Lecture 07—Helping the C Compiler; Dependencies

ECE 459: Programming for Performance

January 28, 2014

Last Time

- Example: level-triggered vs edge-triggered
- A1 content (non-blocking I/O, curl implementation)—curl_multi_perform
- Synchronization primitives: mutexes, spinlocks, read-write locks, semaphores, barriers, lock-free algorithms.

Part I

Making C Compilers Work For You

Three Address Code

- An intermediate code used by compilers for analysis and optimization.
- Statements represent one fundamental operation—we can consider each operation atomic.
- Statements have the form:
  $$result := operand_1 \ operator \ operand_2$$
- Useful for reasoning about data races, and easier to read than assembly.
  (separates out memory reads/writes).

GIMPLE

- GIMPLE is the three address code used by `gcc`.
- To see the GIMPLE representation of your code use the `-fdump-tree-gimple` flag.
- To see all of the three address code generated by the compiler use `-fdump-tree-all`. You'll probably just be interested in the optimized version.
- Use GIMPLE to reason about your code at a low level without having to read assembly.

Live Coding Demo: GIMPLE

`volatile` Keyword

- Used to notify the compiler that the variable may be changed by “external forces”. For instance,

```
int i = 0;

while (i != 255) {
  ...
```

`volatile` prevents this from being optimized to:

```
int i = 0;

while (true) {
  ...
```

- Variable will not actually be `volatile` in the critical section and only prevents useful optimizations.
- Usually wrong unless there is a **very** good reason for it.

Branch Prediction Hints

As seen earlier in class, `gcc` allows you to give branch prediction hints by calling this builtin function:

```
long __builtin_expect (long exp, long c)
```

The expected result is that `exp` equals `c`.

Compiler reorders code & tells CPU the prediction.

The `restrict` Keyword

A new feature of C99: "The restrict type qualifier allows programs to be written so that translators can produce significantly faster executables."

- To request C99 in `gcc`, use the `-std=c99` flag.

`restrict` means: you are promising the compiler that the pointer will never alias (another pointer will not point to the same data) for the lifetime of the pointer.

Example of `restrict` (1)

Pointers declared with `restrict` must never point to the same data.

From Wikipedia:

```
void updatePtrs(int* ptrA, int* ptrB, int* val) {
    *ptrA += *val;
    *ptrB += *val;
}
```

Would declaring all these pointers as `restrict` generate better code?

Example of `restrict` (2)

Let's look at the GIMPLE:

```
void updatePtrs(int* ptrA, int* ptrB, int* val) {
 D.1609 = *ptrA;
 D.1610 = *val;
 D.1611 = D.1609 + D.1610;
 *ptrA = D.1611;
 D.1612 = *ptrB;
 D.1610 = *val;
 D.1613 = D.1612 + D.1610;
 *ptrB = D.1613;
}
```

- Could any operation be left out if all the pointers didn't overlap?

Example of `restrict` (3)

```
1  void updatePtrs(int* ptrA, int* ptrB, int* val) {
2   D.1609 = *ptrA;
3   D.1610 = *val;
4   D.1611 = D.1609 + D.1610;
5   *ptrA = D.1611;
6   D.1612 = *ptrB;
7   D.1610 = *val;
8   D.1613 = D.1612 + D.1610;
9   *ptrB = D.1613;
10 }
```

- If `ptrA` and `val` are not equal, you don't have to reload the data on **line 7**.
- Otherwise, you would: there might be a call

  `updatePtrs(&x, &y, &x);`

Example of `restrict` (4)

Hence, this markup allows optimization:

```
void updatePtrs(int* restrict ptrA,
                int* restrict ptrB,
                int* restrict val)
```

Note: you can get the optimization by just declaring `ptrA` and `val` as `restrict`; `ptrB` isn't needed for this optimization

Summary of `restrict`

- Use `restrict` whenever you know the pointer will not alias another pointer (also declared `restrict`)

It's hard for the compiler to infer pointer aliasing information; it's easier for you to specify it.

$\Rightarrow$ compiler can better optimize your code (more perf!)

Caveat: don't lie to the compiler, or you will get undefined behaviour.

Aside: `restrict` is not the same as `const`. `const` data can still be changed through an alias.

Next topic: Dependencies

Dependencies are the main limitation to parallelization.

Example: computation must be evaulated as `XY` and not `YX`.

Not synchronization

Assume (for now) no synchronization problems.

Only trying to identify code that is safe to run in parallel.

Memory-carried Dependencies

Dependencies limit the amount of parallelization.

Can we execute these 2 lines in parallel?

```
x = 42
x = x + 1
```

Memory-carried Dependencies

Dependencies limit the amount of parallelization.

Can we execute these 2 lines in parallel?

```
x = 42
x = x + 1
```

No.

- Assume x initially 1. What are possible outcomes?

Memory-carried Dependencies

Dependencies limit the amount of parallelization.

Can we execute these 2 lines in parallel?

```
x = 42
x = x + 1
```

No.

- Assume x initially 1. What are possible outcomes?
  x = 43 or x = 42

Next, we'll classify dependencies.

Read After Read (RAR)

Can we execute these 2 lines in parallel? (initially x is 2)

```
y = x + 1
z = x + 5
```

Read After Read (RAR)

Can we execute these 2 lines in parallel? (initially x is 2)

```
y = x + 1
z = x + 5
```

Yes.

- Variables y and z are independent.
- Variable x is only read.

RAR dependency allows parallelization.

Read After Write (RAW)

What about these 2 lines? (again, initially x is 2):

```
x = 37
z = x + 5
```

Read After Write (RAW)

What about these 2 lines? (again, initially x is 2):

```
x = 37
z = x + 5
```

No, z = 42 or z = 7.

RAW inhibits parallelization: can't change ordering.
Also known as a true dependency.

Write After Read (WAR)

What if we change the order now? (again, initially x is 2)

```
z = x + 5
x = 37
```

Write After Read (WAR)

What if we change the order now? (again, initially x is 2)

```
z = x + 5
x = 37
```

No. Again, $z = 42$ or $z = 7$.

- WAR is also known as a anti-dependency.
- But, we can modify this code to enable parallelization.

Removing Write After Read (WAR) Dependencies

Make a copy of the variable:

```
x_copy = x
z = x_copy + 5
x = 37
```

Removing Write After Read (WAR) Dependencies

Make a copy of the variable:

```
x_copy = x
z = x_copy + 5
x = 37
```

We can now run the last 2 lines in parallel.

- Induced a true dependency (RAW) between first 2 lines.
- Isn't that bad?

Removing Write After Read (WAR) Dependencies

Make a copy of the variable:

```
x_copy = x
z = x_copy + 5
x = 37
```

We can now run the last 2 lines in parallel.

- Induced a true dependency (RAW) between first 2 lines.
- Isn't that bad?

Not always:

```
z = very_long_function(x) + 5
x = very_long_calculation()
```

Write After Write (WAW)

Can we run these lines in parallel? (initially x is 2)

```
z = x + 5
z = x + 40
```

Write After Write (WAW)

Can we run these lines in parallel? (initially x is 2)

```
z = x + 5
z = x + 40
```

Nope, $z = 42$ or $z = 7$.

- WAW is also known as an output dependency.
- We can remove this dependency (like WAR):

Write After Write (WAW)

Can we run these lines in parallel? (initially x is 2)

```
z = x + 5
z = x + 40
```

Nope, z = 42 or z = 7.

- WAW is also known as an output dependency.
- We can remove this dependency (like WAR):

```
z_copy = x + 5
z = x + 40
```

Summary of Memory-carried Dependencies

		Second Access	
		Read	**Write**
First Access	**Read**	No Dependency Read After Read (RAR)	Anti-dependency Write After Read (WAR)
	Write	True Dependency Read After Write (RAW)	Output Dependency Write After Write (WAW)