CHRONO-SCHEDULING: A SIMPLIFIED DYNAMIC SCHEDULING ALGORITHM FOR TIMING PREDICTABLE PROCESSORS

F. DIAZ-DEL-RIO*, J. L. SEVILLANO, S. VICENTE,
G. JIMENEZ-MORENO and A. CIVIT-BALCELLS

Escuela Técnica Superior de Ingeniería Informática,
Universidad de Sevilla, Avda. Reina Mercedes s/n
41012 Sevilla, Spain
**fdiaz@atc.us.es*

We propose a simpler and latency-reduced instruction scheduler, called chrono-scheduling algorithm, which avoids large and difficult instruction wake-up in order to reduce power consumption and latencies. The key idea of this scheduler is to extract and record all possible information about the future execution of an instruction during its issue, so as not to look for this information again and again during wait stages at the reservation stations. Therefore, an instruction can be issued with the information about at what cycle its operands must be captured and when it must be executed. The first implementation is targeted to processors that have constant latencies like many embedded microcontrollers, most vector processors without data cache, etc. Its main advantages are: no tags, no renaming, and much simpler waiting stations. When com-pared with classical dynamic schedulers, chrono-scheduling provides approximately the same CPI but with simpler overall circuitry and presumably higher clock speed (mainly because of its simplified stations).

Keywords: Computer architecture; instruction level parallelism; dynamic scheduling; reservation stations; reservation tables; time-predictable processors.

1. Introduction

Dynamic scheduling plays a fundamental role in the extraction of instruction level parallelism (ILP) from a serial instruction stream. It is an unquestionable fact that the inclusion of one of these techniques in a pipelined processor boosts its performance. Conventional dynamic schedulers designs are based on some kind of stations (namely "Instruction Queue", "issue window", or "Reservation Stations" (RS) according to prior work[1]), where instructions "wait" to be "woken-up" when all their data and structural dependences are resolved. Then, instruction is sent to its corresponding Functional Unit (FU) to be out-of-order executed. Therefore, the information needed to check if a preceding instruction is ready to execute is distributed among stations (the first known proposal of this type of distributed

schedulers is Ref. 1). On the other hand, some prior dynamic schedulers[2] were based on the idea of centralizing the information required to decide what instructions might wake up at each cycle. An alternative scheduler is that based on register renaming. Here register operand values are not part of the RS queue, which maintains only the information about input registers readiness. A good summary of classical schedulers can be found in some computer architecture books[3] or in some papers.[4,5]

In an out-of-order engine, the instruction scheduler is responsible for dispatching instructions to execution units based on dependencies, latencies, and resource availability. Therefore, the resultant execution order matches that of the data flow, at least for the group of instructions hold in the stations, that is, the issue window. In this paper, we will use for these stations the classical name "Reservation Stations" (RS) due to Tomasulo.[1] In these dynamic schedulers, RSs are implemented using a monolithic CAM (Content Addressable Memory). On the whole, an issue window is a complex multiported structure that incorporates comparators and data forwarding, wake logic (to identify which instructions are ready for execution) and additional logic for selecting ready instructions.[6]

Several circuit level studies[7] have shown that the scheduler CAM logic dominates the latency of a pipelined processor, and therefore the window size cannot be increased without slowing scheduler clock speed, because wake-up and select operations are not easily pipelined in conventional designs. Moreover, some variants or new proposals are still being suggested, in order to simplify tag-associated circuitry[8] or to avoid this CAM-based wake-up method.[9] Other studies point out that the performance of the scheduler can be improved by decreasing the number of tag comparisons necessary to schedule instructions.[8] In addition, the high complexity of dynamic schedulers implies that a significant fraction of the total CPU power dissipation (often as much as 25%) is expended within the RS.[6,7] What is more relevant: the main sources of power dissipation of a scheduler are those related to associative matching and selection logic. Particularly, those major dissipation sources are[6]: (a) locating a free entry associatively and writing into this selected entry; (b) the associative matching done at the tag comparators to pick up forwarded data; (c) arbitrating for the FU, enabling winning instructions for execute and reading the selected instructions information.

A kind of processors where dynamic scheduling may play a decisive role in the next future is that of advanced embedded processors. In order to increment performance, some embedded processors are implementing deeper pipelines to boost frequency, which usually yield to longer latencies and to an increase of the CPI (clock Cycles Per Instruction). For example, an evolutionary case is the ARM family[10]: first versions had 3 stages, afterwards they had 5 or more (the recent ARMv6 is designed for 7 stages).

As a result, reducing CPI is a key feature for the design of such high range micro-controllers. On the whole, it is well known that embedded processors designs are taking more and more features of high performance processors designs. The

inclusion of some kind of dynamic scheduling to avoid stalls is, together with reducing branch penalties, a crucial point to decrease CPI. However, due to high latency, complexity, high consumption of power, etc., embedded processors usually avoid implementation of classical dynamic scheduling algorithms.

Another extreme example is necessarily found in vector cores and processors: for instance the first implementation of vector instructions in VIRAM processor[11] are deeply pipelined (15 stages) in order to hide memory latency. For those aggressive vector machines, it has been calculated that some kind of dynamic scheduling (in addition to compiler optimizations) will shrink CPI of arithmetic and memory access instructions for vector machines: it may provide a speedup of 1.24–1.72.[12] Nevertheless, the truth is that vector processors also avoid dynamic scheduling; the reason of this lack is unclear.[12]

Having in mind all stated above, it is not surprising that in the last few years, one of the main research topics in both embedded and general-purpose computing has been not only to achieve the desired target performance, but also to deliver it with power efficiency. Many embedded systems demand more and more performance at low power dissipations (e.g., those in small phones or PDAs).

In this paper, we propose the chrono-scheduling algorithm (CS): a simpler and latency reduced scheduler, where large and difficult instruction wake-up is avoided, while preserving the same expected CPI. This first simplified implementation is targeted to processors that have constant latencies. This is the case of many embedded microcontrollers, most vector processors without data cache[12] and some special designs with a single level of memory hierarchy, like those based on IRAM.[11] As timing predictability is a desirable property of real time systems, instruction latencies are fixed in many of them.[13] Moreover we will see that chrono-schedulers can take advantage of another interesting feature: timing estimation can be obtained easier than in classical schedulers.

This paper is structured as follows. In Sec. 2, a broad and comprehensive description of a chrono-scheduling (CS) implementation is summarized; causes of structural stalls are detailed in the following section, and finally conclusions are summarized. Along this work we use a set of acronyms and variables that are summarized in Fig. 1.

2. Chrono-Scheduling Architecture

Classical schedulers must examine repeatedly a waiting instruction for wake-up till it can be issued. The disadvantages of this repetitive examination have been previously observed by other authors.[7,14] On the other hand, the key idea of CS is to extract and record all possible information about the future execution of an instruction during its issue, so as not to look for this information again and again during wait stages in the RS. When an instruction issues, it can take with it all this information, that is, at what cycle its operands must be captured and when it must be executed. So we are in some way centralizing the distributed

ACRONYM	MEANING
BRT	Binary Reservation Table
CDB	Common Data Bus
CS	Chrono-scheduling
FU	Functional Unit
HS	Hold Station
IF IS EX WB	Stages in a classic al Tomasulo processor pipeline: Instruction Fetch, Instruction decode and issue, Execution and Write Back
K_{UF}	Number of bits that can be explored into a BRT in a period
L_{UF}	Full latency (including WB phase) of a FU
m	Issue width
N	Number of cycles elapsed between the "producer" and the "consumer" instructions
n_{HS}	Number of HSs
RF	RegisterFile
RP	Relative Period
RS	Reservation Station
SS	Shift Station
Td	RP of a destination register (when it will be generated)
T_{EX}	First period where a FU is free, and EX stage can begin
Ts	RP of a source register (when it can be captured)
T_{WB}	First period where a CDB is free, and WB stage can begin

Fig. 1. List of acronyms and variables (in alphabetical order).

extraction of information (done at RSs in classical algorithms) into the IS (issue) stage, while keeping operands distributed. In this sense, CS shares some aspects with distributed algorithms[1] (operands are distributed to avoid WAR hazards), but it goes further because it extracts and records timing information at IS stage. It is necessary that duration of the execution phase is predictable at IS stage to extract timing information, which is usual in real time processors (in fact this is a desirable feature for these processors, in order to facilitate execution time estimation of a task or a block of code). As structural dependences usually have a known length (they depend on the number of clocks that a resource is occupied), they can also be chrono-scheduled as well as data dependences. Timing extraction has been already used at software level (mainly in VLIW compilers) or proposed as a module preceding instruction fetch in order to schedule VLIW instructions for a static core processor.[15]

Calculating timing information of an instruction at its earlier stages has an additional advantage: for those instructions whose execution is predicted to be many cycles ahead, scheduling hardware algorithms can be sophisticated; disregarding them if they are time-consuming. Only for those that will execute in few cycles, hardware must be simplified and optimized. On the other hand, in classical dynamic schedulers, all the instructions must have a fast detection of dependences (in order to be woken up in one clock cycle) prior to execution stage; and when the

number of tags is elevated, this detection can decrease clock speed as mentioned above.

For the sake of simplicity and in order to explain CS operation we will write our examples using DLX ISA[3] (where instructions have a maximum of two source registers and one destination register) and the classic pipeline of Tomasulo's Algorithm for a scalar processor.[1] Anyway, the chrono-scheduling ideas can be similarly applied to other kind of dynamic scheduling architectures, like those based on register renaming or other variants,[4] and also it can be observed that extension of CS to superscalar processors is quite straightforward.

The pipeline on a Tomasulo-like processor is segmented in four stages: IF (Instruction Fetch), IS (Instruction decode and issue to the reservation stations RSs), EX (Execution in the FU; the result is stored in the corresponding RS), WB (Write Back of results to register file RF and to other RSs that wait for these data). Forwarding is done in WB phase, through what was called CDB, Common Data Bus.

Let us suppose that full latencies L_{UF} (including WB phase) are 2 and 4 for integer and MULT operations respectively. Firstly we analyze the case of infinite Functional Units (FU) and one CDB for each FU, which means that there will not be any structural conflict when a instruction wants to execute or do WB stage. The case of resource limitations will be explained in the next section. Let us consider the code given in Fig. 2, which execution timing is also represented in this figure, and its data flow in Fig. 3. Note that the code has no structural conflict at phases EX and WB. Symbol ° represents a waiting cycle (due to data dependences) in the RS where an instruction is allocated.

We can realize that all the timing information of an instruction can be extracted at each IS stage. In our first approach, as structural dependences have been avoided, timing information depends only on the data. The "production" of the destination register (the cycle in which the result will be calculated and sent to a CDB) will be done. L_{UF} cycles after both source operands are available at the FU. This is true because at this point we are supposing that a FU is always vacant. Moreover, in a

```
                      T1 T2 T3 T4 T5 T6 T7 T8 T9 T10
MULT R3, R2, R1       IF IS E1 E2 E3 WB
ADD  R4, R3, R1          IF IS °  °  °  EX WB
SUB  R5, R4, R3             IF IS °  °  °  °  EX WB
XOR  R4, R6, R1                IF IS EX WB
SHR  R3, R4, R2                   IF IS °  EX WB
```

Fig. 2. Timing diagram of Example 1.

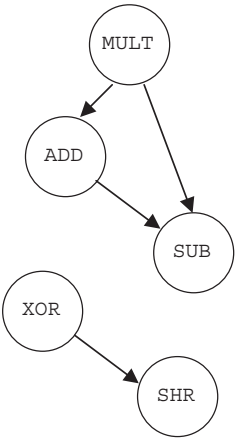

Fig. 3. Data flow of code example.

Tomasulo-like algorithm the cycle in which a source operand is available (see Fig. 2) can be: (a) the present cycle if the datum resides in RF or (b) the exact period in which it was sent to a CDB by another "producer" instruction if the datum was marked with a tag in RF. In other schedulers a generated or available datum may take several cycles to get to the corresponding FU.

Let us consider periods with respect to the IS stage of each instruction that we name "relative periods" (RP). For a given instruction, let us define Ts_j ($j = 1, 2$) as the RP when values of each source register will be available (namely, the number of cycles after the IS stage that a source operand is available). If an operand j is available at IS stage (because its value resides in RF or it is coming at this cycle through a CDB), we assign: $Ts_j = 0$. Analogously, let Td be the period when destination register of this instruction will be generated. Therefore, according to previous assumptions, we have for this instruction that: $Td = \mathrm{MAX}(Ts_1, Ts_2) + L_{UF}$.

This equation is clearly recursive because every source register is also the destination register of a previous instruction (that is, each Ts_j of the previous equation can be calculated through the same equation). Let us call Td_j the destination RP of the instruction that generated operand j. Then for the currently issued instruction, we can write: $Ts_j = Td_j - N_j$, where N_j is the number of cycles elapsed between the "producer" and the "consumer" instructions. As Td_j is measured with respect to the producer, this equation supposes a time reference change. Note that here "number of cycles" N_j is the same that "number of instructions" plus the stall cycles, because we are supposing that processor is scalar (it issues one instruction per cycle).

As an example, in Fig. 4 we show the RP extraction of source and destination registers. The last one is underlined to distinguish it from the other calculations. Let us examine some examples. In MULT operation both source operands are in

```
                T1 T2 T3 T4 T5 T6 T7 T8 T9

MULT R3,R2,R1 IF IS: R2,R1 available. R3 in MAX(+0,+0)+4=+4 (T7)

ADD  R4,R3,R1    IF IS: R1 avail.; R3 in +4(-1)=+3. R4 in MAX(+0,+3)+2=+5 (T9)

SUB  R5,R4,R3       IF IS: R4 in +5(-1)=+4;R3 in+4(-2)=+2. R5 in MAX(+4,+2)+2=+6(T11)

XOR  R4,R6,R1          IF IS: R1,R6 avail.; R4 in MAX(+0,+0)+2=+2 (T8)

SHR  R3,R4,R2             IF IS: R2 avail.;R4 in +2(-1)=+1, R3 in MAX(0,+1)+2=+3(T10)
```

Fig. 4. Timing of operands at IS stage.

the RF, so both Ts_j are 0, and the destination R3 will be produced 4 cycles after its issue stage. But for the next instruction ADD, one of the operands (R3, $j = 1$) will be generated by MULT. R3 was an RP of +4 with respect to MULT, but 1 cycle has elapsed between MULT and ADD ($N_1 = 1$), which means that: $Ts_1 = Td_1 - N_1 = 4 - 1 = 3$ (with respect to ADD). Calculation of destination RP $Td = \text{MAX}(Ts_1, Ts_2) + L_{UF}$, gives that register R4 will be available at $Td = \text{MAX}(+0, +3) + 2 = +5$, with respect to ADD issue. Note that the RP of R4 is changed afterwards by XOR instruction. As this register appears as a destination twice, its RP must be recalculated; it is the same action that occurs with register renaming in classical schedulers.

Therefore, full timing of operands can be extracted at IS stage. One possible implementation to achieve this is keeping all registers timing information in a kind of scoreboard. In a chrono-scheduler, RF must record for each register the RP and the FU that will produce a new value. When a register is destination of a newly issued instruction, the Td value and FU must be written in the RF. On the contrary, for every register that is not a destination, RP must be decremented at every clock cycle; N_j are calculated in this way. When the RP of a register gets to 0, it must capture its new value from the corresponding CDB (that of the FU recorded in the RF).

The evolution of register scoreboard of previous example is shown at Fig. 5. Rows contain the information of registers and columns represent absolute periods. In every column the issued instruction is indicated. Every cell encloses RP value for each register at the end of each cycle. In addition, FU are indicated between brackets if data is not available for a register, that is, if its RP is bigger than 0. Conversely if a data is available, we assign RP = +0 to this register. For example at the end of period T2, all data reside at the RF except that of R3, which will take 4 cycles to come from the MULT FU. At the next cycle R4 is marked with +5 [INT], because this register is destination of ADD instruction (that is, data will come from INT FU), and at its IS stage we can predict that the data for R4 will arrive in 5 cycles. At the same period, register R3 decrements its RP.

Despite the fact that RPs computations are done at IS stages (according to the formula $Td = \text{MAX}(Ts_1, Ts_2) + L_{UF}$), we indicate this calculations in the RF

	T2: MULT	T3: ADD	T4: SUB	T5: XOR	T6: SHR	T7
R1	+0	+0	+0	+0	+0	+0
R2	+0	+0	+0	+0	+0	+0
R3	+4 [MULT]	+3 [MULT]	+2 [MULT]	+1 [MULT]	~~+0~~ +1+2=+3 [INT]	+2
R4	+0	3+2=+5 [INT]	+4 [INT]	~~+3~~+2 [INT]	+1 [INT]	+0
R5	+0	+0	MAX[4,2]+2=4+2= +6 [INT]	+5 [INT]	+4 [INT]	+3

Fig. 5. Evolution of the register scoreboard.

scoreboard of Fig. 5, to follow better this procedure. Nonetheless some comparators (MAX functions) have been eliminated to simplify this figure. In Fig. 5 real data dependences are also shown with arrows, and RPs values that are read at IS stage are underlined. Register renaming done in classic Tomasulo-like schedulers, takes place here with a change of RP value (represented with a cross over a RP). This occurs at T5 for register R4, and in T6 for R3. Therefore, changes of RPs values avoid WAW hazards straightforwardly. In this fashion (and in a similar way to most schedulers) the results of MULT and ADD will never arrive to the RF, but only to the instructions that need them. This will be done at the stations as explained below. Also it is clear that WAR hazards are avoided at the stations, but without using any kind of tags.

In Fig. 6 we show the overall architecture of a CS algorithm. While its general structure reminds us that of a Tomasulo-like scheduler, several simplifications have been done, mainly at the stations previous to the FU. Let us explain at this point how the new CS stations work. An issued instruction can capture its source operands from RF at IS stage, or from a CDB (in case of data dependence) after an exact number of cycles Ts_j. So there is no need for CAM logic at the stations where instructions are waiting to be executed. Instead, RPs values can be stored at waiting stations and also decremented at every cycle until they are zero: at this moment the source value will exactly arrive from the corresponding FU (which must also be kept at the station). Then stations can be organized in a temporal manner: we can introduce a pile of stations for each FU and shift stations down at each cycle. So we can talk about "Shift Stations" (SS), instead of reservation stations. In Fig. 7 three SSs are represented for a 32-bit machine; the last of them is just a register that contains the operand value and the operation type to be executed in the next cycle. When an instruction issues, it must occupy the SS that takes the number of cycles given by $Ts = \mathrm{MAX}(Ts_1, Ts_2)$ to arrive to the FU. The other source RP, $\mathrm{MIN}(Ts_1, Ts_2)$, must be written in the SS to determine when the first operand source must be captured. Note that SSs are in fact in-order

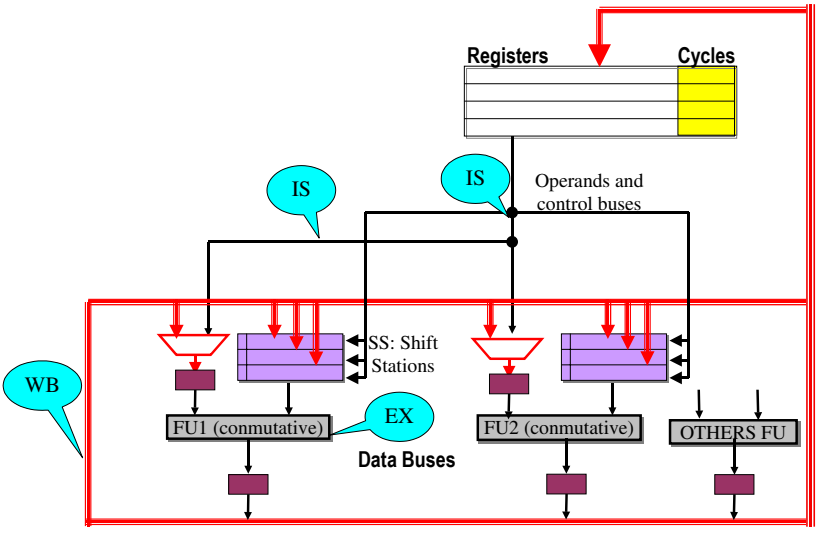

Fig. 6. Architecture of a chrono-scheduler.

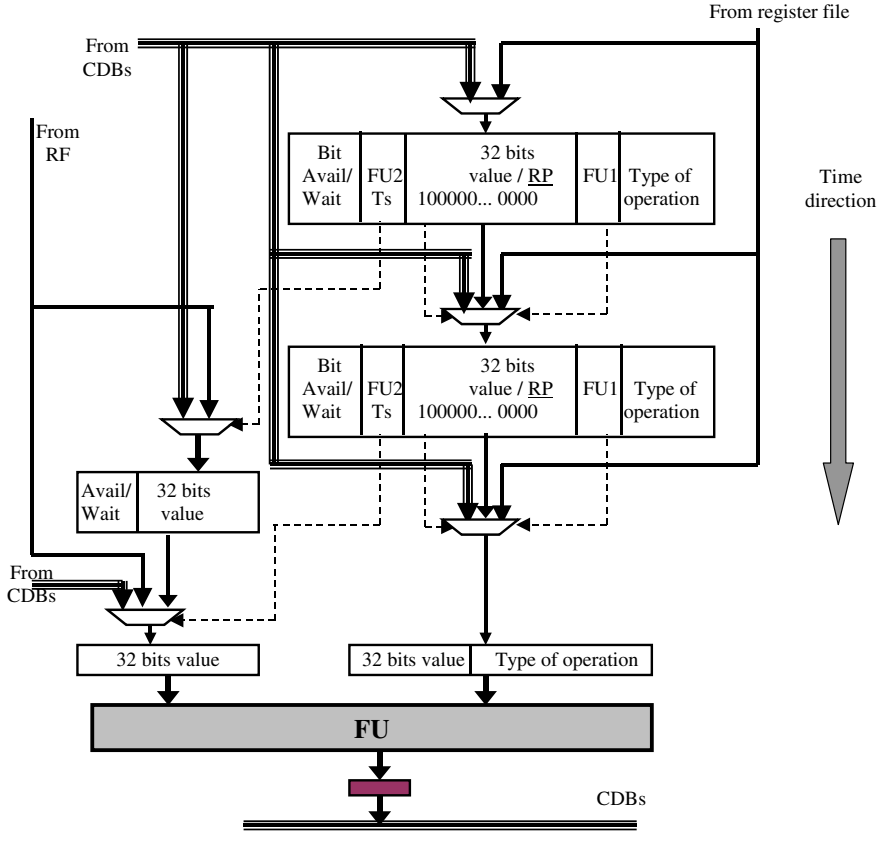

Fig. 7. Architecture of Shift Stations. Dotted lines are control signals.

execution stations (IS stage has ordered them due to RP calculation), that is, there is no need to implement priority circuitry or selection logic that decides which station must execute at each cycle. In this way we can say that time "goes down" to the FU. At each clock cycle, the control information of each SSs is copied to the immediately lower SS (buses that do this copy are not illustrated in Fig. 7 for clearness reasons). Also the operand field goes down to the FU as explained below. At the next section we will study what happens when the SS to be occupied is already busy (that is, FU is held in reserve at the required cycle by another instruction).

For example, instruction SUB has two dependences whose RPs values at issue period are +4 (coming from INT FU) and +2 (coming from MULT FU). These RPs will be decremented at each cycle. Therefore, SUB can be executed when +4 reaches zero, and prior to that cycle, the other operand must be captured from a CDB (when the RP value of +2 decreases until zero).

SSs must contain only one field for its first source operand value (if it exists) that will be captured for the producer FU1. The other operand (if it exists) will just arrive at the cycle previous to execution (field FU2 at Fig. 7 indicates which FU will generate the last operand). All the information in a SS is copied down to the FU, and the SS operand values can arrive through three different sources (see the MUX on each SSs in Fig. 7): directly from the RF if it was available when the instruction was decoded, from the upper SS or from one of the CDBs (when an instruction is waiting for the result of another one).

In this way, the operand value field can be used for a value or for a RP (the bit on the left field of each SS indicates if the operand is available or not). For implementation simplicity, in Fig. 7, RPs are decoded as a 32-bit number with only one active bit, whose position indicates RP value. So instead of decrementing, this number is shifted until the active bit arrives at the lower multiplexor's control signal; at this moment, the first operand is captured from the CDB indicated by FU1. For example, in Fig. 7, upper and second SSs will catch its operands in the next cycle. On the left of each SS, a bit indicates if operand value is available or not (in which case, it represents a RP). The other control signal of MUXs (that is, the one that decides if the operand will arrive from the RF or from the previous SS) are not drawn for reasons of clearness.

As we are using classic Tomasulo's pipeline, inferior SS has always its operand ready to execute, but for other pipeline implementations this could change. Similarly each FU output needs a latch according to Tomasulo's pipeline (which delays the data forwarding to WB stage). This latch could be suppressed if forwarding could be done at the end of EX.

Since SSs have been simplified to have only one source operand, FUs must comply with non-commutative operations. In SUB instructions this fact supposes only to invert the sign of operands. For DIV instructions a swap circuit at the input of FU should be added (potential increment in DIV latency will have little performance penalty, because these instructions are very infrequent and DIV

latencies in embedded processors are usually already large). For store instructions a similar reasoning is valid if a write buffer is implemented.

The number of required SSs depends on the clocks between issue and execute stages of an instruction. Of course, the bigger the latencies (or the bigger the superscalar degree), the more SSs are needed (though many of them might be empty).

As a result of the previous principle, an implementation approach suitable for scalar processors will be that based only on SSs. Here, when an instruction has to wait for an operand more cycles than the number of available SSs, a structural stall must be inserted. In order to prevent performance degradation, we should estimate the number of SSs to reduce these stalls. As a matter of fact, for real scalar processors, the mean number of cycles that an instruction waits is fairly small. While it is not easy to find explicitly this mean number for real processors and benchmarks in current literature, simulations[16] show that even for an aggressive high-ILP-oriented superscalar machine like PowerPC 620, this quantity is very low. Its mean value varies from 1.53 to 2.56 cycles in SPEC92 INT, and from 1.05 to 4.74 cycles in SPEC92 FP or from another point of view, from 1.01 to 2.39 cycles in integer RS, 1.39 to 2.56 cycles in Load/Store RS and 2.45 to 4.74 cycles in FP ALU RS for SPEC92. In PowerPC 620, instructions reside more time in its RS obviously because of bigger FP ALU latencies (3 stages), but mainly because FP execution is actually in order. Note that PowerPC's mean number of cycles in FP and Load/Store RSs are fully valid for our case, because these FUs are single (for integer instructions, PowerPC implement two FUs, so cycles must be higher). But, evidently for the scalar processor we want to "chronoschedule", this number will be much lower. Therefore, according to previous study, it is a fact that the number of SSs necessary to prevent from many stalls is not elevated.

But in the case of more aggressive processors, instead of having a large number of fast SSs, it would be preferable to implement a pool of stations that "holds" instructions that will be executed in a number of cycles bigger than the number of available SSs (in the rest of the paper we will describe this more general case). This pool of "Hold Stations" (HS) needs a common multiplexor (see implementation in Fig. 8) to choose the station that will be launched to SSs. Also a second counter to indicate RP of the last or second operand (which coincides with future EX period) is necessary for each entry. The maximum number of bits of this counter is bounded by the maximum number of clocks that a CS can predict, which will occur for a bizarre piece of code: that composed by a chain of longest latency instructions, each of ones includes a real data dependence with its predecessor. Then the maximum number of predictable clocks $PCLK_{\max}$ is given by:

$$PCLK_{\max} = \max(L_{UF}) \times n_{HS} - \left\lfloor \frac{n_{HS} - 1}{m} \right\rfloor , \tag{1}$$

where $\max(L_{UF})$ is the biggest of all full latencies L_{UF} (including WB phase), n_{HS} is the number of HSs and m is the issue width ($m = 1$ for a scalar processor). The

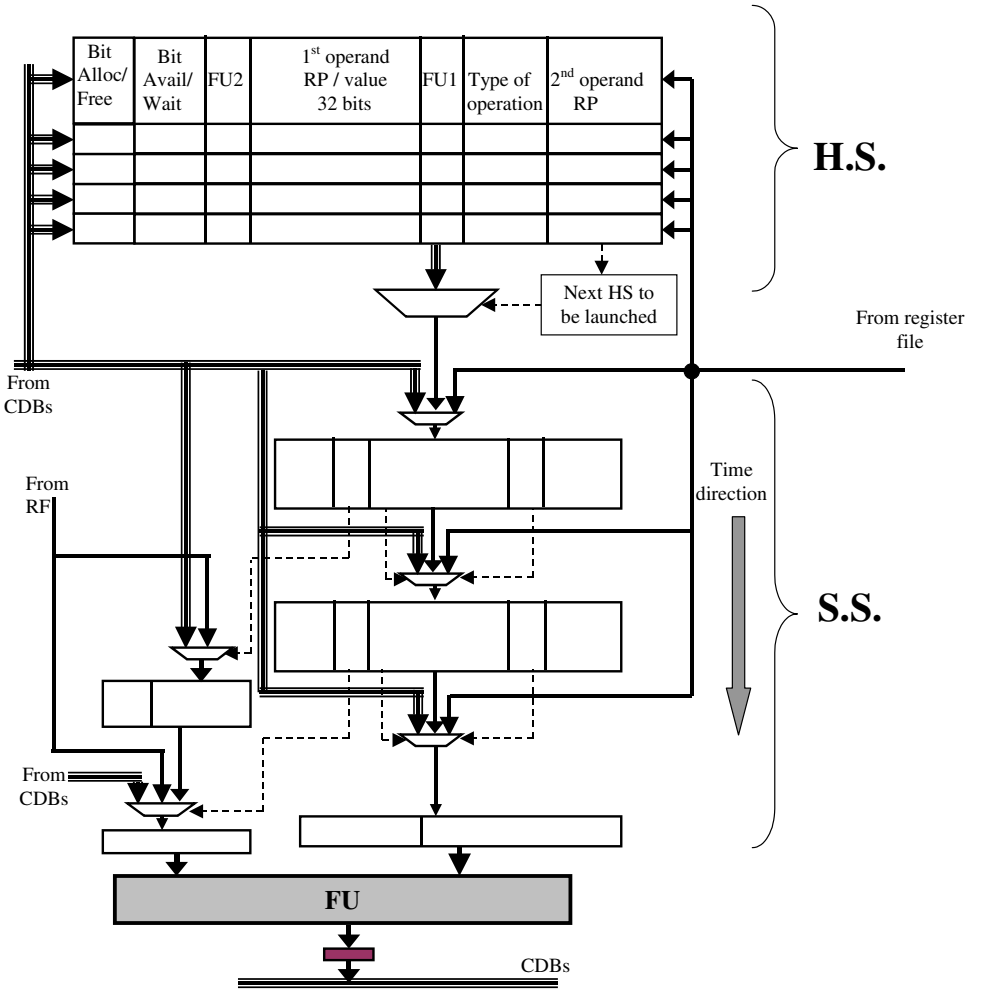

Fig. 8. Architecture of SSs and HSs. Dotted lines are control signals.

last member of the previous equation is introduced to take into account the number of cycles that elapses from the first to the last HS allocation. The maximum number of bits of this counter is therefore:

$$\lceil \log_2 (PCLK_{\max}) \rceil \ . \tag{2}$$

While this maximum number of predictable clocks would only occur in a bizarre piece of code, note that the number of bits of the counter is not elevated: for example, for a longest latency of 8 periods, an issue width of 4 and 64 HSs, then $PCLK_{\max} = 497$ cycles, which means that a 9-bits counter will be enough.

When the last operand's RP of an HS equals the number of SSs plus 1, it must be launched to the first SS (an alternative is to store the difference between the

last RP and the number of SSs plus 1, detecting when it gets to zero). The first operand of an HS is captured in a similar way to SS, while the second operand will be caught just the cycle before doing EX.

As HSs include complete information about an issued instruction, SSs may be unnecessary for some processor designs. In this case note that HSs behave identically as RSs from the viewpoint of allocation and liberation. This is clear if we observe that timing diagram of both schedulers is the same (note for example in Fig. 2 that this diagram is valid for a RS scheduler and for CS). Therefore, for a processor based only on these HSs, it is obvious that processor CPI will be the same than that of a RS-based processor (for a number of RSs equal to the number of HSs). But if our design objective was to reduce the launching time, SSs must be incorporated. The reason is that HS structure is more complicated than that of SS. For example if SSs were not present, the launching time may increase due to the big multiplexor that selects the next HS to be launched (see Fig. 8). As the path needed to send an instruction to its corresponding FU is much shorter for SSs than for HSs, the common solution will be to maintain a short number of SSs under the pool of HSs (as shown in Fig. 8). The exact number of HS and SS should be carefully determined for each CS implementation, according to the selected target processor and the goal performance requirements, because this decision has a decisive impact in the overall performance of the design (frequency, power, area, etc.). In Fig. 9 a cost summary of RS in relation to proposed SS and HS is presented using the architectural parameters involved in these stations (for a 32-bit machine with a CDB for each FU). In HS and SS the 32-bit first operand value is used to contain

Station	**Sequential fields, bits and main associated circuitry**								
RS	Allocate/ Free	Type of operation	Destination tag	1st operand tag	1st operand value	1st operand available bit	2nd operand tag	2nd operand value	2nd operand available bit
Bits	1	$Log_2(N_{type})$	$Log_2(N_{tag})$	$Log_2(N_{tag})$	32	1	$Log_2(N_{tag})$	32	1
Main associated circuitry	Selection Logic (if ready)					Wakeup logic	N_{CDB} Comparators of $Log_2(N_{tag})\times Log_2(N_{tag})$ inputs		Wakeup logic
HS	Allocate/ Free	Type of operation	Avail./Wait Bit	RP-FU1/value	FU2	2nd operand RP			
Bits	1	$Log_2(N_{type})$	1	32	$Log_2(N_{CDB})$	$Log_2(N_{MCLK})$			
Main associated circuitry				$Log_2(N_{MCLK})$-Down counter		$Log_2(N_{MCLK})$-Down counter. Launch logic			
SS	Allocate/ Free	Type of operation	Avail./Wait Bit	RP-FU1/value	FU2				
Bits	1	$Log_2(N_{type})$	1	32	$Log_2(N_{CDB})$				
Main associated circuitry				$Log_2(N_{MCLK})$-Down counter					

Notation: N_{type} : Maximum number of operations of the FU; N_{tag} : number of tags ; N_{CDB} : number of CDBs
N_{MCLK}: number of estimated predictable cycles

Fig. 9. Cost summary of RS, SS and HS (for a 32-bit machine).

both the RP and the type of FU, since the number of bits that specify the FU1 ($\mathrm{Log}_2(N_{(CDB)})$) is small.

In addition, having in mind this combination of HS and SS, we can become aware of other advantages as explained below.

Whereas HSs are more complicated than SSs, we know that their instructions will be executed later, so their circuits need not to be enhanced or made fast. For example, if HS launch were lengthy, it could be pipelined in two cycles and launched. HS would occupy the second SS instead of the first one (or equivalently a HS could begin its launch two cycles ahead). An analogous argument is valid for bypasses from FUs to HSs, because it is not necessary to do these bypasses in just one cycle. On the whole, note that this idea can be extended to any piece of CS hardware. As time information is known for each event that will occur in the scheduler, only the pieces that are going to be executed in the next cycle must last one period. On the other hand, for those parts that will last (say) m cycles to arrive to the FU, we have these m cycles to do all the work involved in the launch. The same reasoning is valid for other events (like issuing an instruction, capturing its operand in the HS, etc.). We can conclude that the only critical pieces of a chrono-scheduler issue window are those related to the last SS (the one that is ready to execute). Therefore, a CS permits to implement an attractive design like that shown in Fig. 10 (for two FUs, analogously for more FUs), which divides the scheduler in two sections of different speeds (and technology if required). The bottom part includes all the critical time scheduler circuitry, while the upper part contains those pieces where time is not critical. The element with the symbol Δ suggests that a delay in these buses will not degrade the processor performance.

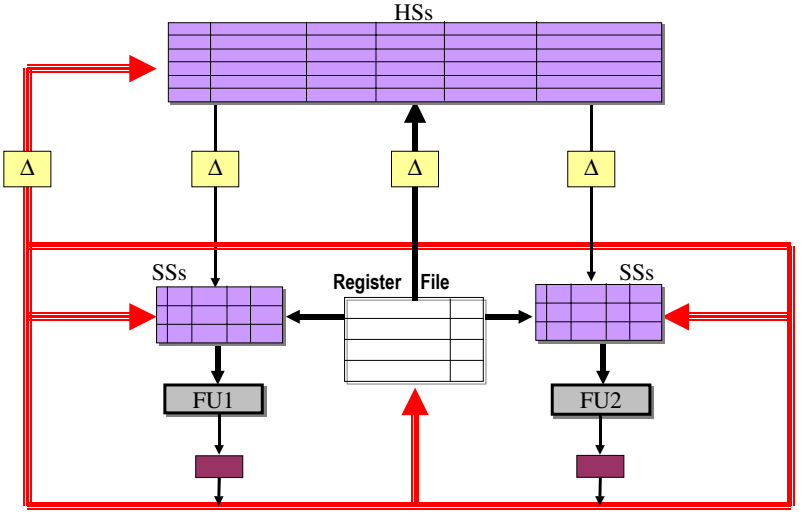

Fig. 10. A two-speed CS design.

Additionally, as it can be presumed that the major dissipation sources of a CS scheduler are found around HSs (it is the biggest CS part, it has to locate associatively free entries, it has an enabling circuitry to launch an HS to the pool of SSs, etc.), a power saving design could be implemented for this upper part. For example, this will be achieved if it works with a lower frequency and with reduced power technology (which would not impact the processor performance as exposed before). A preliminary theoretical power saving estimation can be found for CMOS technology. It is well known that power consumption in this technology is quadratic with voltage supply and linear with switching capacitance and activity.[17] Therefore, if an implementation of HSs worked with half frequency (it must have two SSs — working with full frequency — for each FU), it would reduce its activity to the half, that is, power dissipation of this part of the scheduler would be reduced to one half. Of course a full implementation of a CS-based processor is needed to compare more exactly the savings in power with respect to a RS-based processor.

HSs plays a similar role than classical RSs but they are much simpler: no CAM logic is needed and only one operand must be contained (therefore, they need only half data path inputs of RSs). In addition, as RPs are known we can predict which HS is going to be launched. For example, a simple control register could permanently store the next HS to be launched (that is, MUX control lines) if it were actualized each time a new HS is occupied or a HS is launched (made free). Therefore, this special selection logic will not cause any delay in the launch of a HS, as it can be done in parallel with a previous launch.

3. Resource Limitations and Structural Stalls

When an instruction to be issued does not find an empty HS, a structural stall must be inserted. The other possible stall cause, that is, if an RP does not fit in a counter, is very improbable if a moderate number of count bits are implemented. In order to prevent performance degradation, we must estimate the number of HSs to reduce these stalls. As HSs plays a similar role than RSs, a moderate number of HSs will guarantee a high speedup with respect to a static scheduler processor. What is more, the number of HSs may be lower than that of classical RSs as several instructions (those that are prepared to execute) reside in the SSs.

The challenge of limited number of FUs and CDBs can be resolved with a binary reservation table (BRT) for each shared resource. For example, if an instruction is scheduled at IS stage to capture its last operand at the same period than a previous one (using the same FU), it must be delayed. For instance this case will occur in example of Fig. 2 at period T8 for ADD and SHR instructions if MULT latency were one cycle longer. Then SHR must be delayed to execute one cycle after to prevent the structural hazard. Therefore, once $Ts = \mathrm{MAX}(Ts_1, Ts_2)$ has been determined, a search in the corresponding FU BRT beginning by the Ts^{th} bit (for example using a priority encoder), will find the first period T_{EX} where this FU is free. At this period T_{EX} the instruction must begin its execution.

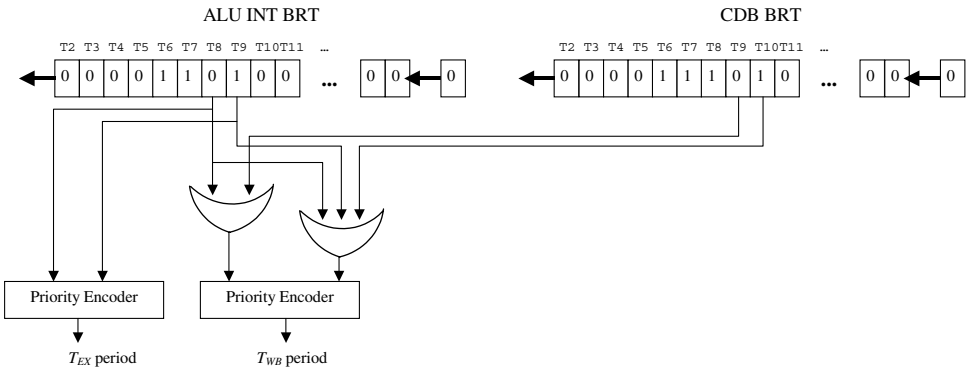

Fig. 11. Implementation design for FU and CDB BRTs searching.

A similar BRT and searching can be implemented if CDBs were limited. For example if processor design had a single CDB, we would need a FU BRT and a CDB BRT. In this case a simple circuitry can be implemented to find the periods T_{EX} and T_{WB}, at which EX and WB stages must begin. Supposing that a "0" in a BRT bit means that the resource is free, and a "1" means occupied, then the priority encoder searching should be preceded by OR operations between bits of FU BRT and CDB BRT (this last shifted by FU duration). In Fig. 11 we schematize this design for a two-bits searching, showing the BRT contents when SHR is going to be issued (at T6 period). Note that for easy understanding in this figure we are showing absolute periods T, instead of the relative ones that will be managed in a real processor. In Fig. 11, searching at CDB BRT is shifted by one cycle with respect to the FU BRT searching. The priority encoders' outputs will give us the RP of EX and WB stages for the SHR instruction.

In the rest of this paper we will work with single FU and a CDB for each FU, for the following reasons. Firstly in our architecture, each SS needs only one path for both operands (the other path is common for all stations, and placed on the left operand). Moreover, most current embedded or vector processors implements single FU but multiple forwarding paths.[11,18] As the number of FUs is reduced for these machines (2 or 3 for each pipe), a double or triple CDB supposes a moderate number of forwarding paths if we compare for example with ARM9TDMI, which in spite of being a simple scalar 5 stages machine, it has 5 forwarding paths. Finally CS CDBs are much simpler than those of common schedulers because of the following: there is no need for priority circuitry to select one of the CBDs, CDBs have no tag and consequently logic associated to tags is not necessary.

Let us suppose that an instruction can look into a BRT for K_{UF} bits in a period. If no "0" is encountered in these K_{UF} bits, the simpler solution is to insert a stall. That is, no instruction is issued at this cycle, and a new K_{UF}-bits search must be done in the next cycle. These stalls would be almost negligible even with a small K_{UF}. In current literature it is not easy to find explicitly for real processors

how frequent an instruction is "ready but waiting for FU". However, simulations show that in a well-balanced machine this case is rare. In PowerPC 620,[16] for FP benchmarks, average busy FU rate is less than 3 per 100 cycles, and for SPEC92 INT it reaches a maximum of 13.67% (in *compress* benchmark). For this processor, then we will expect a maximum structural rate of these stalls given by $(0.1367)^{K_{UF}}$. For instance if this search were 3 bits wide, maximum probability of stall will be less than 2.6 per 1,000 cycles of integer benchmark's execution (for FP programs it will be virtually zero). Bearing in mind these probabilities, the maximum degradation of the CPI of a CS with respect to the CPI of a RS-based scheduler, can be fairly estimated. For a 100-cycle SPEC92 INT execution, if CPI_{RS} is the expected CPI for a RS-based scheduler, then the maximum deceleration can be:

$$A = \frac{CPI_{CS}}{CPI_{RS}} = \frac{100 + (0.1367)^{K_{UF}}}{100} \Rightarrow A = (0.1367)^{K_{UF}}\% \,. \tag{3}$$

For a 3-bit wide BRT search, this gives a deceleration of 1.0026. Note that this CPI degradation for the SPEC92 INT average is very much lower, and for FP benchmarks is virtually inexistent (an average deceleration of $(0.03)^{K_{UF}}\%$, that is, 0.000027% for a 3-bit wide BRT search).

Given that EX stage may be delayed by a maximum of $K_{UF} - 1$ cycles, a new set of $K_{UF} - 1$ shift registers must be inserted previous to the latch at FU left input (in Figs. 7 and 8, an additional register has been added to the left input to illustrate this situation, which means that K_{UF} is "2" for this figure). One of these latches will collect the last operand at cycle given by *Ts*; then *Ts* must be stored in the corresponding SS (in Fig. 7, *Ts* is together with FU2). This new field can be decremented each cycle, in a similar fashion to the right operand, so shift register will be loaded when a SS orders it (*Ts* gets to zero). Another more infrequent stall cause occurs when the length of a BRT is less than the calculated T_{EX}. While this stall can be avoided if BRT structure were modified to log this case, it is clear that the rate of this stall is negligible if BRT contains a sufficient number of bits. Moreover note that the maximum number of bits of this BRT is bounded by $PCLK_{\max}$ (maximum number of clocks that a CS can predict). As explained in Sec. 2 this number is not elevated for usual programs.

To sum up: a chrono-scheduler may stall if a resource is exhausted as it occurs in every dynamic scheduler. In our case, stalls must be inserted if: (*a*) HSs are exhausted; (*b*) When looking for a FU free, there is no success in BRT exploration; (*c*) The length of a BRT is less than the calculated T_{EX}. Then we conclude that the expected CPI of a chrono-scheduler with a number of HSs, will be approximately the same to that of a classical dynamic scheduler with the same number of RSs. Consequently, the impact of most of architectural parameters in CS performance will be similar to the impact for a RS-based scheduler. For example, the impact of queue size, issue width or branch statistics in CPI will be the same as that obtained using classical Reservation Stations; thus well-established CPI studies (like Refs. 3, 14, 17 and so on) can apply to chrono-scheduler performance.

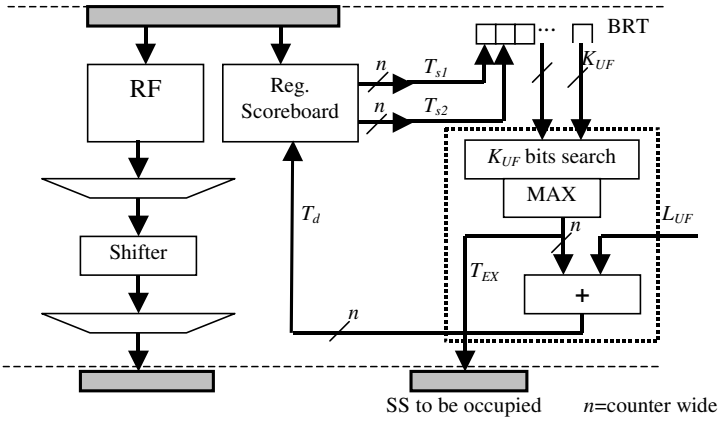

Fig. 12. Chrono-scheduling issue implementation.

An example of a possible implementation of chrono-scheduler issue stage for an XScale processor is given at Fig. 12. The right part of this schematic will calculate RP and SS to be occupied for current instruction (if HSs were implemented, outputs of this circuit will be placed on a empty HS). Here the MAX function has been placed after the BRT access to shorten the total delay of this circuit, so all dotted square can be implemented as a simple two-level AND-OR circuitry. Others details that are not in the critical path are not shown in this figure, like the minimum of (Ts_1, Ts_2), that must be recorded in the SS or HS, and the setting of the T_{EX} bit in the BRT. As it occurs in all superscalar implementations, chrono-scheduler IS stage is not time-scalable with respect to the issue width, so a detailed calculation of maximum width should be done for each machine.

Besides if precise interrupts or dynamic speculation are to be implemented, classical techniques (like reorder buffer) can be employed. However a deeper study of precise interrupts and speculation implementation should be accomplished in future works, in order to find possible simplifications of this piece of hardware when chrono-scheduling information is used.

Ongoing work includes selecting several target processors where CS could fit as a cheap dynamic scheduler, and writing an implementation in VHDL. This will permit us to quantify exactly the savings in circuitry and power, clock frequency increase and issue latency reduction with respect to a similar processor with a register renaming scheduler, and comparing different types of target processors and even multiple architectural factors for a particular processor family.

4. Conclusions

The structure of a chrono-scheduler targeted to time-predictable processors is first presented. It avoids the usual complexity found in RSs of classical dynamic schedulers, because it extracts timing information prior to issue, so no associative logic

is needed. Its main advantages are: there are no tags in the system, no renaming, data buses are not enlarged with tag information, each waiting station is very much simpler (no comparing of logic nor CAM, no priority circuitry, only one operand instead of three, only one datapath instead of two for each RS). Only small adders and BRTs are added after the decoding stage. On the whole, it is apparent that CPI for CS will be similar to that of classical schedulers because stalls come from similar running out of resources, but clock speed may be increased because of its simplified SSs, and circuitry complexity, and power consumption is predicted to be fairly lower. Moreover, we present a design where the critical time part (which is the least) is separated to the non-critical, allowing the implementation of this last part with a power saving technology.

Acknowledgments

This work has been supported in part under Spanish Science and Education Ministry Research Project TIN2006-15617-C03-03 and under Andalusian Government Excellence Research Project P06-TIC-02298.

References

1. R. M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, *IBM J. Res. Dev.* **11** (1967) 25–33.
2. J. E. Thornton, Parallel operation in control data 6600, *Proc. AFIPS Fall Joint Computer Conf.* (1964), pp. 33–40.
3. J. L. Hennessy and D. A. Patterson, *Computer Architecture. A Quantitative Approach*, 2nd edn. (Morgan-Kaufmann, 1996).
4. M. Moudgill, K. Pingali and S. Vassiliadis, Register renaming and dynamic speculation: An alternative approach, *Proc. 26th Int. Symp. Microarchitecture*, Texas (1993), pp. 202–213.
5. D. Sima, The design space of register renaming techniques, *IEEE Micro* **20** (2000) 70–83.
6. D. V. Ponomarev, G. Kucuk, O. Ergin, K. Ghose and P. M. Kogge, Energy-efficient issue queue design, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **11** (2003) 789–800.
7. S. Palacharla, N. P. Jouppi and J. E. Smith, Complexity-effective superscalar processors, *Proc. 24th Ann. Int. Symp. Computer Architecture (ISCA'97)*, June 1997, pp. 206–218.
8. D. Ernst and T. Austin, Efficient dynamic scheduling through tag elimination, *Proc. 29th Ann. Symp. Computer Architecture (ISCA'02)* (2002), pp. 37–46.
9. M. A. Ramirez, A. Cristal, A. V. Veidenbaum, L. Villa and M. Valero, Direct instruction wake-up for out-of-order processors, *Proc. Innovative Architecture for Future Generation High-Performance Processors and Systems* (2004), pp. 2–9.
10. D. Brash, *The ARM Architecture Version 6 (ARMv6)*, ARM White Paper, January 2002.
11. C. E. Kozyrakis and D. A. Patterson, Scalable, vector processors for embedded systems, *IEEE Micro* **23** (2003), pp. 36–45.
12. R. Espasa, M. Valero and J. E. Smith, Out-of-order vector architectures, *Proc. 30th Ann. Int. Symp. Microarchitecture* (1997), pp. 160–170.

13. L. Thiele and R. Wilhelm, Design for timing predictability, *Real-Time System* **28** (2004) 157–177.
14. S. Önder and R. Gupta, Instruction wake-up in wide issue superscalars, *Euro-Par 2001*, LNCS, Vol. 2150 (Springer-Verlag, Berlin, Heidelberg, 2001), pp. 418–427.
15. S. Banerjia, S. W. Sathaye, K. N. Menezes and T. M. Conte, MPS: Miss-path scheduling for multiple-issue processors, *IEEE Trans. Comput.* **47** (1998) 1382–1397.
16. T. A. Diep, C. Nelson and J. P. Shen, Performance evaluation of the powerPC 620 microarchitecture, *Proc. 22nd Ann. Int. Symp. Computer Architecture (ISCA'95)*, Italy, pp. 163–174.
17. D. Folegnani and A. González, Energy-effective issue logic, *Proc. 28th Ann. Int. Symp. Computer Architecture*, Sweden (2001), pp. 230–239.
18. S. Segars, The ARM9 family — high performance microprocessors for embedded applications, *Proc. Int. Conf. Computer Design*, October 1998, pp. 230–235.