Parallel Architecture

Kevin Heiser
4/7/04

Some standard types of Architecture

• SIMD – Single Instruction, Multiple Data
• Vector – A special case of SIMD
• MIMD – Multiple Instruction, Multiple Data
• Each one has benefits and drawbacks
**SIMD/Vector Architecture**

- Single instruction fetch
- Instructions executed in lock step
- Values stored in private memory or vector registers
- Excellent for static loops or programs with no branching
- Can’t easily do data dependent branching

**MIMD Architecture**

- Each processor executes independently
- May have private or shared memory
- Processes must be synchronized
- Handles data dependent branching well
- May perform poorly when there is a lot of synchronization and message passing
Another approach:

- Design a universal data-parallel architecture that adds mechanisms to handle different types of program behavior
- Adding features to increase versatility
- Discussed in first paper

Some Mechanisms

- Software managed cache
- Local instruction storage
- Instruction revitalization
- Local program counter control
- Local operand storage
- Operand revitalization
- L0 storage for constants
How it’s used…

• The machine is configured so that the most useful combination of mechanisms are combined
• It can be set up to run in a SIMD or a MIMD mode
• Adding additional mechanisms can help for some programs

Results

• Several configurations worked the best
• Best configuration performed better than specialized hardware in many cases
• Using a configuration that is not optimal can result in poor performance
Part 2

• Explicit versus implicit parallelism
• When is writing explicitly parallel code worth it?
• When is using implicitly parallel mechanisms good enough?

Explicit Parallelism

• Programmer determines where the parallelism should be
• This is what we have been doing for projects for these classes
Advantages

• Programmer can potentially find the best possible parallelism
• Have control over where synchronization occurs

Disadvantages

• Very time consuming to create parallel code by hand
• Easy for human programmer to make mistakes
• We need the time it takes to be worth the speedup that is achieved
Implicit Parallelism

• Hardware determines where the parallelism should be
• Often requires some help from an optimizing compiler to have the best results

Advantages

• Happens without any extra help from a programmer
• Finds ways to add parallelism where it would not otherwise be
Disadvantages

- May not be maximizing parallelism
- Often depends heavily on the compiler
- Usually requires a more complex architectural design

Some examples of implicit parallelism

- Pipelining – the old favorite
- Master/Slave Speculative parallelism (MSSP)
- Implicitly-Multithreaded Processors (IMP)
Pipelining

• What is it?
• An instruction uses only one part of the architecture in a given clock cycle
• Pipelining exploits this by starting a new instruction before the previous one is completed to extract instruction level parallelism

Issues For Pipelining

• Data Dependencies
• Some operations take multiple clock cycles
• Branching
• Structural hazards
Some Solutions

- Optimizing compilers
- Hardware detection of data dependencies
- Forwarding
- Branch prediction
- Out of order execution
- If we have to we can stall or flush the pipeline

Implicitly-Multithreaded Processors

- Spawns multiple threads
- Compiler specifies where threads begin
- Original thread is non-speculative, additional threads run speculatively
- Uses several novel techniques to improve it’s effectiveness
Techniques for optimization

- Instruction fetch policy – checks for dependencies to fetch
- Multiplexing hardware contexts – using multiple hardware contexts for each thread
- Hiding thread start-up delay – begins new thread while previous thread’s instructions are still being fetched

Aims of IMT

- Create parallelism out of a sequential program by running compiler specified threads
- Outperform a superscalar architecture and other similar proposals by implementing the features described on the previous slide
Results

• IMT generally outperforms superscalar on benchmarks
• Needs the compiler to specify threads to execute

Master/Slave Speculative Parallelization

• Parallelizes sequential programs during execution
• Uses a concept called a distilled program
• The master process executes the distilled program
• The master spawns slave processes to execute tasks
What is a distilled program?

- Basically, it is the most likely path that the program will take
- Running this program will hopefully be much faster than running the full program
- The distilled program is created by a special compiler for this architecture

What do the tasks do?

- The master program occasionally will assign an idle processor with a “task”
- These tasks are essentially mechanisms to check if the master is doing the correct thing
- If the master is doing something wrong, it’s thread will be squashed and restarted with the correct values
More on MSSP

- Takes advantage of the fact that most programs have a path that they take most of the time
- Takes advantage of the fact that this path is usually much faster than running the entire program

Why is it useful?

- Achieves speedup by taking a shorter path than the original program
- Uses processors that are idle
- Does not require that the distilled program be correct
Drawbacks

- Relies heavily on the distiller to produce a good distilled program and tasks for good performance
- Heavy cost for squashing the master thread
- Other processors can often be idle

Conclusions

- Implicit parallelization is useful to take advantage of resources that would not be otherwise used
- It usually is only as good as the compiler
- Usually falls short of well written explicitly parallel code
Discussion Questions

Both MSSP and IMT try to extract parallelism using generic means. Do you think it’s possible for them to approach the performance of hand-parallelized code?

Will flexible parallel architectures like the TRIPS processor replace more specialized parallel architectures? Why or why not?
What do all these advancements mean for writing parallel programs? Will explicit multithreading fade away like assembly coding?