Delayed Exceptions — Speculative Execution of Trapping Instructions

M. Anton Ertl Andreas Krall

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
{anton,andi}@mips.complang.tuwien.ac.at
Tel.: (+43-1) 58801 {4459,4462}
Fax.: (+43-1) 505 78 38

Abstract. Superscalar processors, which execute basic blocks sequentially, cannot use much instruction level parallelism. Speculative execution has been proposed to execute basic blocks in parallel. A pure software approach suffers from low performance, because exception-generating instructions cannot be executed speculatively. We propose delayed exceptions, a combination of hardware and compiler extensions that can provide high performance and correct exception handling in compiler-based speculative execution. Delayed exceptions exploit the fact that exceptions are rare. The compiler assumes the typical case (no exceptions), schedules the code accordingly, and inserts run-time checks and fix-up code that ensure correct execution when exceptions do happen.

Key Words: instruction-level parallelism, superscalar, speculative execution, exception, software pipelining

1 Introduction

Computer designers and computer architects have been striving to improve uniprocessor performance since the invention of computers[JW89, CMC+91]. The next step in this quest for higher performance is the exploitation of significant amounts of instruction-level parallelism. To this end superscalar, superpipelined, and VLIW processors[1] can execute several instructions in parallel. The obstacle to using these resources is the dependences between the instructions. Scheduling (code reordering) has been employed to reduce the impact of dependences. However, the average instruction-level parallelism available within basic blocks is less than two simultaneous instruction executions [JW89].

To circumvent this barrier several methods have been developed to execute basic blocks in parallel. They are based on speculative execution, i.e. the processor executes instructions from possible, but not certain future execution paths. This can be implemented in hardware through backup register files, history buffers or reservation

[1] For simplicity, we will use the term "superscalar" in the rest of the paper, but delayed exceptions can be used with any of the techniques for exploiting instruction-level parallelism.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: Compiler Construction (CC '94), Springer LNCS 786, 1994, pages 158-171

stations [Tom67, HP87, SP88, Soh90]. A less expensive approach relies on compiler techniques for global instruction scheduling like trace scheduling, software pipelining and percolation scheduling [RG81, Fis81, Ell85, Nic85].

Exceptions pose serious problems to compiler-only approaches: The compiler must not move exception-generating instructions up across conditional branches (unless the instruction appears in all directions that the branch can take). E.g., a load that is moved up across its guardian NULL pointer test will trap wrongly. In addition, exception-generating instructions must not be reordered with respect to other exception-generating instructions, because the order of the exceptions would be changed. All these control dependences restrict the instruction-level parallelism to a low level: It is hardly higher than the level possible without speculative execution.

In this paper we propose *delayed exceptions*, a technique that combines low hardware cost, high performance and correct exception handling by putting most of the responsibility for exception handling on the compiler. Delayed exceptions can be implemented as a binary compatible extension of current architectures.

2 The Basic Idea

Delayed exceptions exploit the fact that exceptions are rare. The compiler assumes the typical case (no exceptions), schedules the code accordingly, and inserts run-time checks and fix-up code that ensure correct execution when exceptions do happen.

To implement this idea, every register is augmented with an exception bit; two versions of exception-generating instructions are needed: The trapping version is used in non-speculative execution and behaves traditionally. The trap-noting version is used for speculative execution; it does not trap, but notes in the trap bit of the result register whether the instruction would have trapped. This trap-noting instruction can be moved around as freely as other non-trapping instructions. Instructions dependent on that instruction can then also be moved up by speculating on the outcome to the exception-checking branch.

Finally a branch instruction checks trap-notes and branches to the fix-up code if necessary. The fix-up code triggers the trap and recalculates registers that received wrong values. The fix-up code is subject to the same control dependences that the exception-generating instruction was originally.

3 A Motivating Example

We will introduce the concept of delayed exceptions by a small example. Figure 1 shows the C function `strlen` which computes the length of a zero-terminated string. Figure 2 shows the assembly language output of a compiler for the MIPS R3000. We have changed the register names to make the program more readable.

The problem in software pipelining this loop is the `lb` (load byte) instruction, which can trap on illegal memory access. Assuming a two-cycle load latency and a one-cycle branch latency, each iteration needs three cycles even on a superscalar processor, unless delayed exceptions are used to enable speculative execution of the `lb`.

```
int strlen(char *s) {
  char *t = s;
  while (*s != '\0')
    s++;
  return s-t;
}
```

Fig. 1. The C function `strlen`

```
 #   1    int strlen(char *s) {
strlen:
 #   2    char *t = s;
        move    t,s         # t=s
 #   3    while (*s != '\0')
        lb      t0,0(s)     # t0=*s
        beqz    t0,end      # while (t0 != '\0')
loop:
 #  4     s++;
        addu    s,s,1       # s++
        lb      t0,0(s)     # t0=*s
        bnez    t0,loop     # while (t0 != '\0')
end:
 #   5    return s-t;
        subu    v0,s,t      # return_value = s-t
        j       ra          # return
```

Fig. 2. MIPS R3000 assembly language source of function `strlen`

		move t,s		
	lbx$_0$ t0,0(s)	addu$_0$ s,s,1		
	lbx$_1$ t1,0(s)	addu$_1$ s,s,1		
loop0:	lbx$_n$ t2,0(s)	addu$_n$ s,s,1	bx$_{n-2}$ t0,xcept0	beqz$_{n-2}$ t0,ret0
loop1:	lbx$_{n+1}$ t0,0(s)	addu$_{n+1}$ s,s,1	bx$_{n-1}$ t1,xcept1	beqz$_{n-1}$ t1,ret0
loop2:	lbx$_{n+2}$ t1,0(s)	addu$_{n+2}$ s,s,1	bx$_n$ t2,xcept2	bnez$_n$ t2,loop0
ret0:		subu s,s,3		
ret1:		subu v0,s,t		j ra
xcept0:	lb$_{n-2}$ t0,-3(s)			
				bnez$_{n-2}$ t0,loop1
		subu s,s,3		b ret1
xcept1:	lb$_{n-1}$ t1,-3(s)			
				bnez$_{n-1}$ t1,loop2
		subu s,s,3		b ret1
xcept2:	lb$_n$ t2,-3(s)			
				bnez$_n$ t2,loop0
		subu s,s,3		b ret1

Fig. 3. Software pipelined version of Fig. 2 with delayed exceptions

We software pipelined this code using delayed exceptions (see Fig. 3). We unrolled the loop thrice and renamed registers to eliminate write-after-read (WAR) dependencies. The loop now executes in one cycle/iteration on a hypothetical superscalar processor[2], unless an exception occurs. We assume that the processor has enough resources to execute one line (of Fig. 3) per cycle.

A few words of explanation are necessary. `lbx` (load byte and note exception) is the trap-noting version of `lb`; It sets the exception bit of the result register if an exception occurs and clears it otherwise. If one of the earlier bytes was zero, the function will return without ever seeing the `bx` belonging to the `lbx`. I.e., exceptions caused by wrong speculations are ignored. However, if the speculation was right, the `bx` will be executed; If the exception bit is set, the `bx` (branch on exception bit) instruction branches to the fix-up code. In the present case, the fix-up code consists of a `lb` that accesses the same address as the `lbx` and thereby calls the trap handler.

The indices of the instructions indicate the iteration the instructions belong to. The `addu` is executed speculatively, too. We could have renamed registers to save the old value of `s` for the off-loop execution paths and for the fix-up code. Instead, our code repairs `s` by subtracting 3.

4 The Compiler Technique

4.1 The Percolation Scheduling Framework

Percolation scheduling is a general framework for global instruction scheduling [Nic85]. It contains a few *core transformations* for moving instructions. *Enabling transformations* (e.g. register renaming) give the core transformations greater freedom to move the code. *Guidance rules* decide when and where to apply the transformations.

In this framework, the exception delaying transformation described below is an enabling transformation.

Delayed exceptions can also be fitted into other global scheduling models. E.g., in the context of trace scheduling [Fis81, Ell85], exception-generating instructions would be moved around freely; the fix-up code is inserted by the book-keeping process.

4.2 The Exception Delaying Transformation

The basic transformation used in delayed exceptions is shown in Fig. 4. The trapping instruction is split into a trap-noting instruction and the exception-checking branch `bx` to the fix-up code.

The fix-up code must trigger the trap and set the registers to the correct values. Before applying other transformations, the only register to be recomputed is the result register of the instruction. These functions are performed by the trapping version of the instruction[3].

---

[2] We did not exploit more parallelism (e.g. by *combining*[NE89] the `addu`s or by speculating in both directions) in order to keep the example simple.

[3] Of course, the transformation should not be reapplied to the trapping instruction in the fix-up code.

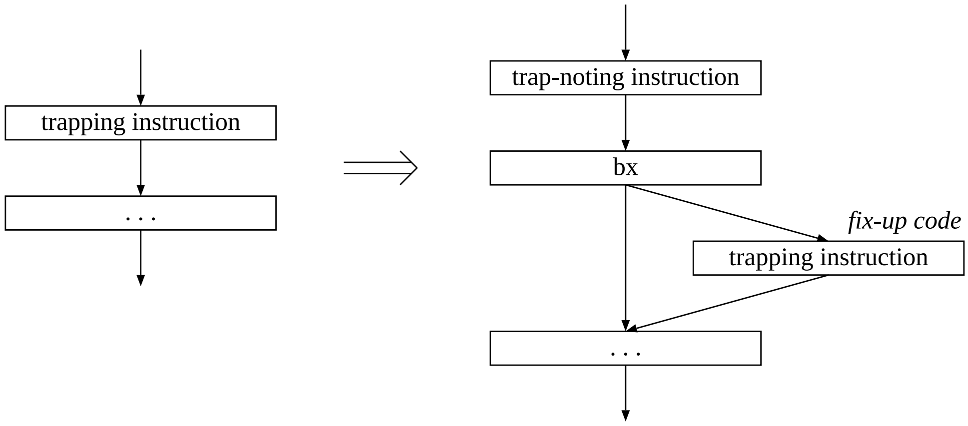

Fig. 4. The exception delaying transformation

The trapping version of the instruction in the fix-up code is still subject to the old control-dependences, i.e. it must not be moved across other trapping instructions (this preserves the order of exceptions) or up across branches.

This transformation may appear to be a bad deal, because it increases the number of executed instructions. However, on superscalar processors the execution time is determined mainly by dependences between instructions. Due to the dependences there are often idle resources (instruction bandwidth, functional units); these resources can be utilized for executing independent instructions without increasing execution time. Therefore, replacing dependences with additional instructions is often a win.

4.3 Enabled transformations

The exception delaying transformation is an enabling transformation. First of all, it enables the compiler to move the trap-noting version of the instruction up across branches (i.e. speculative execution of the instruction). More importantly, it also makes speculation on the outcome of the trap-check possible. I.e., instructions that depend on the exception-generating instruction can be moved up across the exception-checking branch and other branches.

How is this done? When moving the instructions up into the branches of the exception-checking conditional, both branches get a copy of the instruction. The copy in the fix-up code stays there, bound by the dependence on the trapping instruction, while the other copy can move further up until it reaches the trap-noting instruction. The code motions have to take the data dependences into account, so the source registers of the operations in the fix-up code are preserved, since they are live until the fix-up code. Transformations like register renaming or repairing [EK92] can be used to remove these dependences and enable further moves. The trap-notes have to be treated like normal registers, i.e. the trap-noting instruction is a definition and the trap-check is a use (see Section 5.3).

```
if (p != NULL) {
  i = j + p->info;
  p = q;
  ...
}
```

Fig. 5. A common C fragment

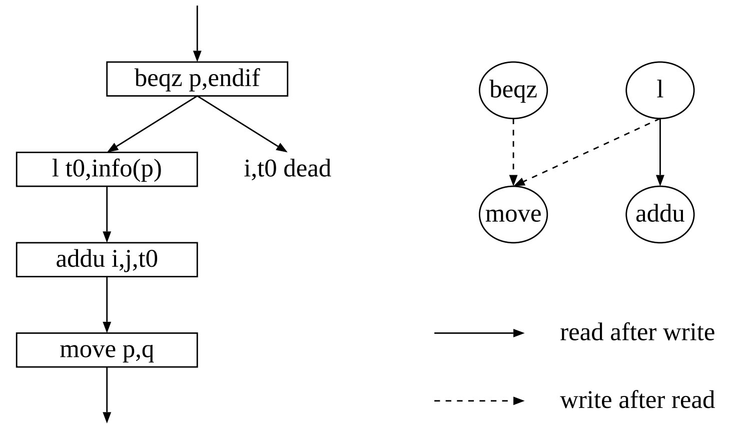

Fig. 6. Assembly version of Fig. 5

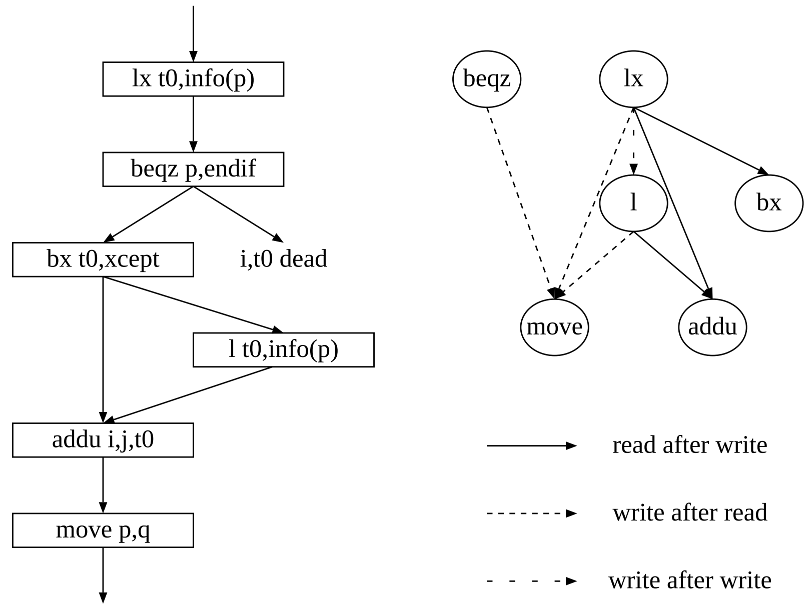

Fig. 7. Figure 6 after the exception delaying transformation

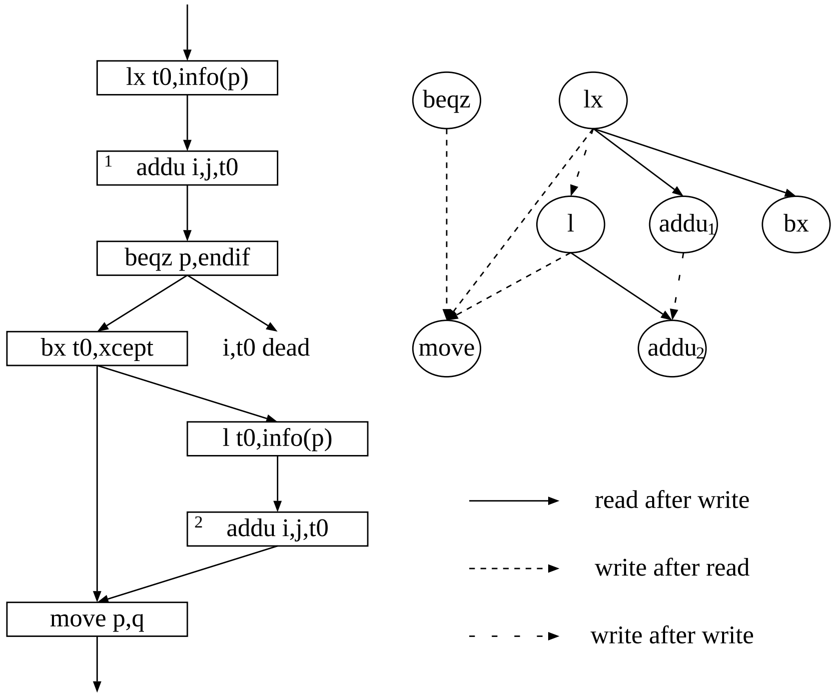

Fig. 8. Figure 7 after another code motion

As an example, we transform a typical C idiom (see Fig. 5). Figure 6 shows its assembly language version and the data dependence graph. Let us assume that there is a need to move the operations up in order to reduce the impact of later dependences. First we apply the exception delaying transformation (see Fig. 7). Then we try to move the other instructions (see Fig. 8). This is possible with the `addu`, but the `move` cannot be moved above the `bx`, since it would destroy the source register of the `l` (this is represented as a write-after-read dependence in the data dependence graph).

4.4 Controlling the Code Expansion

The example also shows a possible problem with delayed exceptions: code explosion. The two remedies used in the context of run-time disambiguation [Nic89] should work well for delayed exceptions, too: applying delayed exceptions only to frequently used program parts; and using one piece of fix-up code for several trapping instructions. Finally, the cache and paging behaviour can be improved by moving the fix-up code out-of-line into extra pages. An upper bound for the size of the recovery code is the original code size times the average depth of speculation, if there is one piece of recovery code per (non-`bx`) branch.

5 Architectural Considerations

5.1 Which Instructions are Affected

A program can trap on memory access exceptions, arithmetic exceptions (overflow and divide by zero) and explicit trap instructions.

Memory access exceptions can be generated by loads and stores. Loads can be easily executed speculatively using delayed exceptions. Stores alter memory, so they can hardly be undone. Therefore exceptions are not the only obstacle to speculative execution of stores. Fortunately there is no need to execute stores speculatively, since dependences on the store can be eliminated by run-time disambiguation [Nic89] and register renaming. Therefore a trap-noting store is neither necessary nor sensible.

Arithmetic exceptions can be generated by many floating-point and integer instructions. Trap-noting versions of these instructions are needed. Instructions that also have a non-trapping version (e.g. `add` (trapping) and `addu` (non-trapping)) are a special case. Instead of producing a third version, the non-trapping instruction could be changed into a trap-noting instruction in this case. However, this can affect register allocation, as explained in Section 5.3. The architect has to balance opcode waste against a (probably tiny) performance loss due to possible spilling.

Trap instructions are usually used for system calls, emulation of non-implemented hardware or bounds checks. Noting a trap early by speculation offers no advantages. Therefore trap-noting versions of trap instructions do not make sense.

5.2 Accessing the exception bits

The exception bits have to be conserved across context switches. Therefore they can be read and written through (a) special control register(s). Before reading them they must be up-to-date. This can be achieved by waiting until all pipelines have run dry.

This possibility of access can prove useful for other situations, too: Interrupt handlers can use delayed exceptions, if they save and restore the exception bits. It could even be used for saving and restoring around procedure calls, enabling better scheduling across calls. Of course this would require user-level access to the exception bits and a faster method than letting the processor run dry.

5.3 Noting the Exception

In the examples above, the exception is noted by a bit associated with the result register of the trap-noting instruction. This provides for simple adressing, but restricts register allocation: The register may not be used as the result register of a trap-noting instruction until the note is dead, i.e. until it is checked or an execution path is chosen that does not contain the check (i.e. an execution path that would not have contained the trapping instruction in the first place).

As an alternative, the notes could be addressed explicitly and allocated separately. This would make register allocation easier, but trap-noting instructions would need extra bits in the instructions for addressing the note.

5.4 Precise Exceptions

Some instructions (e.g. memory access and floating point on the MC88100) can cause exceptions late in the pipeline, when other, later instructions have already modified the processor state. Therefore, the processor cannot be simply restarted from the faulting instructions. These imprecise exceptions have several disadvantages, e.g. they make exception handlers implementation-dependent. In order to implement precise interrupts, many expensive hardware schemes for restoring the processor state have been proposed [SP88, HP87].

Delayed exceptions open the road to precise exceptions without any backup hardware: The exception delaying transformation must be applied to every instruction in the program that can cause imprecise exceptions. In the fix-up code, a special instruction `tx` (trigger exception) is prepended to the trapping instruction. `tx` traps early and therefore precisely.

There's just one problem: The trap does not occur at the instruction that originally caused the exception, but at the `tx`. However, the compiler makes sure that the input registers have the right values for rerunning the exception-causing instruction.[4] In addition to the values of the registers, the exception handler usually wants to know the exception-causing instruction and the type of the exception. The latter can be stored in the exception note (which has to be extended for this purpose), while the instruction resides just after the `tx`.

5.5 Page Faults

Like on any other processor, instruction fetch page faults are handled immediately. Data access page faults caused by a trap-noting instruction are noted like other exceptions and later handled in the fix-up code, if the processor executes the appropriate path. If a different path is executed, the note has no effect. This is an advantage over the approach proposed for general percolation [CMC+91], where speculative page faults are always serviced, even if they are not needed.

5.6 Instruction Issue Bandwidth

In the programs we measured (see Section 6) 16%–24% of the dynamically executed instructions can generate exceptions. Since this is nonnumerical code, most of these instructions will be executed speculatively on a high-degree superscalar machine. Each of these speculatively executed instructions would add a `bx` instruction to the instruction stream. Delayed exceptions increase the needed instruction issue bandwidth by up to 24% and need additional functional units for executing the `bx` instructions. These needs can be reduced by having `bx` instructions that test several notes and branch to a combined piece of fix-up code.

[4] These are the same compiler techniques that ensure correct processing of delayed exceptions (see Section 4.3).

6 Potential Speedup

To evaluate the potential benefit of delayed exceptions, we performed a trace-driven simulation [BYP+91, Wal91, LW92]. To get an upper bound, we assumed infinite resources (instruction bandwidth, functional units), perfect register renaming, perfect alias detection, and perfect branch prediction. To have a few machine models between the extremes, we restricted perfect branch prediction, to predict only the next n branches. Note that 0 predicted branches prevents speculative execution, but some global scheduling is still possible: instructions can be moved down.

In other words, for our perfect model, we considered only read-after-write (RAW) dependences, through both registers and memory, and dependences from branches to all later stores. For limited branch-prediction we added dependences between branches and all instructions that are more than n branches later in the trace. Without delayed exceptions, we also added dependences between branch instructions and all later exception-generating instructions.[5]

Throughout the simulations, we used the latencies of the R3000 processor (e.g. 2-cycle loads). The instruction level parallelism is computed by dividing the number of instructions by the critical path length of the dependence graph.

The benchmark programs used are: an abstract Prolog machine interpreter (Prolog), an instruction scheduler (sched), and compress 4.1 (compress). They were compiled on a DecStation 5000 with the manufacturer's C compiler and then run with typical input to produce traces. Due to limitations of our tracer we produced and analysed only short traces ($\approx 500,000$ instructions).

The results are shown in Fig. 9. Without delayed exceptions, even perfect branch prediction gives only speedups of 1.08–1.31 over having no speculation. This clearly shows that speculative execution is hardly worth bothering, if it cannot be applied to exception-generating instructions (another variation of Amdahl's Law). Even with only one-deep speculation delayed exceptions beat the perfect model without delayed exceptions. In other words: Every machine that has enough resources to profit from speculative execution will profit from delayed exceptions. The perfect models differ by a factor of 3.8 (sched), 7.4 (compress) and 9.5 (Prolog).

The improvement on a realistic machine with a real scheduling algorithm is of course somewhat lower, but still impressive: Mahlke et al. report a speedup of 18%–135% (average 57%) for sentinel scheduling (see Section 7) on non-numeric programs for a superscalar processor of degree 8 [MCH+92]. They report an average speedup of 32% for numeric benchmarks. Delayed exceptions should give similar results.

7 Related Work

Ignoring exceptions by using non-trapping instructions has been proposed for circumventing the problem [CMC+91]. Instead of trapping on e.g. an illegal memory

[5] A compiler could use control-dependence analysis to move instructions up across branches in a non-speculative way. If our simulation took this into account (i.e. did not count those branches), it would result in a somewhat higher instruction-level parallelism for all models but the perfect model. Due to the data given in [LW92] and Amdahl's Law we believe that the effect of this optimization would not be very large and would not change our conclusions.

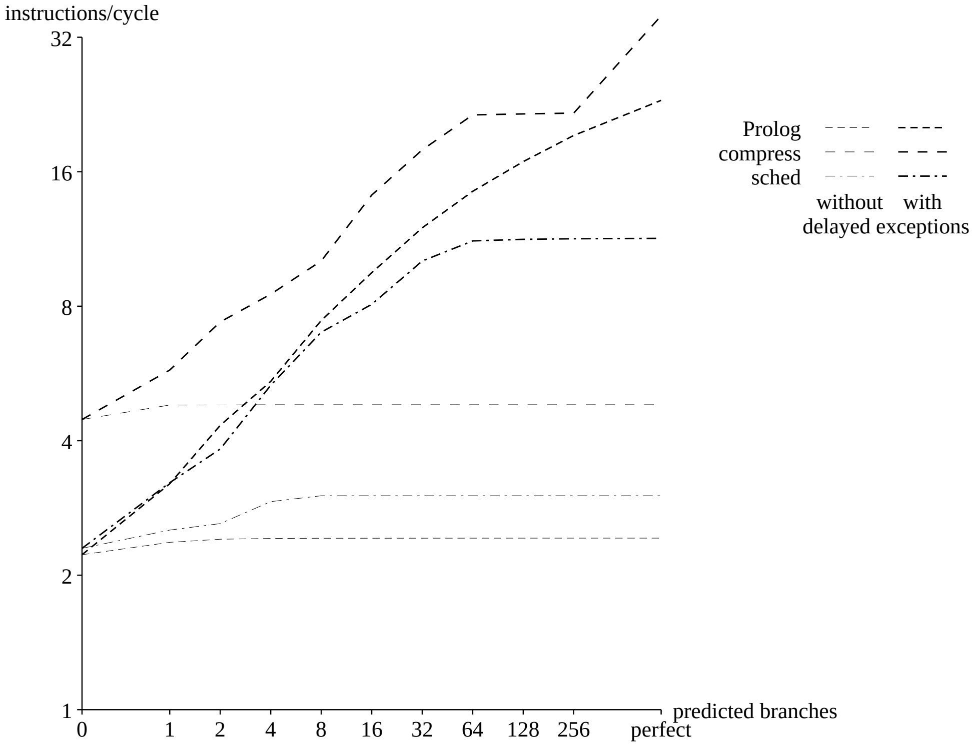

Fig. 9. Instruction-level parallelism with and without delayed exceptions

access, a garbage value is returned. The justification for this behaviour is that correct programs do not trap. Unfortunately this justification is wrong. Exceptions are used in many applications. [AL91] lists several applications of memory access exceptions. Besides, in our opinion the assumption of completely correct programs is unrealistic. Delayed exceptions solve the problem instead of ignoring it.

Speculative loads [RL92] note the exception in a bit associated with the result register. The first instruction that uses the result triggers the trap. This means that the load can be executed speculatively, but not its use. In contrast, delayed exceptions permit arbitrary chains of speculative operations.

The *TORCH* architecture [SLH90, SHL92, Smi92] uses programmer-visible shadow register files for compiler-based speculative execution. TORCH as described in [SLH90] can handle exceptions using a reexecution scheme implemented in hardware. In the meantime they have switched to using compiler-generated recovery code. In contrast to delayed exceptions, TORCH uses hardware to restore the state before the exception and then executes the recovery code. TORCHs recovery code contains all speculative instructions since the exception, while our fix-up code contains only instructions that dependent on the trap-noting instruction.

Sentinel scheduling [MCH+92] uses a bit in every instruction that says whether the instruction is executed speculatively. Speculative instructions set and propagate exception bits; the first non-speculative instruction triggers the trap (if the bit is set). Recovery is performed by reexecuting the whole code starting from the instruction that caused the exception. The bottom-line differences between delayed exceptions and sentinel scheduling are: Sentinel scheduling does not preserve the order of the exceptions; It doubles the number of opcodes, whereas delayed exceptions only double the exception-generating instructions. Sentinel scheduling needs special hardware for propagating the exception bits and the address of the exception-generating instruction; and it produces more register pressure (the source registers for all instructions between speculative instruction and sentinel have to be preserved, while our fix-up code needs only the source registers for instructions that depend on the trap-noting instruction); it cannot move exception-generating instructions beyond irreversible instructions like stores. Delayed exceptions use more instruction bandwidth (for exception-checking branches) and they produce more code (fix-up code).

Write-back suppression [BMH+93] is specific to superblock[6] scheduling. It uses hardware to suppress the register file updates by the excepting instruction and all subsequent instructions that have the same or higher speculation distance. If execution reaches the home basic block of the excepting instruction, these instructions are reexecuted by hardware with write-back and trapping enabled. This mechanism produces less register pressure than delayed exceptions, because the hardware ensures that instructions that will be reexecuted later will not write over registers needed during reexecution. The main disadvantage of write-back suppression is its restriction to superblock scheduling and unidirectional speculation.

8 Conclusion

Speculative execution is the key to making optimal use of superscalar processors. However, in a pure compiler-based approach exception-generating instructions must not be executed speculatively. This restriction virtually eliminates speculative execution and its advantages.

In the spirit of RISC technology, *delayed exceptions* combine a simple hardware extension with sophisticated compiler techniques to solve this problem. Speculative exception-generating instructions just note the exception in a bit associated with the result register. If the speculation was right, the bit is checked by a special branch instruction. If the bit is set (i.e. there was an exception), it branches to compiler-generated fix-up code. This code triggers the trap and ensures correct recovery. Delayed exceptions permit the compiler to move exception-generating instructions as freely as non-trapping instructions.

Trace-driven simulation shows that every processor that profits from speculative execution (e.g. most superscalar processors) would profit from delayed exceptions. The upper bound for speedups achievable by delayed exceptions for the studied traces is 3.8–9.4.

[6] A superblock can be entered only from the top, but can be left at several points.

Acknowledgements

The referees, Thomas Pietsch and Franz Puntigam provided valuable comments on earlier versions of this paper. Section 5.2 was inspired by the "Trapping speculative ops" discussion on the Usenet news group comp.arch, especially by Stanley Chow and Cliff Click.

References

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In ASPLOS-IV [ASP91], pages 96–107.

[ASP91] *Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, 1991.

[ASP92] *Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, 1992.

[BMH+93] Roger A. Bringman, Scott A. Mahlke, Richard E. Hank, John C. Gyllenhaal, and Wen-mei W. Hwu. Speculative execution exception recovery using write-back suppression. In *26th Annual International Symposium on Microarchitecture (MICRO-26)*, pages 214–223, 1993.

[BYP+91] Michael Butler, Tse-Yu Yeh, Yale Patt, Mitch Alsup, Hunter Scales, and Michael Shebanow. Single instruction stream parallelism is greater than two. In ISCA-18 [ISC91], pages 276–286.

[CMC+91] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Warter, and Wen-mei W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In ISCA-18 [ISC91], pages 266–275.

[EK92] M. Anton Ertl and Andreas Krall. Removing antidependences by repairing. Bericht TR 1851-1992-9, Institut für Computersprachen, Technische Universität Wien, 1992.

[Ell85] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1985.

[Fis81] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[HP87] Wen-mei Hwu and Yale N. Patt. Checkpoint repair for high-performance out-of-order execution machines. *IEEE Transactions on Computers*, 36(12):1496–1514, December 1987.

[ISC91] *The* 18^{th} *Annual International Symposium on Computer Architecture (ISCA)*, Toronto, 1991.

[JW89] Norman P. Jouppi and David W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 272–282, 1989.

[LW92] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *The* 19^{th} *Annual International Symposium on Computer Architecture (ISCA)*, pages 46–57, 1992.

[MCH+92] Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In ASPLOS-V [ASP92], pages 238–247.

[NE89] Toshio Nakatani and Kemal Ebcioğlu. "Combining" as a compilation technique for VLIW architectures. In 22^{nd} *Annual International Workshop on Microprogramming and Microarchitecture (MICRO-22)*, pages 43–55, 1989.

[Nic85] Alexandru Nicolau. Uniform parallelism exploitation in ordinary programs. In *1985 International Conference on Parallel Processing*, pages 614–618, 1985.

[Nic89] Alexandru Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.

[RG81] B. R. Rau and C. D. Glaeser. Some scheduling techgniques and an easily schedulable horizontal architecture for high performance scientific computing. In *14th Annual Microprogramming Workshop (MICRO-14)*, pages 183–198, 1981.

[RL92] Anne Rogers and Kai Li. Software support for speculative loads. In ASPLOS-V [ASP92], pages 38–50.

[SHL92] Michael D. Smith, Mark Horowitz, and Monica S. Lam. Efficient superscalar performance through boosting. In ASPLOS-V [ASP92], pages 248–259.

[SLH90] Michael D. Smith, Monica S. Lam, and Mark A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *The* 17^{th} *Annual International Symposium on Computer Architecture (ISCA)*, pages 344–354, 1990.

[Smi92] Michael David Smith. *Support for Speculative Execution in High-Performance Processors*. PhD thesis, Stanford University, 1992.

[Soh90] Gurindar S. Sohi. Instruction issue logic for high-performance, interruptable, multiple functional unit, pipelined processors. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[SP88] James E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[Wal91] David W. Wall. Limits of instruction-level parallelism. In ASPLOS-IV [ASP91], pages 176–188.

This article was processed using the LaTeX macro package with LLNCS style