Checker Processors

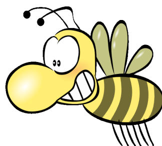
Smruti R. Sarangi

Department of Computer Science
Indian Institute of Technology
New Delhi, India

Outline

1. Introduction
2. Checker Processors
   - Checker Pipeline
   - Checking Mechanism
3. Advanced Checker Processors
   - Issues with DIVA
   - Reducing the Loss of Performance in DIVA
     - Core
     - Checker
     - L1 Failure
4. Design and Implementation

The Pentium Bug



Example

Story of the pentium bug: There was an error in the 9th digit for floating point division. The bug typically showed up in large financial and scientific calculations. Intel recalled all the Pentium chips and replaced them. It had lost about 500 million dollars, 20 years ago.

Stories of Some More Bugs

- A bug in AMD Opteron forced the designers to turn off prefetching. This had a massive performance impact.
- Bugs in IBM's G3 processor forced the designers to turn off power management.
- Due to soft errors, IBM stopped shipping its servers. Customers turned to Sun servers.
- Many more such bugs have been reported, resulting in massive losses to companies.
- Is there a way to stop faults from becoming failures?

Sources of Faults

- Hard Errors: Permanent defects in wires or transistors that typically manifest over time.
- Soft Errors: Transient faults caused by current pulses generated by alpha particles and neutrons.
- Design Bugs: Errors in cpu design.
- Process Variation related Faults: Faults due to problems introduced by the fabrication process.

Idea of a Checker Processor

- Have a checker processor that can check the results of the main processor.
  - Has significantly simpler logic. The checker is very extensibly verified. Hence, it has a low chance of having a design fault.
  - The transistors are bigger in size. This reduces the chances of soft errors.
  - The frequency is lower. This along with the fact that the design is simpler, ensure that the temperature is significantly lower. Low temperature $\Rightarrow$ less hard errors

Outline

1. Introduction
2. Checker Processors
   - Checker Pipeline
   - Checking Mechanism
3. Advanced Checker Processors
   - Issues with DIVA
   - Reducing the Loss of Performance in DIVA
     - Core
     - Checker
     - L1 Failure
4. Design and Implementation

Processor Pipeline

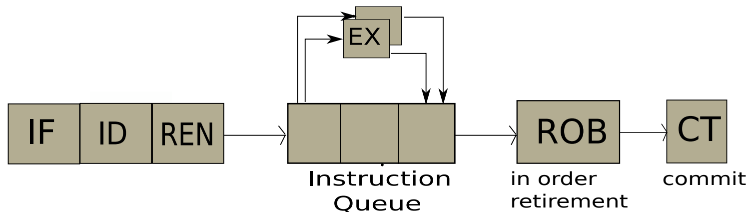


Figure 1: Regular processor pipeline

- Instructions are executed out of order.
- The reorder buffer(ROB) aggregates them, and commits their state in order. (Why ???)

DIVA Pipeline

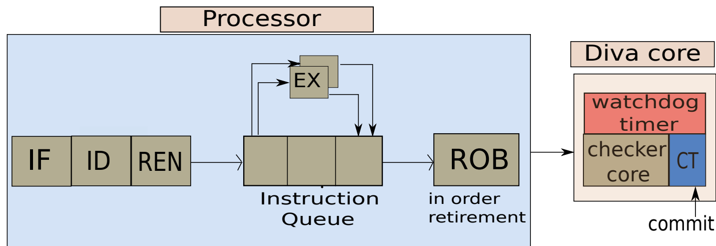


Figure 2: DIVA pipeline

- The DIVA pipeline adds a few extra stages.
- The output from the reorder buffer is sent to the checker core.
- The checker core commits the instruction after execution.

Outline

1. Introduction
2. Checker Processors
   - Checker Pipeline
   - Checking Mechanism
3. Advanced Checker Processors
   - Issues with DIVA
   - Reducing the Loss of Performance in DIVA
     - Core
     - Checker
     - L1 Failure
4. Design and Implementation

Passing Instructions to the Checker

- Every cycle the core pipeline sends an instruction packet to the checker pipeline.
- instruction packet $\Longrightarrow$ $<$program counter, opcode, operands, operand values, result, memory address accessed $>$

Example

Inst 20: add r3,r1,r2 (r3 = r1 + r2)
r3 = 5, r1 = 2, r2 = 3
The instruction packet will have the following elements

- Program Counter $\rightarrow$ 20
- opcode $\rightarrow$ add
- operands $\rightarrow$ r3, r1, and r2
- operand values $\rightarrow$ 2,3
- result $\rightarrow$ 5
- mem. address $\rightarrow$ none

The Process of Checking

- **Computation** : We need to check if the computation is correct
- **Communication** : We need to check if the correct operand values have been read and the correct result has been written.

ChkComp Pipeline

- Re-execute the instructions
- Verify that the result is correct

The Process of Checking

- **Computation** : We need to check if the computation is correct
- **Communication** : We need to check if the correct operand values have been read and the correct result has been written.

ChkComp Pipeline

- Re-execute the instructions
- Verify that the result is correct

ChkComm Pipeline

- Read the operands from the register file
- Re-execute loads from memory
- Verify that the register and memory values are correct

The Checker Core Pipeline

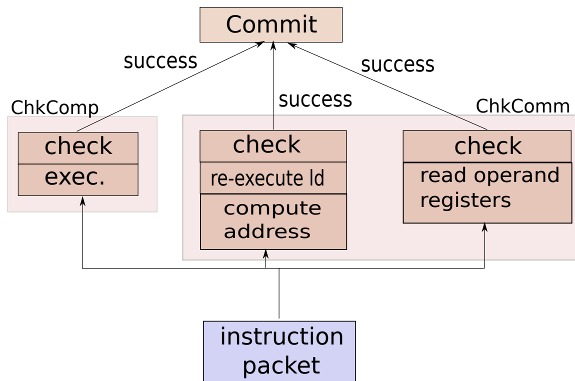


Figure 3: Checker core pipelines

What Happens when an Error is Detected

1. The checker sends a “flush pipeline” message to the core
2. The core flushes its pipeline
3. The checker commits the correct results to the architectural state
    - Core’s register file
    - Checker’s register file
    - Caches
4. The core resumes execution from the next instruction (one instruction after the erroneous instruction in program order)

Outline

1 Introduction

2 Checker Processors
- Checker Pipeline
- Checking Mechanism

3 Advanced Checker Processors
- Issues with DIVA
- Reducing the Loss of Performance in DIVA
  - Core
  - Checker
  - L1 Failure

4 Design and Implementation

Problems with DIVA

- What can be the problems with DIVA?

$$Performance = Frequency * IPC$$

- The checker is typically running at a slower frequency. Hence, its IPC needs to be higher.
- Why should the checker's IPC be higher
  - There are fewer dependencies between instructions in the checker (why not zero ???)
  - The core prefetches memory values for the checker
  - To summarize: A checker has higher ILP (Instruction Level Parallelism)

Problems with DIVA-II

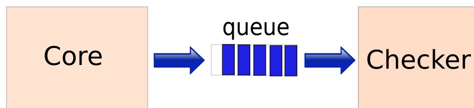
- Why should the checker's IPC be lower
  - The core typically has a higher peak issue rate than the checker.

Example

- For a core that can issue upto four instructions per cycle, it might reach a peak IPC of 1.5. It is sufficient for the checker to have an IPC of 2 in this case.
- In this case, we can have a checker with a smaller area.

  - What if the core has an IPC that is greater than the checker?
    - While designing the checker, we need to ensure that this case does not arise very frequently
    - However, there might be some benchmarks or instances within a benchmark that exhibit this behaviour

Problems with DIVA-III



Definition

ILP Instruction Level Parallelism : The average number of instructions that can be executed in parallel in one cycle

- For codes with high ILP, the core will outrun the checker
- In this case the checker pipeline will stall.
- This will lead to core stalls, and the overall performance will reduce.

Outline

1. Introduction
2. Checker Processors
   - Checker Pipeline
   - Checking Mechanism
3. Advanced Checker Processors
   - Issues with DIVA
   - Reducing the Loss of Performance in DIVA
     - Core
     - Checker
     - L1 Failure
4. Design and Implementation

Argument from an IPC Perspective

The overall of IPC in the system can decrease because of the:

core

Resource contention

- Register File
- Ld/St Queue
- L1 Cache

Lack of entries in

- Reorder Buffer
- Ld/St Queue

checker

- Very high ILP
- Data dependences
- Cache misses

Register File Contention

- Every cycle the core accesses its register file for read and write accesses.
- The checker needs to access the register file to ensure that the correct values are read, and the correct values are written.

Register File Contention

- Every cycle the core accesses its register file for read and write accesses.
- The checker needs to access the register file to ensure that the correct values are read, and the correct values are written.

Replicate the register file. Have a dedicated checker RF

Register File Contention

- Every cycle the core accesses its register file for read and write accesses.
- The checker needs to access the register file to ensure that the correct values are read, and the correct values are written.

Replicate the register file. Have a dedicated checker RF

- The checker needs only architectural registers.
- The checker populates the register file of the core after a fault is detected.

Register File Contention - II

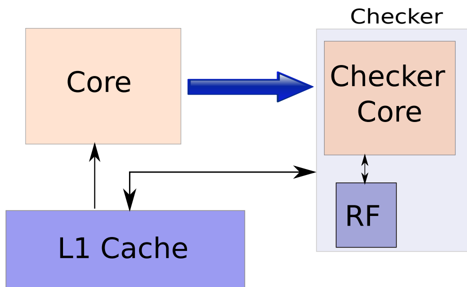
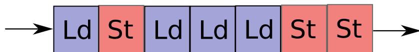


Figure 4: DIVA with a dedicated checker register file

Load-Store Queue Contention

Definition

Load-Store Queue: It is a FIFO queue of load and store entries. Each entry stores the ld/st address and in the case of a store, the value.



Rules for using the LSQ

- For any load find the first matching store (if any) and forward the value.
- All stores happen after the instruction commits.
- A LSQ entry is allotted at the time of instruction decode. The entry is populated with the ld/st address after it is generated.

Load-Store Queue Contention-II

- When loads in the checker execute, they need to check the ld/st queue of the core for any pending stores.
- They will collide with loads from the core, and this will result in contention.

Solution:

- Replicate just the store queue in the checker.
- This will eliminate some of the accesses to the ld/st queue in the core. Not all.
- When a store instruction commits in the checker, the corresponding entries in the core and checker store queues are removed

Load-Store Queue Contention-II

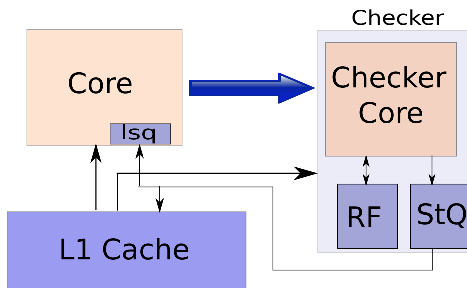


Figure 5: Addition of a store queue to the checker

Eliminating Cache Conention

- The core accesses the L1 cache for loads and stores.
- The checker also needs to do the same.
- Hence, the bandwidth requirement of the L1 cache will double.

Eliminating Cache Conention

- The core accesses the L1 cache for loads and stores.
- The checker also needs to do the same.
- Hence, the bandwidth requirement of the L1 cache will double.

Idea: Create a small L0 cache for the checker. Any data that is touched by the core will automatically be forwarded to the L0 cache.

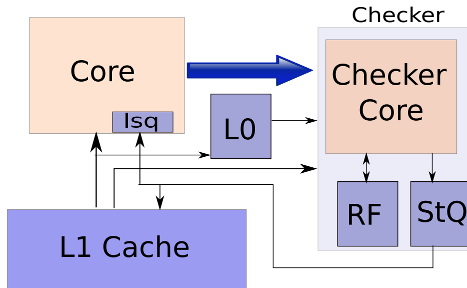


Figure 6: Checker with L0 cache

Lack of Entries

With a checker in place, speculative entries remain within the pipeline for a longer time. This puts pressure on resources.

- Lack of entries in the reorder buffer(ROB)
  - Increase the size of the ROB.
- Lack of entries in the Ld/St queue
  - Once a ld/st instruction moves to the checker, which has a store queue, remove the entry.

Very High ILP

Example

Consider a program that is adding two arrays:
c[1...n] = a[1...n] + b[1...n] There are potentially *n* parallel operations.

- With properly written code, we potentially have an IPC of *n*
- The core will clearly outperform the checker

Very High ILP

Example

Consider a program that is adding two arrays:
c[1...n] = a[1...n] + b[1...n] There are potentially n parallel operations.

- With properly written code, we potentially have an IPC of n
- The core will clearly outperform the checker

Solutions

- Only Divine intervention can help you !!!

Data Dependences

Example

Dependent Instructions
Inst1: r1 = ld r2
Inst2: r3 = r1 + r4
Inst1 and Inst2 are dependent through register, r1

- The ChkComm pipeline for Inst2 will stall. It will not be able to check that r1 is read correctly, because of Inst1.
- Inst1 and Inst2 cannot be checked parallely.
- If Inst2 can be checked just one cycle after Inst1, then we are fine. (Can be achieved through bypass paths)
- If there is a longer latency, between these dependent instruction, then we are limiting the IPC

Cache Misses in the Checker

Solutions

- The store queue will reduce write misses.
- The L0 cache will reduce cache misses to almost zero.

Example

Nowadays, the L1 cache, and even sometimes the L2 cache is part of the core. What if there is a problem in the caches that ECC cannot fix?

- Bugs in the cache access logic
- Bugs in the cache coherence protocol
- The clock cycle is too fast, and there are timing errors.

Idea: Have a separate L1 cache for the checker.

- The core's L1 cache only holds speculative state.
- The checker's L1 cache is the real L1.
- Once a fault is detected : flush the pipeline and clean up the core's L1

Diva with Split L1

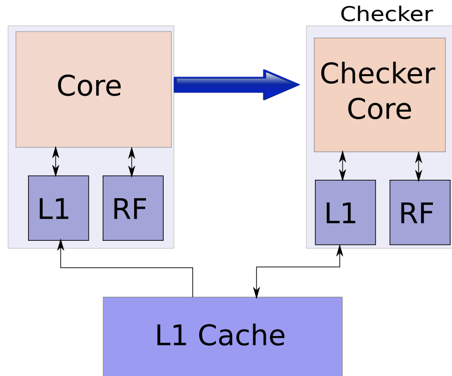


Figure 7: DIVA with a split L1 cache

- DIVA was designed and fabricated by Todd Austin and group in 2001.
- Experiment configuration
  - Base Processor : Alpha 21264, four issue, seven stage pipeline
  - Four wide, three stage checker
  - Synthesized with Synopsys 250 nm library

Conclusions

- 1.5% power overhead
- 6% area overhead
- Timing overhead limited to 3%

DIVA: a reliable substrate for deep submicron microarchitecture design, Todd Austin, Micro, 2000 http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.17.5782&rep=rep1&type=pdf

Efficient Checker Processor Design, Saugata Chatterjee, Chris Weaver, Todd Austin, MICRO-33, 2000 http://www.eecs.umich.edu/~taustin/papers/MICRO33-divadesign.pdf

Fault Tolerant Approach to MicroProcessor Design, Chris Weaver, Todd Austin, DSN 2001.