (2)

- Compiler may optimize code by <u>restructuring of work</u>
  - Example 1: (Better way to execute a division)

```
unsigned int b;
…
unsigned int a = b / 2;
```

  - Example 2: (Saving conditional code)

```
if((value & 1) == 1)
    odd = 1;
else
    odd = 0;
```

3.5 How the Compiler Impacts Performance

Compiler Optimization (3)

- Compiler may optimize code by Inlining
  - Example:

```
int do_mul(int p1, int p2)
{
    return(p1 * p2);
}

long do_mul_add(long addp, int mul1p, int mul2p)
{
    return(addp + do_mul(mul1p, mul2p));
}

int main()
{
    long sum = 0;

    for(int i = 0; i < 10000000; i++) {
        sum = do_mul_add(sum, i, 2);
    }

    printf("sum = %ld\n", sum);
}
```

3.5 How the Compiler Impacts Performance
Compile Flags

- Compiler optimization can be enabled and configured by compile flags
- GNU gcc compiler supports ~100 flags, most of them are related to code optimization
- GNU gcc compile flags every developer should know:

-g: Compiler adds additional code and data in order to support efficient debugging.

-O0: Compiler applies no optimization techniques in order to optimize execution speed or memory footprint.

-O1: Compiler tries to reduce code size and execution time, without performing any optimizations that significantly increases compilation time

-O2: Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. But this option increases compilation time.

-O3: Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on additional optimization flags which may increase compilation time (and hopefully speed)

-Os: Enables all -O2 optimizations except those that increase code size.

3.5 How the Compiler Impacts Performance
Cross-File Optimization (1)

- Compiler/Linker can better optimize code if it sees entire code, but compiler’s view is limited to only ONE source code file
- But linker sees all object code files. Basic idea:
  - Include additional information that enables linker to further optimize code by inter-procedural optimization, e. g. using inlining

3.5 How the Compiler Impacts Performance

Cross-File Optimization (2)

- Example:

```
extern long do_mul_add(long addp, int mul1p, int mul2p);

int main()
{
    long sum = 0;

    for(int i = 0; i < 10000000; i++) {
        sum = do_mul_add(sum, i, 2);
    }

    printf("sum = %ld\n", sum);
}
```

Compile&Link instruction:

```
gcc -O2 -flto -c libA.c
gcc -O2 -flto -c libB.c
gcc -O2 -flto -c main.c
gcc -O2 -flto main.o libA.o libB.o
    -o multisource
```

```
extern int do_mul(int p1, int p2);

long do_mul_add(long addp, int mul1p, int mul2p)
{
    return(addp + do_mul(mul1p, mul2p));
}
```

```
int do_mul(int p1, int p2)
{
    return(p1 * p2);
}
```

3.6 Exercises

1. Latency and throughput are two very important performance metrics. Which of both is suitable to describe the performance of a web-server?
2. Please discuss the pros and cons of having entire application source code stored in one file.
3. Please discuss the pros and cons of using shared libraries.
4. How does the layout of data in main memory, especially its density and locality, impact performance?
5. What is the characteristic of the row-major ordering used to store a two-dimensional array? Which programming language supports this kind of ordering?
6. What is the difference between gcc’s -Os and -O2 compile flags?