CS60203: Design Optimization of Computing Systems *September 5, 2024*

Why write multi-core programs?

Instructor: Mainack Mondal *Scribe-By: <Swarup Padhi >*

Contents

1 Introduction to ILP and Multi-Core Programming

Instruction Level Parallelism (ILP) is a technique used to increase processor throughput by executing multiple instructions simultaneously. It can be classified into two main approaches:

- **Dynamic ILP (Hardware-based):** The hardware automatically detects parallelism at runtime, exploiting opportunities as they arise during program execution.
- **Static ILP (Software-based):** The compiler identifies parallelism during the compilation phase, rearranging instructions to expose more parallelism.

However, ILP has its limitations, particularly with basic blocks. Basic blocks, which are sequences of consecutive instructions without branches, limit the scope of ILP. In practice, most RISC programs contain basic blocks with only a few instructions (typically around six). This small size restricts the ability to find independent instructions that can be executed in parallel.

To overcome this, various techniques such as loop unrolling, instruction scheduling, and pipelining are employed. Multi-core programming, which is increasingly dominant, complements

ILP by exploiting thread-level parallelism, allowing programs to run across multiple cores simultaneously.

2 Challenges in ILP

Several challenges arise when trying to exploit ILP:

2.1 Data Hazards

Data hazards occur when instructions have dependencies on each other, and they can manifest in several forms:

- **Read After Write (RAW):** Occurs when an instruction tries to read a value before the previous instruction writes to it. This is the most common type of data hazard.
- **Write After Write (WAW):** Occurs when two instructions write to the same register in an out-of-order fashion, potentially causing incorrect results.
- **Write After Read (WAR):** Happens when an instruction writes to a register before a previous instruction has finished reading from it, leading to incorrect read values.

Example of a RAW hazard:

Listing 1: RAW Hazard

```
int x = 5;
int y = x + 2;  // x must be written first before y can read it
x = y * 2;      // Dependent on previous instruction
```

2.2 Control Hazards

Control hazards are caused by branch instructions, which introduce uncertainty about which instructions to execute next. These hazards limit the ability to reorder instructions, as the outcome of the branch instruction must be known first.

3 Software Techniques to Overcome ILP Challenges

Several compiler-based techniques are used to expose more ILP by mitigating data and control hazards:

3.1 Loop Unrolling

Loop unrolling involves duplicating the loop body multiple times within a single iteration, reducing the overhead associated with loop control (e.g., branch instructions) and allowing more instructions to execute in parallel. However, this technique increases code size and is less effective for loops with unpredictable iteration counts.

Listing 2: Loop Unrolling Example

```
for (int i = 0; i < n; i++) {
    sum += arr[i];
}
// Unrolled version
for (int i = 0; i < n; i+=4) {
    sum += arr[i];
```

```
    sum += arr[i+1];
    sum += arr[i+2];
    sum += arr[i+3];
}
```

3.2 Instruction Scheduling

This involves reordering instructions to minimize pipeline stalls by considering data dependencies and latencies. A well-designed instruction scheduler can greatly improve the efficiency of the pipeline, minimizing delays between instruction execution.

Listing 3: Instruction Reordering Example

```
int a = 5;
int b = a + 3;  // b depends on a
int c = 2;      // Independent of a and b, can be scheduled earlier
```

4 Hardware Techniques for ILP

4.1 Dynamic Scheduling

Dynamic scheduling techniques enable the CPU to execute instructions out of order to improve efficiency. Two popular approaches are:

- **Scoreboarding:** Introduced in the CDC 6600, scoreboarding dynamically tracks data hazards and schedules instructions for execution when their operands are available.
- **Tomasulo's Algorithm:** A more advanced technique, introduced in the IBM System/360 Model 91, Tomasulo's algorithm performs dynamic register renaming to eliminate WAW and WAR hazards. It also allows speculative execution and is widely used in modern processors.

4.2 Speculative Execution

Speculative execution predicts the outcome of branches and executes instructions ahead of time. If the prediction is correct, execution proceeds normally; otherwise, the speculative results are discarded. This allows the processor to overlap multiple basic blocks, enhancing throughput.

5 Drawbacks of SIMD and Vectorization

SIMD (Single Instruction, Multiple Data) enables parallel processing by performing the same operation on multiple data points simultaneously. However, several challenges limit its effectiveness:

5.1 Terrible Auto-Vectorization Support in GCC and Clang (2021 Status)

According to Wojciech Muła's analysis in early 2021, auto-vectorization in GCC and Clang has shown little improvement between versions. For example, algorithms such as **accumulate_custom_epi8** and **copy_epi8** are still not vectorized in GCC 10 or Clang 11 for AVX2 and AVX512 targets. Even when SIMD instructions are present, they may not be utilized effectively, resulting in no speedup. The main loop of **copy_if_epi8** doesn't get vectorized either, limiting SIMD effectiveness.

Here is a code snippet that demonstrates manual intervention is still required to help the compiler vectorize code properly:

Listing 4: Manual Vectorization for is$_s$*orted*

```
bool is_sorted3(int32_t* a, size_t n) {
    size_t i = 0;
    if (n > 4) {
        for (/**/; i < n - 4; i += 4) {
            if ((a[i] > a[i + 1]) | (a[i + 1] > a[i + 2]) |
                (a[i + 2] > a[i + 3]) | (a[i + 3] > a[i + 4])) {
                return false;
            }
        }
    }
    for (/**/; i + 1 < n; i++) {
        if (a[i] > a[i + 1])
            return false;
    }
    return true;
}
```

The above example shows how manual adjustments are needed for the compiler to detect opportunities for vectorization.

5.2 Downclocking Due to SIMD Instructions

Another issue with SIMD is that processors, particularly Intel CPUs, tend to downclock when executing AVX or AVX-512 instructions, reducing the overall system performance. This effect is especially harmful when running multi-core applications, as all cores may be affected.

6 Why Not Just Use Multiple Computers?

While distributed computing offers parallelism, it introduces significant overheads such as network communication delays, synchronization issues, and scalability limits, making it less effective for tasks with fine granularity or tight interdependencies.

7 Moving Towards Multi-core Programming

Given the challenges of SIMD and distributed computing, modern CPUs focus on multi-core architectures to handle parallelism more efficiently. Multi-core processors allow thread-level parallelism, which avoids the complexities of SIMD instruction sets and network communication delays of distributed systems.

8 Code Examples Highlighting ILP Limitations

Even with techniques such as loop unrolling and instruction reordering, there are situations where ILP reaches its limits due to data dependencies and control hazards. The following example demonstrates that even with loop unrolling, the performance gain is minimal due to inherent data dependencies:

Listing 5: Limits of ILP

```
for (int i = 0; i < n; i++) {
    arr[i] = arr[i] + arr[i-1];  // Dependent on previous iteration
}
```

```
// Even unrolling this loop won't help because of data dependencies
for (int i = 0; i < n; i+=4) {
    arr[i] = arr[i] + arr[i-1];
    arr[i+1] = arr[i+1] + arr[i];
    arr[i+2] = arr[i+2] + arr[i+1];
    arr[i+3] = arr[i+3] + arr[i+2];
}
```

This example highlights the limits of ILP, showing that certain problems cannot be parallelized due to sequential dependencies between iterations.

9 Conclusion

While ILP and SIMD offer some performance benefits, their limitations — data hazards, poor auto-vectorization, and downclocking effects — restrict their practical usage. Modern CPUs increasingly rely on multi-core architectures to efficiently handle parallelism without the drawbacks associated with SIMD or distributed computing. The combination of hardware and software improvements in instruction scheduling, dynamic execution, and thread-level parallelism continues to drive the evolution of high-performance computing.