The Impact of Hardware Scheduling Mechanisms on the Performance and Cost of Processor Designs

S. M. Mueller*, H. W. Leister†, P. Dell, N. Gerteis, D. Kroening†
Dept 14: Computer Science, University of Saarland, Germany
email: {smueller, holy, kroening}@cs.uni-sb.de

To Appear in Proc. 15th GI/ITG Conference ARCS'99.

Abstract

Hardware schedulers supporting out-of-order execution are widespread nowadays. Nevertheless, studies quantifying the impact of schedulers on the performance and cost of processors are rare. The paper tries to close this gap. It turns out that the hardware schedulers can double the performance at a moderate increase (10–24%) in a processor's gate count. Earlier rearranging of instructions *allows* for better performance, but it does *not guarantee* it. The lack of features like forwarding and non-blocking resources can nullify this gain. Despite of its out-of-order dispatch capability, the original Scoreboard scheduler, for example, performs significantly worse than a standard in-order pipeline. The paper also identifies the aspects responsible for this poor performance and quantifies their impact. The single most important aspect is the lack of result forwarding.

Keywords: dynamic hardware scheduling, out-of-order execution, Tomasulo scheduler, Scoreboard, reorder buffer, cost-performance analysis.

1 Introduction

Current processors comprise multiple functional units which can work in parallel. The latencies of the functional units vary by a lot. For a better performance and hardware utilization, many designs therefore allow the rearranging of instructions (*out-of-order execution*). The instruction execution and the resources are then governed dynamically by hardware schedulers, most of which are based on the result shift register [26], the Scoreboard [28], and the Tomasulo Algorithm [29]. One major difference between the schedulers is how early in the processing they allow instructions to leave the in-order execution. In case of *out-of-order completion*, the instructions and their operands are passed to the functional units (FU) in-order, but the results might be provided in a different order due to the non-uniform latencies of the FUs. Whereas with *out-of-order dispatch*, the execution of the operation in an FU is already started out-of-order.

Since hardware schedulers are widespread nowadays [2, 5, 7, 9, 10, 11, 24, 27], one would aspect plenty of studies which quantify the impact of the different type of schedulers on the performance and cost (gate count) of processors. However, the open literature [16, 12, 26] rather focus on the impact of design changes. For a given scheduler, they study the speedup gained by varying the degree of superscalarity, the forwarding mechanism, the number of FUs, or the buffer size. Comparisons across schedulers are mostly qualitative. However, one gets the following notion:

Claim 1 *The more flexible the execution of the instructions is, the higher is the performance improvement of the scheduler.*

Thus, it is tempting to believe that out-of-order dispatch is more powerful than out-of-order completion, and that both concepts are more powerful than in-order execution. This paper falsifies this hypothesis. Earlier rearranging of instructions *allows* for better performance, but it does *not guarantee* it. Despite of its out-of-order dispatch capability, the original Scoreboard scheduler, for example, performs worse than a standard in-order pipeline. This suggests that there are scheduler features with a much stronger performance impact. The paper also identifies these features and quantifies the factor to be gained in performance and cost.

2 Hardware Schedulers

Due to lack of space, detailed specifications of the schedulers will be omitted; those can be found in the literature. We rather focus on the underlying execution schemes, instead. Wherever necessary, we also describe the key features and the characteristics of the schedulers.

* partially supported by the German Science Foundation DFG
† supported by the DFG graduate program 'Effizienz und Komplexität von Algorithmen und Rechenanlagen'

fetch F	issue I	dispatch D	execute E	complete C	terminate T
in-order		possibly out-of-order			in-order

Table 1 Phases of the instruction execution.

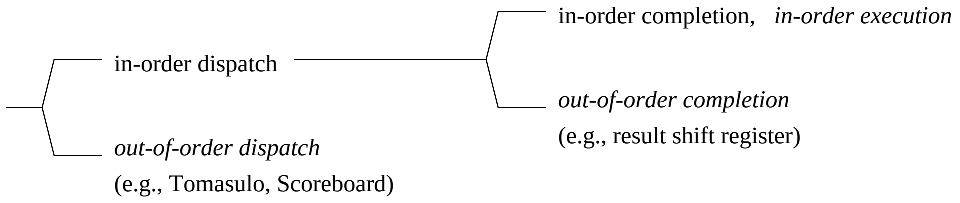
in-order dispatch
out-of-order dispatch
(e.g., Tomasulo, Scoreboard)
in-order completion, in-order execution
out-of-order completion
(e.g., result shift register)

Fig. 1 Classification of execution schemes

2.1 Classification

All our schedulers partition the instruction execution in a similar way (**Table 1**). After fetching, the instruction is *issued*, i.e., it is decoded and assigned to a functional unit (FU). The instruction and its operands are then sent (*dispatched*) to the FU which performs the operation. The *execution* includes the data memory access. Once the operation is *completed*, the result is buffered, and it is provided to later instructions. During the last phase (*terminate*), the exception check and the actual update are performed.

Fetch, issue, and termination are always performed in program order. The in-order termination, for example, is essential for the precise interrupt handling and the rollback on misspeculation. Thus, only the dispatch, execution, and completion of instructions can be performed out-of-order. A major difference between the schedulers is how early in the processing they allow instructions to leave the in-order execution. We therefore classify them according to the first stage in which instructions can be rearranging, i.e., we classify them as *in-order execution*, *out-of-order completion*, or *out-of-order dispatch* (**Fig 1**).

2.1.1 In-Order Execution

In the in-order execution, which serves as the reference model, the instructions are processed in a pipelined fashion. The functional units are pipelined as well. Issue and dispatch are performed in a single cycle; the same holds for the phases complete and terminate. This scheme is mainly used in early RISC processor series and in processors aimed for the embedded (fixed-point) marked, like the PA-7200 [8], the MIPS R4300i [19], or the StrongARM [4].

For single cycle operations, RAW (read after write) data dependences can be resolved by result forwarding without stalling the pipeline. For operations with a longer latency that is not always possible. Thus, if an instruction depends on the result of such a slow operation, it must be stalled until the result is available.

Despite of the non-uniform latency of the FUs, the instructions are executed strictly in program order. This can, for example, be enforced by a result shift register [26]. The dispatch of a low latency instruction I_i is also delayed if it follows a high latency instruction I_j.

Thus, data dependences and slow operations are two major sources of performance degradation, as illustrated in **Fig 2**. If I_i and I_j have a latency of l_i and l_j, respectively, I_i must be stalled for $l_j - l_i$ cycles. More elaborate schedulers try to reduce this degradation by out-of-order execution.

2.1.2 Out-of-Order Completion

Under this execution scheme, the instructions and their operands are dispatched to the FUs in-order, but the results might be delivered in a different order due to the non-uniform latencies of the FUs. The results are then buffered in the so called reorder buffer (ROB) [26] until they can be written back in program order. Thus, the REOB realizes the in-order termination.

In the example of Fig 2, the second instruction is then started and completed two cycles earlier than under in-order execution. Since the results buffered in the ROB can be forwarded, out-of-order completion also speeds up depending instructions. Thus, this scheme compensates for slow FUs. It is used in the UltraSparc processor series [10, 27] designed for the high-end server market.

However, there are also constraints to out-of-order completion, namely that at most one instruction is completed per cycle and that RAW dependences are obeyed. Thus, the instruction dispatch must still be stalled if the operands of an instruction are not available, or if the instructions would compete for the result bus. This scheduling can be implemented by a result shift register in combination with a ROB [26].

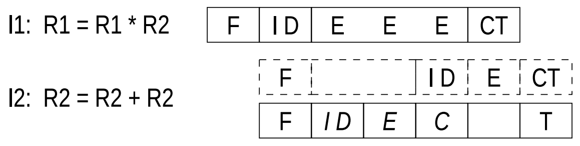

Fig. 2 Illustration of the out-of-order completion execution scheme. The dashed box indicates the in-order execution schedule.

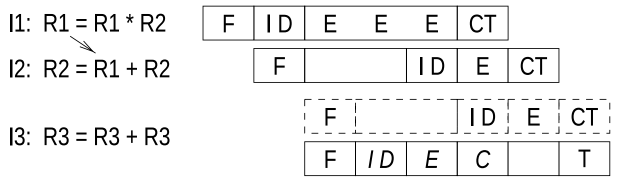

Fig. 3 Illustration of the out-of-order dispatch execution scheme. The dashed box indicates the out-of-order completion schedule.

2.1.3 Out-of-Order Dispatch

Under out-of-order completion, instructions can overtake another while they are processed in the FUs, whereas under out-of-order dispatch, the processing in the FUs can already be started out-of-order. Under out-of-order dispatch, an instruction which waits for its operands no longer blocks the instruction dispatch, it rather steps aside and lets later instructions proceed. Thus, an instruction can be dispatched once its operands and its FU become available. Out-of-order dispatch therefore compensates for slow FUs and for RAW dependences which cannot be resolved by forwarding alone.
In the example of **Fig 3**, instruction I_2 is stalled for two cycles because it depends on the result of I_1. Under out-of-order completion, I_3 must be stalled as well, although it is data independent. When splitting issue and dispatch, I_3 can be dispatched and completed two cycles earlier.
The Scoreboard [28, 22] and the Tomasulo scheduler [29, 12] are the two classical schedulers supporting out-of-order dispatch. Both schedulers require complex data structures. The Tomasulo scheduler is considered to be more powerful due to its renaming capabilities; it is widely used in the high performance microprocessors like the PowerPC family [9], the P6 family [11], the MIPS R10000, the AMD K5, and the AMD 29K [24, 2]. A pure Scoreboard scheduler is rarely used nowadays; Motorola MC88100, Intel i860 [1], and Alpha 21164 [5] apply a modified Scoreboard scheduler.

2.2 Performance Hypothesis

The two out-of-order dispatch schedulers can rearrange instructions earlier than under the out-of-order completion scheme, and should therefore be more flexible. Under in-order execution, the instructions cannot be rearranged at all. According to Claim 1, out-of-order dispatch is more powerful than out-of-order completion (OC), and both schemes are more powerful than in-order execution (IN). For the runtime of a given workload, this implies

$$T_{TOM} \leq T_{SCB} \leq T_{OC} \leq T_{IN}, \qquad (1)$$

where SCB and TOM denote the out-of-order dispatch schemes with Scoreboard or Tomasulo scheduler, respectively.
In the following, we check this hypothesis and quantify the factor to be gained in cost and performance when switching from one scheduler to another.

3 Methodology of the Comparison

The different schedulers are compared based on their performance and gate count. Since both quantities depend on the structure of the processor, the schedulers need to be integrated in a processor design. We are interested on the sole impact of the scheduler. The underlying processor must therefore be the same, at least with respect to the instruction set, the functional units, the memory system, and the issue bandwidth.

Processor Model We use a single-issue DLX core [12, 21]; except for the divider, all functional units (**Table 2**) are fully pipelined. The original Scoreboard scheduler does not support pipelined FUs, reducing significantly the computational power of the design. In the Scoreboard design, all the FUs are therefore duplicated, except for the floating-point divider.
All our design variants support precise interrupt handling, which relieson in-order termination. Except for

functional unit	fixed-point		floating-point				
	ALU	addr	add	mul	div	test	convert
latency	1	1	5	5	15	1	4
number in IN, OC, TOM	1	1	1	1	1	1	1
number in SCB	2	2	2	2	1	2	2

Table 2 Set of functional units

	not	nand, nor	and, or	xor, xnor	flipflop	mux	3-state driver
cost	1	2	2	4	8	3	5
delay	1	1	2	2	4	2	2

small on-chip RAM with $A \leq 64$ entrees of width n			
cost	$2 \cdot (A + 3) \cdot (n + \log \log n)$	delay	$\log n + A/4$

Table 3 Cost (gate equivalents) and delay of the basic components

the in-order varian IN, all designs thereforeuse a re-order buffer.

The hierarchical memory subsystem is modeled after the Pentium system [15] with an Intel PCIset [14]. It includes two 8 Kbyte primary caches and a unified 512 Kbyte secondary cache. All three caches are 2-way set associative with LRU replacement. The transfer of a cache line between the caches requires 11 cycles, whereas the transfer between secondary cache and main memory requires 18 cycles.

Hardware Model For a quantitative comparison of the different design variants, we at least need a rough estimate of their gate counts and cycle times. When specifying the designs at gate level, the hardware model of [21] permits to derive these values.

In this model, the hardware is constructed from basic components, like gates, flipflops, RAMs, and 3-state drivers. Their cost (in gate equivalents) and propagation delay can be extracted from any design system, but they are highly technology dependent. **Table 3** lists these parameters for Motorola's H4C CMOS sea-of-gate design series [23]; they are normalized relative to the cost and delay of an inverter.

The cost (gate count) of the entire hardware is then the cumulative cost of all its basic components. The cost of the off-chip main memory is usually ignored. The cycle time is the maximal delay of all paths through the hardware. Based on macro libraries [21, 20], an entire processor can be specified in a hierarchical and recursive manner with reasonable effort.

Quality Model Except for the variant with Scoreboard scheduler, all the design variants are specified on gate level; gate counts and cycle times are determined within the framework outlined above. Since caches usually consume a major portion of the chip area [25], we consider two types of gate counts: the gate count of the sole CPU core and the gate count of the processor including primary caches. The entire designs, their gate counts, and cycle times are given in [17, 18, 20].

For all the design variants, the cycle time is virtually the same; it is dominated by the IEEE compliant floating point unit. Thus, their performance can be modeled by the CPI ratio on a given workload, which in our case are the SPEC-92 benchmarks (**Table 4**). The CPI ratios are obtained by trace-driven simulations [3, 6]. The traces are taken from [13].

4 The Impact of Rearranging

4.1 Performance Impact

Tables 5 and **6** list the CPI ratio of all SPEC-92 benchmarks under the four execution schemes. These data indicate that out-of-order completion reduces the CPI ratio and the runtime by about 20% over in-order execution. Out-of-order dispatch with a Tomasulo scheduler gains an additional 15%. For the tomcatv.047 benchmark, the CPI is even improved by a factor of 2.6; whereas on the benchmarks espresso.008, li.022 and alvin.052, the out-of-order execution only gains a speedup of 8 to 14%. Altogether, these results conform with the hypothesis of equation (1) and with Claim 1, but for the Scoreboard the matter is quite different.

The Scoreboard performs poorly: it is between 1.9 and 2.8 times slower than the Tomasulo scheduler; the average slowdown is 2.2. Despite of its out-of-order dispatch capability, the Scoreboard scheduler causes a 70% higher CPI ratio than under the out-of-order completion scheme. On most benchmarks, it is even significantly slower than the in-order execution. The only exception is the tomcatv.047 benchmark, where it achieves a 15% speedup. This benchmark comprises 34% multi-cycle operations, that are 2.5 as many as in the average floating-point workload.

integer		floating point			
ID	Name	ID	Name	ID	Name
008	espresso	015	doduc	056	ear
022	li	034	mdljdp2	077	mdljsp2
023	eqntott	039	wave5	078	swm256
026	compress	047	tomcatv	089	suc2cor
085	gcc	048	ora	093	nasa7
		052	alvin	094	fppp

Table 4 SPEC-92 Benchmark Suite

benchmark		008	022	023	026	085	average
absolute	IN	1.2	1.4	1.5	2.6	1.9	1.7
	OC	1.1	1.2	1.3	2.4	1.3	1.5
	TOM	1.0	1.2	1.1	1.2	1.6	1.2
	SCB	2.8	2.7	2.8	2.8	3.1	2.8
relative to IN	OC	0.92	0.86	0.87	0.92	0.68	0.85
	TOM	0.83	0.86	0.73	0.46	0.84	0.71
	SCB	2.33	1.93	1.87	1.08	1.63	1.65

Table 5 CPI ratio of the SPEC-92 integer benchmarks (absolute and relative to in-order execution)

Thus, the original performance hypothesis summarized in Equation (1) is wrong; for most benchmarks, it holds

$$T_{TOM} \leq T_{OC} \leq T_{IN} \leq T_{SCB}.$$

Note that this does not falsify Claim 1, it just suggests that the schedulers implement additional features which have a stronger impact on the performance than the early rearranging of instructions. Before identifying these features, we first quantify the cost impact of the other three schedulers.

4.2 Cost-Performance Analysis

Table 7 lists the gate count values of the three design variants and of their major hardware components. In the in-order variant IN, the scheduler almost comes for free, whereas out-of-order execution increases the cost of the scheduler significantly. Our out-of-order completion scheduler (OC) increases the cost of the CPU core by 44%, and the Tomasulo scheduler even doubles the cost of the CPU core. However, for the entire processor, i.e., core with primary caches, the cost increase is quite moderate, between 11% and 24%. This is due to the expensive caches.
The hardware support for the out-of-order completion is worth-while. Compared to the design with in-order execution, the performance is improved by 20% at a 11% higher gate count. The Tomasulo scheduler is also cost efficient; it achieves an additional 15% performance improvement at a 13% bigger chip size.
The original Scoreboard scheduler is not competitive. It performs worse than the other three design variants, and especially worse than the in-order pipeline. Due to the duplicated functional units and its complex scheduling hardware, the Scoreboard design is definitely more expensive and less cost efficient than the in-order pipeline.
Thus, it stands to reason that the Tomasulo scheduler is widely used in current processors, whereas the Scoreboard in its original form is rarely used.

5 The Impact of Further Scheduler Features

In the following, we identify the features which allow the design variants IN, OC, and TOM to outperform the variant with the Scoreboard scheduler.

5.1 Characteristics of the Schedulers

Scoreboard Scheduler In this study, we consider the original Scoreboard of [28, 12] but with the corrections presented in [22]. It has the following key features: The instruction processing is split into the standard six phases (Table 1), each of which is assigned to a separate cycle.
For each FU, the global data structure of the Scoreboard provides one entry which keeps track of the instruction to be executed by the FU. The scheduler makes bad use of pipelined function units, because it supports only *one* outstanding instruction per unit. If subsequent instructions require the same FU, they are executed sequentially.

benchmark		015	034	039	047	048	052	056	077	078	089	093	094	av
abs	IN	1.8	1.7	1.5	4.4	2.2	1.2	1.9	1.6	2.4	3.1	2.4	2.8	2.2
	OC	1.3	1.5	1.5	3.2	2.0	1.1	1.8	1.4	2.3	2.1	1.9	2.1	1.9
	TOM	1.3	1.3	1.2	1.7	1.4	1.1	1.5	1.2	1.7	1.9	1.5	1.8	1.5
	SCB	2.8	3.0	2.3	3.8	3.3	2.7	3.4	2.6	3.3	3.6	2.8	3.7	3.1
rel	OC	0.72	0.88	1.00	0.73	0.91	0.92	0.95	0.87	0.96	0.68	0.79	0.75	0.82
	TOM	0.72	0.76	0.80	0.39	0.64	0.92	0.79	0.75	0.71	0.61	0.62	0.64	0.65
	SCB	1.56	1.76	1.53	0.86	1.50	2.25	1.79	1.62	1.38	1.16	1.17	1.32	1.38

Table 6 CPI ratio of the SPEC-92 floating-point benchmarks (absolute and relative to in-order execution); 'av' denotes the average CPI ratio.

	CPU						+ 16KB Cache		
	FUs	reg files	scheduler	misc	total		cache	total	
IN	93 kG	20 kG	3 kG	2 kG	118 kG		375 kG	493 kG	
OC	93 kG	20 kG	55 kG	2 kG	170 kG	(144%)	375 kG	545 kG	(113%)
TOM	93 kG	20 kG	121 kG	2 kG	236 kG	(200%)	375 kG	611 kG	(126%)

Table 7 Gate count (in kilo Gates) of the three designs with in-order execution (IN), out-of-order completion (OC), or Tomasulo out-of-order dispatch scheduler (TOM).

The issuing of an instruction is also stalled on false data dependences, i.e.:

- on a WAW (write after write) dependence, the destination register R of the instruction is still reserved by another instruction, and
- on a WAR (write after read) dependence, the old value of R must still be read.

This constraint becomes even more severe due to the late operand fetch. All operands of an instruction are read in the same cycle and only from the register files; result forwarding is not supported.

Tomasulo Scheduler [29, 6] is much more flexible because it supports result forwarding, and it allows for several outstanding instructions per FU and destination register. For each FU, multiple instruction buffers are provided; these buffers are known as reservation stations. The scheduler can therefore make full use of the pipelined functional units. During instruction issue, the destination register is renamed, resolving all WAW and WAR dependences. Thus, the issuing is no longer stalled on false data dependences.

In-Order Dispatch In contrast to the Scoreboard scheduler, the in-order execution (IN) and the out-of-order completion (OC) schemes both dispatch the instructions in program order, i.e., the possible rearranging is delayed to later stages. However, due to the in-order dispatch, a write to register R never overtakes a preceding read of R. Thus, there is no need to stall the issuing on false data dependences.

In addition, both schemes, IN and OC, support result forwarding and make use of the pipelined functional units. Since issue and dispatch are now performed in the same cycle, the depth of the pipeline is also reduced. Note that in the in-order pipeline, completion and termination are performed in a single cycle, but there is an additional stage for the data memory accesses.

Common Features The three schedulers implementing the schemes IN, OC, and TOM have the following features in common. They support result forwarding, they support multiple outstanding instructions for each destination register, they make use of pipelined functional units, and they never stall on false data dependences. The original Scoreboard scheduler lacks all of these features.

5.2 Scoreboard vs. In-Order Dispatch

The CPI ratios of espresso.008 and tomcatv.047 suggest that in general, the Scoreboard is not more flexible than the two schemes IN and OC with in-order dispatch, nor vice versa. That is because features like result forwarding and the non-blocking use of resources are traded for the early rearranging of instructions.

There are situations where out-of-order dispatch cannot compensate for the lack of these features, and therefore, the Scoreboard can only improve the performance if the instructions are distributed over the FUs and the registers. Two of these situations are illustrated in **Figs 4** and **5**.

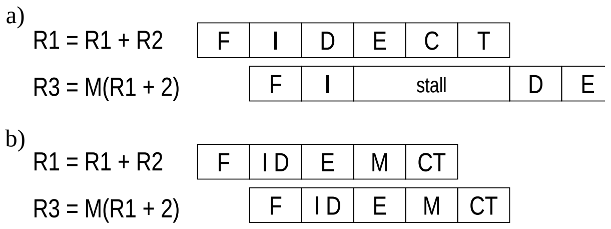

Fig. 4 The impact of a RAW data dependence in a design with Scoreboard (a) and with an in-order pipeline (b)

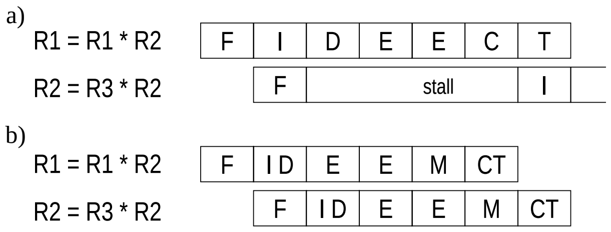

Fig. 5 Blocking (a) and non-blocking (b) use of a pipelined FU

The first example illustrates the advantage of result forwarding. The address computation of the load requires the result of the addition. Due to result forwarding, the in-order pipeline can execute both instructions at full speed. Whereas with the Scoreboard, the load cannot be dispatched before the first instruction has updated the register file. Thus, the Scoreboard looses three cycles over the in-order execution.

The example of Fig 5 illustrates the advantage of the non-blocking use of a pipelined FU. The two instructions are data independent, but they require the same FU. Since the multiplier is fully pipelined, the standard in-order pipeline can process both instructions at full speed. Since the Scoreboard only provides one instruction buffer per FU, it blocks the multiplier during issue and frees it during completion. Thus, the issuing of the second multiplication must be delayed for several cycles.

5.3 Scoreboard vs. Tomasulo Scheduler

Both schedulers support out-of-order dispatch, but the Tomasulo scheduler also supports result forwarding, register renaming, and provides multiple instructions buffers for each FU. In order to quantify the impact of these features, we extend the Scoreboard scheduler step by step (**Table 8**). Such an extended Scoreboard scheduler can, for example, be found in the Alpha 21164 processor [5]. Its scheduler is based on the Scoreboard, but it supports result forwarding, and except for the integer multiplier and the floating-point divider, the FUs are operated in a non-blocking pipelined fashion.

Result Forwarding First, a common data bus (CDB) is added; the CDB is used for result forwarding. An instruction now reads its operands as soon as they get valid and not necessarily all in the same cycle. This has two advantages: Since the results are read earlier, the instruction can also be dispatched earlier. Second, issuing is still stalled on false data dependences, but WAR dependences are now resolved faster.

The CPI ratio of the original Scoreboard scheduler is about 120% higher than the one of the Tomasulo scheduler, whereas the Scoreboard with forwarding is only 40% slower than the Tomasulo scheduler. Thus, result forwarding already eliminates two thirds of the additional cycles. This improved Scoreboard scheduler achieves a similar performance as the in-order pipeline.

Non-Blocking Resources In addition, our third Scoreboard version provides multiple instruction buffers for each type of FU, and it supports register renaming. Thus, the issuing is no longer stalled due to false data dependences or blocked FUs. Several instructions with the same destination register can now be processed simultaneously. Note that the functional units are not pipelined yet, i.e., any unit processes at most one instruction at a time. These extensions reduce the slowdown from 40% to about 6%.

6 Conclusion

Earlier rearranging of instructions *allows* for better performance, but it does *not guarantee* it. It is also essential for the performance of a scheduler how well the data dependences are resolved (e.g., by stalling, for-

	forwarding	instructions / FU, register	absolute CPI		relative CPI	
			INT	FP	INT	FP
Pipeline	yes	multiple	1.71	2.24	1.4	1.5
Scoreboard	no	1	2.83	3.11	2.3	2.1
+ forwarding	yes	1	1.76	2.06	1.4	1.4
+ non-blocking	yes	multiple	1.28	1.58	1.05	1.07
Tomasulo	yes	multiple	1.22	1.46	1.0	1.0

Table 8 CPI ratio of the extended Scoreboard schedulers under an average integer or floating-point SPEC-92 workload. The last two columns list the CPI relative to the one of the Tomasulo scheduler.

warding, and renaming) and whether the resources are used in a blocking or non-blocking fashion.

On average, out-of-order completion reduces the CPI by 20%, and out-of-order dispatch reduces it by another 15%. However, the lack of forwarding and the blocking of resources can nullify this gain, as it happens for the original Scoreboard scheduler. Its CPI ratio is about 120% higher than the one of the Tomasulo scheduler. Two thirds of these additional cycles are due to the lack of result forwarding. The remaining slowdown is largely caused by the blocking use of resources. Thus, it stands to reason that the Tomasulo scheduler is widely used in high performance processors, whereas the Scoreboard in its original form is not used.

The schedulers supporting out-of-order execution increase the gate count of the CPU core between 44 and 100%. However, with respect to the whole processor, i.e., CPU core plus onchip caches, the increase of the gate count is far more moderate (11 to 24%). Thus, out-of-order execution combined with forwarding and non-blocking resources is worth-while, except if the gate count is crucial like in embedded systems.

Ongoing research includes extending these studies to superscalar designs with speculation and predication.

References

[1] A. Bode, editor. RISC-Architekturen. BI-Wissenschaftsverlag, 1988.

[2] B. Case. AMD unveils first superscalar 29K core. Microprocessor Report, 8(14):23–26, 1994.

[3] P. Dell. The performance impact of the standard mechanisms for out-of-order dispatch on a RISC processor (in German). Master's thesis, University of Saarland, Computer Science Department, Germany, 7/1998.

[4] Digital Semiconductor. StrongARM Microprocessors: SA-110 Microprocessor. Data Sheet, Rev 24 July, 1997.

[5] J.H. Edmondson, P. Rubinfeld, and R. Preston. Superscalar instruction execution in the 21164 Alpha microprocessor. IEEE Micro, 15(2):33–43, 1995.

[6] N. Gerteis. The performance impact of precise interrupt handling on a RISC processor (in German). Master's thesis, University of Saarland, Computer Science Department, Germany, 4/1998.

[7] L. Gwennap. MIPS R10000 uses decoupled architecture. Microprocessor Report, 8(14):18–22, 1994.

[8] L. Gwennap. PA-7200 enables inexpensive MP systems. Microprocessor Report, 8(3):12–15, 1994.

[9] L. Gwennap. PPC 604 powers past Pentium. Microprocessor Report, 8(5):1, 6–9, 1994.

[10] L. Gwennap. UltraSparc unleashes SPARC performance. Microprocessor Report, 8(13):1, 6–9, 1994.

[11] L. Gwennap. Intel's P6 uses decoupled superscalar design. Microprocessor Report, 9(2):9–15, 1995.

[12] J.L. Hennessy and D.A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, INC., San Mateo, CA, 2nd edition, 1996.

[13] M. Hill. SPEC92 Traces for MIPS R2000/3000. University of Wisconsin, 1995, "`ftp://ftp.cs.newcastle.edu.au/pub/r3000-traces/din`".

[14] Intel Corporation. 82430FX PCIset Datasheet 82437FX System Controller (TSC) and 82438FX Data Path Unit (TDP), 1995.

[15] Intel Corporation. Pentium Processor Family Developer's Manual, Vol. 1-3, 1995.

[16] W. Johnson. Superscalar Microprocessor Design. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[17] D. Kroening. Design and evaluation of a RISC processor with a Tomasulo scheduler. Master's thesis, University of Saarland, Computer Science Department, Germany, 2/1999.

[18] H. Leister. Quantitative analysis of precise interrupt mechanisms for processors with out-of-order execution. Ph.D. Thesis, Preliminary Version, University of Saarland, Computer Science Department, Germany, 9/1999.

[19] MIPS Technologies, Inc. R4300i Microprocessor — R4300 Data Sheet. Rev 0.3, April, 1997.

[20] S.M. Mueller and W.J. Paul. The Complexity of Simple Computer Architectures II, 1999. Monograph Draft. Email: {smueller, wjp}@cs.uni-sb.de.

[21] S.M. Mueller and W.J. Paul. The Complexity of Simple Computer Architectures. Lecture Notes in Computer Science 995. Springer, 1995.

[22] S.M. Mueller and W.J. Paul. Making the original scoreboard mechanism deadlock free. In Proc. 4th Israel Symposium on Theory of Computing and Systems (ISTCS), pages 92–99. IEEE Computer Society, 1996.

[23] C. Nakata and J. Brock. H4C Series: Design Reference Guide. CAD, 0.7 Micron L_{eff}. Motorola Ltd., 1993. Preliminary.

[24] M. Slater. AMD's K5 designed to outrun Pentium. Microprocessor Report, 8(14):1, 6–11, 1994.

[25] M. Slater. The microprocessor today. IEEE Micro, 16(6):32–44, 1996.

[26] J.E. Smith and A.R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. IEEE Transactions on Computers, 37(5):562–573, 1988.

[27] P. Song. UltraSparc-3 aims at MP servers. Microprocessor Report, 11(14):29–34, 1997.

[28] J.E. Thornton. Design of a Computer: The Control Data 6600. Scott Foresman, Glenview, Ill, 1970.

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