Code optimisations

course "Essentials of computing systems"

Feb.2022

contents

contents

writing efficient code

- Writing an efficient program in a given high-level programming language requires several types of skills.
- One must select the adequate algorithms and data structures.
- Source code must be in such a condition that the compiler can effectively produce code that runs fast or that occupies the minimum space.
- Speed and size are the traditional criteria of optimisation addressed by compilers.
- Energy consumption is gaining importance.
- It is relevant to comprehend how compilers try to apply some type of code optimisation to the programs.
- Programmers must know how to make programs more amenable for compilers to generate (more) efficient code.

readability vs. performance

- There are differences between a normal algorithm and a more efficient one.
- The former can be programmed in a matter of minutes, while the latter requires more effort to implement and refine.
- Many low-level optimisations tend to reduce the readability and the modularity of the program, making harder its modification.
- For code whose performance is relevant, applying optimisations is worthwhile.
- One should maintain some level of readability in the code.
- A compiler takes a valid program written in the source language code and generates a behaviourally-equivalent machine-level program.
- Compilers make use of sophisticated mechanisms to generate code that can be optimised according to different metrics.

contents

level of optimisations

- Most compilers employ advanced techniques to determine what values are computed in a program and how they are used.
- The compilers can exploit opportunities:
  1. to simplify expressions,
  2. to use a single computation in several different places,
  3. to reduce the number of times a given computation must be performed.
- Compilers allow programmers to control the level of optimisations.
- For example, `gcc` can be used with the option `-O` to specify which type of optimisations to apply.

compilers are conservative

- Compilers must only apply safe optimisations to a program.
- The resulting program must have the same external behaviour as an unoptimised version for all possible paths the program may take.
- A compiler is expected to be conservative in applying optimisations: whenever in doubt, it does not apply them.
- The compiler operates in a constrained context:
    1. It must not modify the behaviour of the program under any possible condition.
    2. The compiler has a limited and localised view of the program, so broader optimisations are not applied.
    3. The compilation process must be fast enough, so marginal gains are not appreciated if the compiler takes much longer.

optimisation blockers

- Compilers usually have problems in dealing with optimisation blockers.
- Optimisation blockers can greatly limit the opportunities for a compiler to generate optimised code.
- Often, the optimisation blockers are dependent on the execution environment.

```
int pr1(int *x, int *y) {
   *x += *y;
   *x += *y;
}
```

6 memory accesses

```
int pr2(int *x, int *y) {
   *x += 2* *y;
}
```

3 memory accesses

- Both programs add twice the value stored at the location designated by pointer `y` to that designated by pointer `x`.

optimisation blockers

- It seems that a compiler when handling procedure `pr1` could generate more efficient code based on the computations performed by the equivalent `pr2`.
- This approach cannot be applied when `x` and `y` are equal.

pr1	`*x += *x;`	`/* double value at x */`
	`*x += *x;`	`/* double value at x */`
pr2	`*x += 2* *x;`	`/* triple value at x */`

- The compiler cannot assume that arguments `x` and `y` are not equal.
- The situation where two pointers may designate the same memory location is called as memory aliasing.

optimisation blockers

```
int a, *b;
...
a   = 5;
*b  = 10;
...
a   = a+3;
```

optimisation blockers

- Another optimisation blocker may occur when a function has side effects.
- A function (or expression) has a side effect, if it alters the values of some variables outside its local context.
- The function has an observable effect besides returning a value to the calling procedure.

```
int func1(int x) {
   return (f(x)+f(x));
}
```

```
int func2(int x) {
   return ( 2*f(x) );
}
```

- Both versions of the function appear to have the same behaviour.

optimisation blockers

```
int f(int p) {
    return (p+counter++);
}
```

- `counter` is a global integer variable.
- The expression `p+counter++`, calculates the value `p+counter` and afterwards increments the value of `counter` by one.
- If `counter` is equal to 0, calling:
  - `func1(5)` returns 11 (5+6)
  - `func2(5)` returns 10 (2x5)
- The value of `counter` is also different in both cases: 2 after calling `func1` and 1 after `func2`.

optimisation blockers

```
void lower2upper(char *s) {
  int i;
  for (i=0; i<strlen(s); i++)
    if (s[i]>='a' && s[i]<='z')
      s[i] -= ('a' - 'A');
}
```

```
void lower2upper(char *s) {
  int i, length=strlen(s);
  for (i=0; i< length ; i++)
    if (s[i]>='a' && s[i]<='z')
      s[i] -= ('a' - 'A');
}
```

- Compilers tend to keep the call to `strlen` inside the loop, because it can have side effects.
- The loop body may change the string and its size.
- Compilers tend to treat procedures as black boxes that cannot be analysed.
- Compilers assume the worst case and the function call remains intact.

contents

types of optimisations

- Machine-independent optimisations improve the target code, but ignore any properties at the machine level.
- They include choosing the best (i.e., the fastest) algorithm for the problem at hand → not addressed in this course.
- Other optimisations that fit in this group are:
  1. code motion;
  2. elimination of unnecessary accesses to memory;
  3. loop unrolling;
  4. reduction of the number of procedure calls.

code motion

- Loops are good candidates for improvements.
- The most typical cases of code motion consist of statements in a loop that can be moved outside its body, without affecting its semantics.
- Moving statements from the inside to the outside of the loop obviously reduces the execution time of the program.

```
do {
   x=y+z;
   a[i]=i+x*x;
   i++;
} while (i<n);
```

```
x=y+z;
int const j=x*x;
do {
   a[i]=i+ j ;
   i++;
} while (i<n);
```

code motion

```
above  = val[(i-1)*n+j];
below  = val[(i+1)*n+j];
left   = val[i*n+j-1];
right  = val[i*n+j+1];
sum    = above + below + left + right;
```

code motion

```
above  = val[(i-1)*n+j];
below  = val[(i+1)*n+j];
left   = val[i*n+j-1];
right  = val[i*n+j+1];
sum    = above + below + left + right;
```

$(i * n) - n + j$

$(i * n) + n + j$

$(i * n) + j - 1$

$(i * n) + j + 1$

code motion

```
long inj = i*n + j;
above  = val[inj-n];
below  = val[inj+n];
left   = val[inj-1];
right  = val[inj+1];
sum    = above + below + left + right;
```

$(i * n) - n + j$

$(i * n) + n + j$

$(i * n) + j - 1$

$(i * n) + j + 1$

code motion

- Similar transformations can be applied to `WHILE-DO` loops, but in some cases the result is more complex.
- `WHILE-DO` loops execute zero or more times, while `DO-WHILE` loops execute at least once.

```
while (i<n) {
    x=y+z;
    a[i]=i+x*x;
    i++;
}
```

```
if (i<n) {
    x=y+z;
    int const j=x*x;
    do {
        a[i]=i+ j ;
        i++;
    } while (i<n);
}
```

code motion

- Compilers try to transform `WHILE-DO` and `FOR` loops in equivalent `DO-WHILE` loops, to avoid the `IF-THEN` statement.

```
for (i=0; i<100; i++) {
    ...
}
```

$\Updownarrow$

```
i=0;
while (i<100) {
    ...
    i++;
}
```

```
i=0;
do {
    ...
    i++;
} while (i<100);
```

- It is guaranteed that the `FOR` loop executes at least once.
- The assembly code for `DO-WHILE` loops, in general, can be made faster than equivalent `FOR` and `WHILE-DO` loops.

elimination of unnecessary accesses to memory

- Accesses to main memory are also obvious candidates to optimise the performance of a program.

```
int addAll (int *a, int *v)
{  int i;
   *v=0;
   for (i=0; i<100; i++)
      *v += a[i];
}
```

3 memory accesses per iteration

```
int addAll (int *a, int *v)
{  int i;
   int acc=0;
   for (i=0; i<100; i++)
      acc  += a[i];
   *v=acc;
}
```

1 memory access per iteration

loop unrolling

- Loop unrolling aims reducing the number of loop iterations, by increasing the number of elements computed on each iteration.
- Each loop iteration incurs in some non-effective computations, related to the control of the loop.

```
i=0;                  i=0;                    arr[0]=0;
do {                  do {                    arr[1]=0;
   arr[i]=0;             arr[i]=0;            arr[2]=0;
   i++;                  arr[i+1]=0;          arr[3]=0;
} while (i<120);         arr[i+2]=0;          ...
                         arr[i+3]=0;          ...
                         i+=4;                arr[118]=0;
                      } while (i<120);        arr[119]=0;
```

- Loop unrolling constitutes a space-time tradeoff.

loop unrolling

```
    movl $0, %eax
.lbl:
    movl $0, arr(,%eax,4)




    incl %eax
    cmpl $120, %eax
    jle .lbl
```

$\boxed{\text{481 } (1 + 120\text{x}4) \text{ instructions}}$

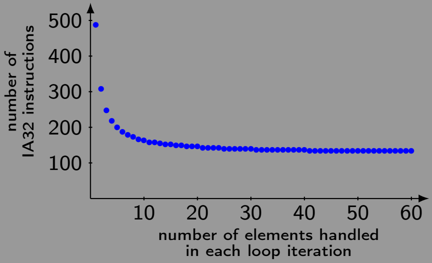
```
    movl $0, %eax
.lbl:
    movl $0, arr (,%eax,4)
    movl $0, arr+ 4(,%eax,4)
    movl $0, arr+ 8(,%eax,4)
    movl $0, arr+12(,%eax,4)
    addl $4, %eax
    cmpl $120, %eax
    jle .lbl
```

$\boxed{\text{211 } (1 + 30\text{x}7) \text{ instructions}}$

- If all instructions take the same time to execute (not the case), the reduction in time is around 56%.

loop unrolling

- For an array with n positions and a block of $k \leq \frac{n}{2}$ positions manipulated in each iteration, the total number of instructions is $1+(3+k)\times\lfloor\frac{n}{k}\rfloor+(n \bmod k)$.
- If n is not a multiple of k, additional instructions need to be put at the beginning or the end of the loop.
- If $n=21$ and $k=4$, there are five iterations with blocks of four elements, plus one instruction outside the loop.
- The performance improves whenever the block has more elements, but the code also occupies more space.



reduction of the number of procedure calls

- Procedure calls imply a great overhead and also tend to block many possible program optimisations.
- The idea of replacing a procedure call by the body of the called procedure is called inline expansion.
- It reduces time, at the cost of increasing the space usage.
- An inlined procedure runs faster than the normal procedure, as the calling overheads are avoided.
- However, code gets larger.
- Maintenance of the procedure gets harder, because when the body of the procedure needs to be changed, one must update it in several places.
- Inline expansion is adequate for small functions.

reduction of the number of procedure calls

```
int isEven (int num) {
   return !(num & 1);
}
```

```
if (isEven (number))
```

```
if (!(number & 1))
```

contents

different instructions

- Machine-dependent optimisations depend on the specific processor that is being considered.
- Optimisations that work in a given processor are not necessarily effective in a different one.
- These optimisations are related to the use of instructions that are faster or occupy less space in memory.
- When a given high-level statement is supported by different machine-level alternatives, one can use the best one, according to the preferred criterion.
- In IA32, assigning the value 0 to a register can be done with the following alternatives:

more space	less space
`movl $0, %eax`	`xorl %eax, %eax`

multiplication by a constant

- Another example occurs with multiplications, which in some processors take longer to execute than additions and shifts.
- Compilers transform a multiplication involving a variable and a constant in a series of additions and shifts.

```
y = 8*a;                  movl %eax, %ebx
                          sall $3, %ebx
```

- Things are not that simple when the constant is not a power of two.

```
y = 37*a;                 imull $37, %eax, %ebx
```

- Since $37 = 32+4+1$, calculating $37 \cdot a = 32 \cdot a + 4 \cdot a + a$.

```
movl %eax, %ecx           ecx = a
movl %ecx, %ebx           ebx = a
sall $2, %ecx             ecx = 4xa
addl %ecx, %ebx           ebx = a + 4.a = 5.a
sall $3, %ecx             ecx = 8.4.a = 32.a
addl %ecx, %ebx           ebx = 5.a + 32.a = 37.a
```

addresses

- The efficient computation of memory addresses is another machine-dependent optimisation.

```
arr[i]=0;
```

- If `i` is in ecx and the address array `arr` is in ebx, we have two alternatives for the assembly code:

$\boxed{\text{slower}}$

```
movl %ecx, %edx
sall $2, %edx
addl %edx, %ebx
movl $0, (%ebx)
```

$\boxed{\text{faster}}$

```
movl $0, (%ebx,%ecx,4)
```

contents

good locality

- A program that exhibits good locality is very likely to execute faster than one that does not.
- Good locality is possible if a program refers data items that are near other recently referenced data items or that were recently referenced themselves.
- This situation may have a great impact on the performance of the program, because the hit rate of the cache gets higher.
- Programmers must understand the principle of locality to positively exploit it.
- This principle is present in all levels of modern computer systems.
- For example, web browsers explore temporal locality by locally caching recently referenced documents.

good locality

```
int addAll (int *arr) {
   int i, acc=0;
   for (i=0; i<100; i++)
      acc += arr[i];
   return (acc);
}
```

- This procedure has a good temporal locality with respect to local variables `i` and `acc`.
- The elements of array `arr` are accessed one after the other.
- The procedure has good spatial locality with respect to array `arr`.
- Overall, the `addAll` procedure exhibits good locality.
- It has a stride-1 reference pattern, as it accesses each element of the array sequentially.

stride

- The stride of an array is the number of locations in memory between successive array elements.
- It is measured in bytes or in units of the size of the array's elements.
- As the stride increases, the spatial locality decreases, since two element arrays consecutively accessed are more distant.

stride

stride 1 unit

```
for (i=0; i<100; i++)
     for (j=0; j<100; j++)
        acc += arr[i][j];
```

stride 100 units

```
for (i=0; i<100; i++)
     for (j=0; j<100; j++)
        acc += arr [j][i] ;
```

1	[0][0]	1
2	[0][1]	101
3	[0][2]	201
…		…
99	[0][98]	9801
100	[0][99]	9901
101	[1][0]	2
102	[1][1]	102
…		…
200	[1][99]	9902
201	[2][0]	3
…		…
10 000	[99][99]	10 000