A Two-Level Dynamic Chrono-Scheduling Algorithm

F. Diaz-Del-Rio, J.L. Sevillano, S. Vicente,
D. Cagigas, M.R. López-Torres
Escuela Técnica Superior de Ingeniería Informática.
Universidad de Sevilla. Phone (34) 954 556144.
FAX: (34) 954 552899.
Avda. Reina Mercedes s/n 41012 Sevilla, Spain.
fdiaz@atc.us.es

***Abstract*—We propose a dynamic instruction scheduler that does not need any kind of wakeup logic, as all the instructions are "programmed" on issue stage to be executed in pre-calculated cycles. The scheduler is composed of two similar levels, each one composed of simple "stations", where the timing information is recorded. The first level is aimed to the group of instructions whose timing information cannot be calculated at issue (for example, those instructions whose latency is not predictable). The second level contains simple "stations" for the instructions whose execution and write back cycle have been already calculated. 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. Another additional advantage is that time critical parts can be identified as instruction timing information is available, so high speed and frequency logic can be used only in these parts, while the rest of the scheduler can work at lower frequencies, therefore consuming much less power. The lack of wakeup and CAM (Content Addressable Memory) means that power consumption and latencies would be presumably reduced, frequency would probably be made higher, while CPI (clock Cycles Per Instruction) would remain approximately the same.**

***Keywords*: Computer architecture, instruction level parallelism, dynamic scheduling, reservation stations, reservation tables, superscalar processors.**

I. INTRODUCTION

The inclusion of a dynamic scheduler in a superscalar processor permits to extract a high amount of instruction level parallelism (ILP), and boosts its performance in a transparent way to the programmer. 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 [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], etc.).

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-up 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 the scheduler clock speed, because wakeup 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 [7][6]. What is more relevant: the main sources of power dissipation of a scheduler are those related to associative matching and selection logic. Particularly, these 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 to execute and reading the selected instructions information. Having in mind all stated above, it is not surprising that in the last few years, one of the main research topics in computer architecture has been not only to achieve the desired target performance, but also to deliver it with power efficiency.

This paper is structured as follows. In section 2, a description of a basic chrono-scheduling (CS) implementation is summarized. This basic CS was presented in Ref. [10] and was targeted to processors that have constant latency operations like many embedded microcontrollers, most vector processors without data cache, etc. The general CS, targeted to any processor (like superscalar processors that include variable latencies operations and complex memory

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.

hierarchy) is introduced in this paper. In section 3, the problem of resource limitations is introduced, so the second CS level is illustrated. Then a first CS level is presented to include the case of operations whose latency is not predictable (like accesses to memory hierarchy) in the following section. Finally conclusions are summarized.

II. Basic 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][11]. On the contrary, 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 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 for CS that duration of the execution phase is predictable at IS stage to extract timing information (which is usual in real time processors, but not in a general purpose processor). 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 [12].

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 made more sophisticated or can work at low frequencies to save energy, disregarding if they are time-consuming. Only for those that will execute in few cycles, hardware is time-critical and it 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.

In this section we analyze the case of predictable latencies so we will describe the CS architecture [10]; we introduce the solution of unpredictable latencies in next sections. For the sake of simplicity and in order to explain CS operation we will write our examples using DLX ISA [3] (where instructions has a maximum of two source registers and one destination register) and the classic pipeline of Tomasulo's Algorithm [1] for a 4-width issue superscalar processor. 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]. 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 Common Data Bus (CDB).

Let us suppose that full latencies L_{UF} (including WB phase) are 2 for integer and memory access operations, 3 for FP ADD and 4 for MULT operations, and that the processor has 2 integer units and can access twice to memory. The rest of Functional Units (FU) are single, each FU has one CDB, which means that there will not be any structural conflict when an instruction does its WB stage. Consider the classical DAXPY code given in **Fig. 1**, which execution timing is also represented in this figure. Symbol ° ° represents a waiting cycle due to data dependences, (symbols – and __ will be explained later). To contemplate variable latency operations, we consider that L1 cache accesses hit in the first iteration of DAXPY loop, but that a L1 miss occurs in the second Load of the second iteration.

We can realize as follows that all the timing information of an instruction can be extracted at each IS stage. Let us first suppose that there is not any structural dependence, so timing information depends only on the data. In this case, 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 Tomasulo-like algorithm the cycle in which a source operand is available (see Fig. 1) can be: *a)* the present cycle if the datum resides in RF or *b)* the exact period in which it is sent to a CDB by a "producer" instruction if the datum is marked with a tag in RF. Let us consider periods with respect to the IS stage of each instruction, which 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: $Td= MAX(Ts_1, Ts_2)+ L_{UF}$.

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. Therefore, full timing of operands can be extracted at IS stage.

```
LOOP:             T1 T2 T3 T4 T5 T6 T7 T8 T9 T0 T1 T2 T3 T4 T5 T6 T7
LD  FX, (RX)0     IF IS L1 WB
LD  FY, (RX)2048  IF IS L1 WB
MULTD FM,FX,FA    IF IS -- __ M1 M2 M3 WB
ADDD  FAD,FM,FY   IF IS -- -- __ °° °° °° A1 A2 WB
--------------------------------------------------------------------------------
SD   (RX)2048,FAD    IF IS -- -- __ °° °° °° °° °° EX WB
ADDI  RX,RX, 8       IF IS EX WB
SLT   R3,RX,REND     IF IS __ °° EX WB
BNEZ  R3, LOOP       IF IS -- __ °° °° EX WB
--------------------------------------------------------------------------------
LD  FX, (RX)0           IF IS °° L1 WB
LD  FY, (RX)2048        IF IS °° L1 L2 L2 L2 L2 L2 WB
MULTD FM,FX,FA          IF IS -- -- __ M1 M2 M3 WB
ADDD  FAD,FM,FY         IF IS -- -- -- -- -- __ °° °° A1 A2 WB
--------------------------------------------------------------------------------
SD   (RX)2048,FAD          IF IS -- -- -- -- -- __ °° °° °° °° EX WB
ADDI  RX,RX, 8             IF IS EX WB
SLT   R3,RX,REND           IF IS __ °° EX WB
BNEZ  R3, LOOP             IF IS -- __ °° °° EX WB
--------------------------------------------------------------------------------
LD  FX, (RX)0                 IF IS °° L1 WB
LD  FY, (RX)2048              IF IS °° L1 L2 L2 L2 L2 WB
```

Fig. 1: Timing diagram of example 1 (FA contains the value of constant ‘a’; REND points to the end of vector X)

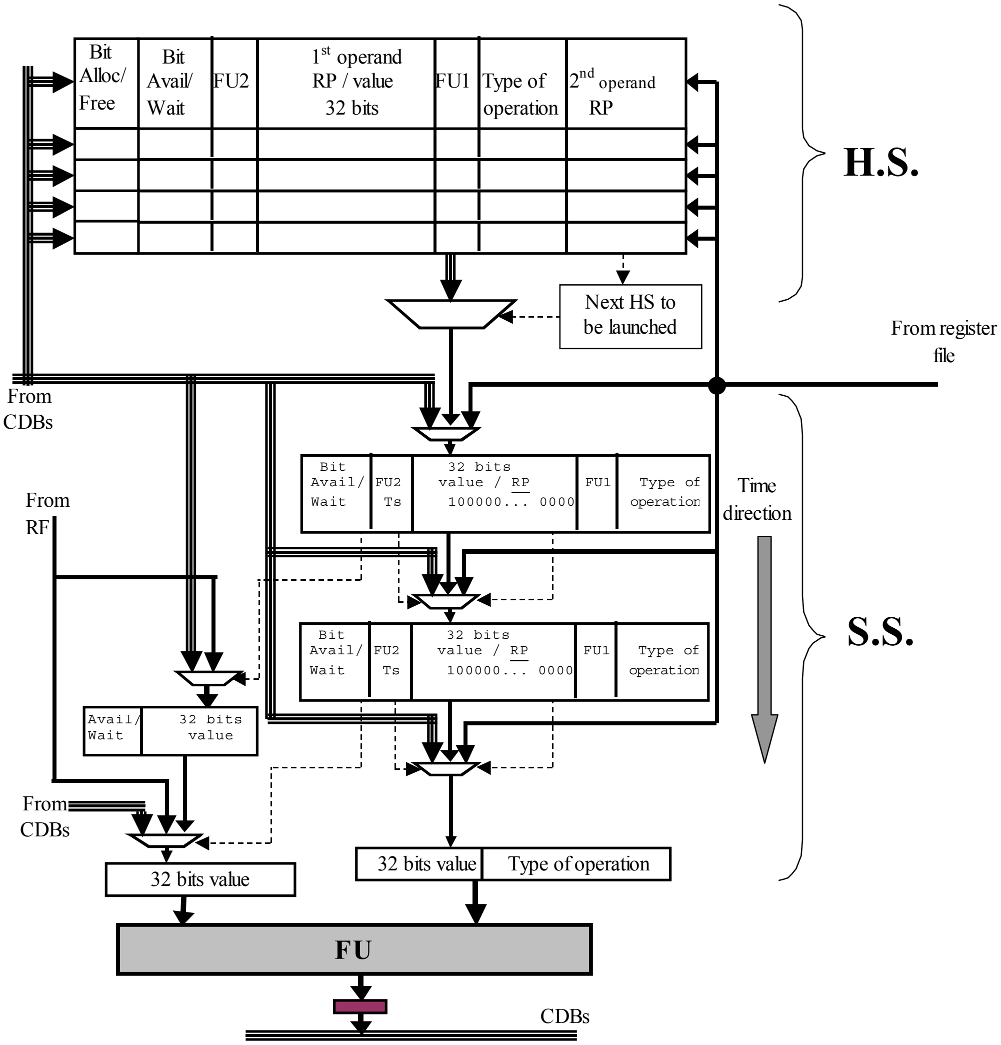

Fig. 2: Architecture of SSs and HSs. Dotted lines are control signals

One possible way to implement this is to keep all registers timing information in a kind of scoreboard. In a chrono-scheduler, RF must record for each register the *Td* (as a 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 (see the four left fields in Fig. 6). 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).

Register renaming done in classic Tomasulo–like schedulers, takes place in a CS with a change of RP value. Therefore changes of RPs values avoid WAW hazards straightforwardly. When this occurs the results of the first instruction of a WAW will never arrive to the RF, but only to the instructions that need them (similarly to most schedulers). 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. 2 we show a possible implementation of a CS algorithm. In a CS station, 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 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. 2 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.

Let us consider first that an issued instruction is ready to execute in less than 4 cycles (that is, its $Ts<4$). Then, it must occupy the SS that takes the number of cycles given by Ts=MAX(Ts_1, Ts_2) to arrive to the FU. The other source RP, MIN(Ts_1, Ts_2), must be written in the SS to determine when the first operand source must be captured (if it were not caught from the RF). Note that SSs are in fact in-order 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. 2 for clearness reasons). Also the operand field goes down to the FU as explained below. At the next section we will examine 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).

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. 2 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. 2): 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).

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

An implementation approach (suitable for simple 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. As a matter of fact, for many 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 [13] 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 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 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. This pool of "Hold Stations" (HS) needs a common multiplexor (see implementation in Fig. 2) 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. For example, for a longest latency of 8 periods, an issue width of 4 and 64 HSs, then maximum number of predictable clocks ($PCLK_{\max}$) is 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. 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 almost identically than RSs from the viewpoint of allocation and liberation. This is clear if we observe that timing diagram of both schedulers is the same (for example diagram in Fig. 1 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 almost 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 were to reduce the launching time, SSs must be incorporated. The reason is that HS structure is more complicated than that of SS. As the path needed to send an instruction to its corresponding FU is much shorter for SSs than for HSs, a good solution will be to maintain a short number of SSs under the pool of HSs (as shown in Fig. 2). 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 (frequency, power, area, etc.). A more precise cost summary of RS in relation to proposed SS and HS could be consulted in [10]. 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 definitively 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 (or similarly we can reduced the frequency operation m times). The same reasoning is valid for other events (like issuing an instruction, capturing its operand in the HS, etc.).

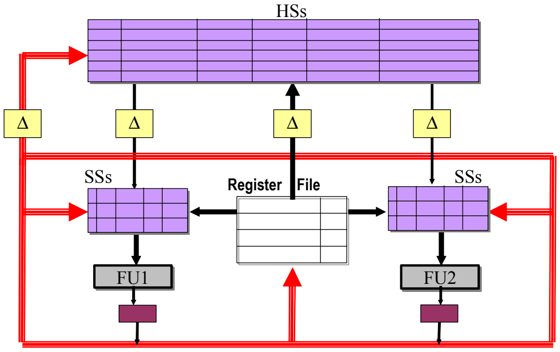

Fig. 3: A two-speed CS design

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. 3 for two 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. 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). 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.

III. 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. Moreover, the number of HSs may be lower than that of classical RSs to get the same performance, 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. 1 for LD instructions and several INT operations if there were only one FU for them. Then later instructions in this situation must be delayed to execute one cycle. Therefore once $Ts=MAX(Ts_1, Ts_2)$ 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.

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. 4 we schematize this design for a two bits searching, for some piece of code. Note that for to 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. 4, searching at CDB BRT is shifted by one cycle with respect to the FU BRT searching (for INT operations due that its duration is 1). The priority encoders' outputs will give us the RP of EX and WB stages for an INT instruction.

In the rest of this paper we will work with finite 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, the existence of many CDB does not complicate wakeup logic as it does not exist for CS. 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.

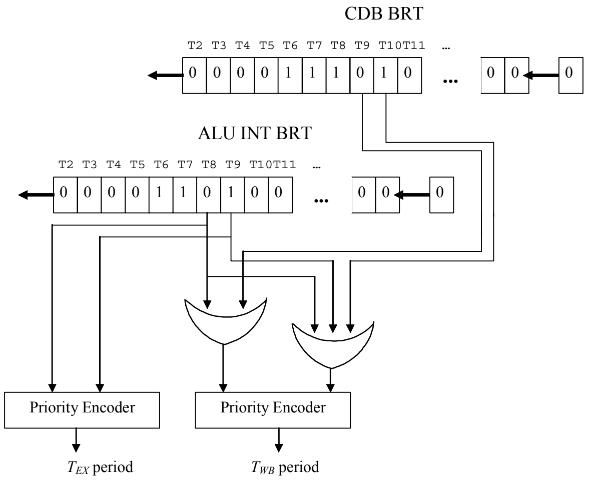

Fig. 4: Implementation design for FU and CDB BRTs searching

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 [13], 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 1000 cycles of integer benchmark's execution (for FP programs it will be virtually zero). Being in mind this 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:

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

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 Fig. 2 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. 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 order 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 section 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 close to that obtained using classical Reservation Stations; thus well-established CPI studies (like [3], [11], [14] and so on) can apply to chrono-scheduler performance.

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.

IV. The case of variable duration operations and wide issue superscalar processors.

In previous sections we part from the hypothesis that all timing information could be known at issue stage, because all latencies were predictable. In the general case two difficulties have to be overcome. First the case of wide-issue processors, because the calculation of period *Td* (period at which destination register is to be generated) depends on the *Td* of producer instructions, which can be issuing at the same cycle. This occurs for the three integer instructions in Fig. 1. If we do not want to increase the clock period, it is necessary that *Td* calculations of successor instructions wait for the calculations of their producers. The second challenge appears in operations whose duration is variable, like memory access (which depends on hits in hierarchy levels), or complex operations (like DIV, SQRT, and so on). For these instructions, *Td* and T_{EX} cannot be known when issuing. Both problems can be solved with a first level of a chrono-scheduling hardware similar to that explained in previous sections. This level is not to work and "traffic" with data but with RP. Therefore its FUs are composed of the hardware to compute the periods *Td* and T_{EX} of these latency unpredictable instructions.

Let us consider the worst issue case for this first CS level: only one *Td* can be calculated in parallel for two instructions with a RAW. In Fig. 1 symbol __ indicates the stage where RP can be calculated, while symbol –– represents a waiting cycle due to a delay on this calculation. For example, *Td* of the first two loads cannot be precisely determined until we know they hit in L1 cache. Consequently MULTD and ADDD delayed their *Td* calculation because of the RAW they have. Similarly in the second issue group only ADDI can resolve its *Td*, while each of the other INT operations must wait one additional cycle. As a conclusion, from IS stage to *Td* calculation several instructions are waiting in the first CS level. In the second iteration a Load misses L1 cache, so its *Td* calculation is delayed several cycles more (in this figure we have supposed that during the third cycle of L2 access the hit/miss condition is discovered).

Once an operation has determined its *Td* (underlined stage), it is sent to HSs (or SSs) of the second level, where it is programmed to be executed on a predefined cycle. Of course, instructions whose *Td* can be calculated at issue are sent directly to HSs (or SSs).

The associated buses of this first CS level are very much smaller than those of a classical scheduler or the second CS level, because they work with RP values instead of full data. In this sense they can be called "Relative Periods Buses" (RPB). For example, for Fig. 1 the maximum RP is reached in the RAW between ADDD and SD instructions at period T6, and its value is 6 cycles, that is, a width of only 3 bits. Additionally these buses must incorporate a line to indicate if a delay on a variable latency operation had occurred (the "delay line" in Fig. 5). Those stations (or fields in the register scoreboard) that receive this line as activated, must postpone its RP calculation (this matter is explained more deeply below).

Similarly functional units required in the first CS level are very simple; in fact they only calculate RP, so they can be called "Relative Periods Calculation Units" (RPCU). An example of a possible implementation of one RPCU is represented in Fig. 5 (inside the dotted square). 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 two level AND-OR circuitry. The first CS level will be composed of one or more RPCU (like that shown in Fig. 5) for each FU. Others details that are not in the critical path are not shown in this figure.

Let us analyse the working of the first challenge (the existence of a RAW in a group of instructions) through the integer instructions of Fig. 1. At period T3, the producer instruction ADDI can calculate its T_{EX} and *Td* periods (for example at the INTa RPCU), can write its *Td* at the register scoreboard (field "*Td* of Value"), and can occupy a SS (or HS) of the second CS level. During the same cycle the successor SLT detects the RAW and checks that the field "RP of *Td*" is empty: then it concludes that its RP calculations will take place a cycle later. Therefore it occupies the first SS on the first CS level (for example in the INTb RPCU), filling the necessary fields for both operands (REND is supposed to be available). At the same time BNEZ detects a RAW with a previous successor instruction, so its RP calculations must take place two cycles later, and it occupies the second INTa RPCU SS. At period T4 SLT captures the *Td* of ADDI (from the RPB connected to the output of INTa RPCU) and calculates its own *Td*, which is written in the RF scoreboard. Also SLT is sent to the corresponding SS (or HS) of the second CS level. At the same time BNEZ has gone one SS down (and decremented the RP stores in its SS). Finally at T5, BNEZ can capture the *Td* of SLT, calculate its T_{EX}, and be sent to the second CS level.

The case of variable duration operations must include some peculiarities with respect to the previous one. Let us observe the working of the second iteration loads in Fig. 1. At period T4, both loads issue to the second CS level, but they mark in their destination register scoreboard that the RP of *Td* will be computed in two cycles (at T6), due that register RX will be generated one cycle later (the field "*Td* of Value" was previously marked by ADDI instruction). Therefore, their successors MULTD, ADDD occupy the third and fourth SS of their corresponding RPCUs, waiting for the *Td* of loads. As the first load is supposed to hit in L1 cache, the working of the MULTD is similar to that of the integer instructions. But at period T6 second load misses at L1 cache, so its *Td* calculation has to be delayed (in 3 cycles as it was supposed in Fig. 1). Then it fills the LOADa RPCU output with this number 3 and with the "delay signal". In addition the failed load must occupy again another upper SS (the third one in this case), so as to try to resolve its *Td* when accessing to L2 cache. The scoreboard for register FY also sets its field "RP of *Td*" to +3. During the next period, instruction ADDD, which was waiting for the *Td* of this load, observes that the "delay line" is activated and sets the corresponding field in its SS to the quantity indicated in the LOAD RPCU output. At the next period, ADDD reaches the ADDDFP RPCU, but it cannot calculate its T_{EX} and *Td* periods. Therefore it also transmits the "delay signal" to its possible successors (like the instruction SD) and to the register scoreboard, and try to occupy a new upper SS corresponding to this incoming delay. Finally at T9, load instruction hit L2 cache and set the field "*Td* of Value" of the register FY to +3 (that is, T12). At T10, ADDD can calculate its T_{EX} and *Td*, which are sent in a usual manner to the rest of the first CS level.

Three details must be taken into account in the design of the first CS level. Firstly, if one of the delays in a latency variable operation were very big (more than the implemented number of SS), a small pool of HS will be required. A big number of HS in this pool is not necessary, because they are only going to be used in infrequent cases (for example after a L2 miss if detecting the L3 hit/miss condition were too slow).

Secondly as RPCUs are simple it seems to be preferable to implement a sufficient number of them to avoid structural conflicts to the maximum. Therefore reservation tables to check these conflicts would be avoided. In contrast this means that the system would have to stall in the improbable case of a lack of RPCU. Finally, the occupancy of SSs can be originated in new issued instructions as well as in those RCPU outputs that had detected a delay.

Ongoing work includes doing simulations using standard benchmarks to estimate the number of RPCUs necessary to make negligible the structural stalls (due to the lack of SS), and selecting several processors where CS could fit as a cheap dynamic scheduler to write an VHDL implementation. 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.

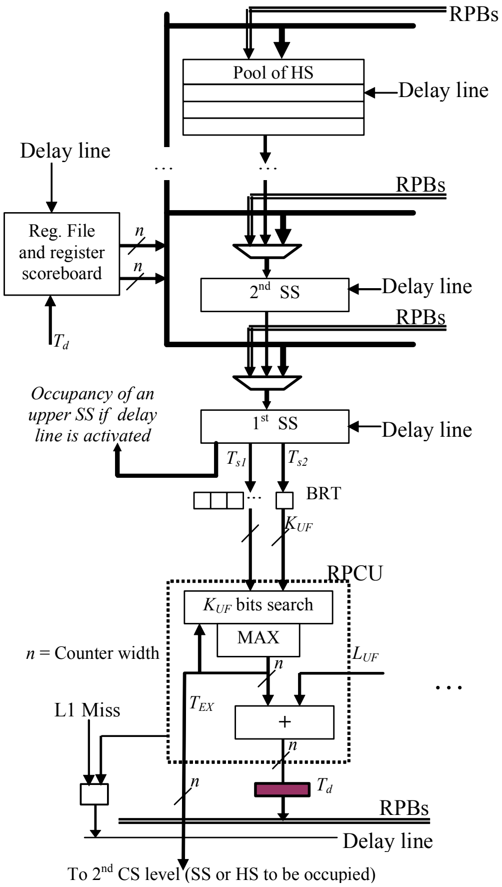

Fig. 5: First level Chrono-scheduler implementation.

Register Number	Value (if available)	*Td* of Value (if available)	Value Producer FU (if available)	RP of *Td*	Producer RPCU
RX	-	T5 (+2)	INT1	-	-
R3	-	-	-	T4 (+1)	INTb RPCU
REND	Value	-	-	-	-

Fig. 6: Register scoreboard fields (filled up for integer register at T3)

V. Conclusions

The general case of a two-level chrono-scheduler is first presented. It avoids the usual complexity found in reservation stations (RS) of classical dynamic schedulers, because it extracts timing information during the issue and 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 much simpler (no compare logic nor CAM, no priority circuitry, and first level stations have only one operand and one datapath instead of two for each RS). The first level contains simple functional units composed of small adders and BRTs. On the whole, it is apparent that CPI for CS will be similar to that from 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.

References

[1] Tomasulo, R. M. "An efficient algorithm for exploiting multiple arithmetic units". *IBM J. Res. & Develop.,*, 11: pp 25–33, Jan. 1967.

[2] J.E. Thornton. "Parallel operation in Control Data 6600". *Proc. AFIPS Fall Joint Computer Conference*, number 26, part 2, 1964, pages 33-40.

[3] J.L. Hennessy, D.A. Patterson. *Computer Architecture. A Quantitative Approach*. Morgan-Kaufmann (2nd Edition), 1996.

[4] Moudgill, M.; Pingali, K.; Vassiliadis, S.; "Register renaming and dynamic speculation: an alternative approach". *Proceedings of the 26th Annual International Symposium on Microarchitecture,* 1993. 1-3 Dec. 1993, pp. 202 – 213.

[5] D. Sima, "The Design Space of Register Renaming Techniques," *IEEE Micro*, vol. 20, no. 5, Sept./Oct. 2000, pp. 70-83.

[6] Ponomarev, D.V.; Kucuk, G.; Ergin, O.; Ghose, K.; Kogge, P.M. "Energy-efficient issue queue design". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* Volume 11, Issue 5, Oct. 2003, pp. 789-800

[7] S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," *Proc. 24th Ann. Int'l Symp. Computer Architecture (ISCA '97)* , June 1997, pp. 206-218.

[8] D. Ernst, T. Austin. "Efficient Dynamic Scheduling Through Tag Elimination". *Proc. Of the 29th Annual Symposium on Computer Architecture (ISCA'02).* Pages: 37 – 46. 2002.

[9] Ramirez, M.A.; Cristal, A.; Veidenbaum, A.V.; Villa, L.; Valero, M. "Direct Instruction Wakeup for Out-of-Order Processors". *Innovative Architecture for Future Generation High-Performance Processors and Systems,* 2004. Proceedings. Jan. 2004, pp.2-9.

[10] Díaz-del-Río, F., Sevillano, J.L. , Vicente, S. , Jiménez-Moreno, G., Civit-Balcells, A. "Chrono-Scheduling: A Simplified Dynamic Scheduling Algorithm For Timing Predictable Processors". Accepted for publication in *Journal of Circuits, Systems, and Computers*.

[11] Önder, S. and R.Gupta, "Instruction Wake-Up in Wide Issue Superscalars", *Euro-Par 2001, LNCS 2150, 2001. Springer-Verlag Berlin.* Heidelberg 2001, pp. 418–427.

[12] Sanjeev Banerjia, Sumedh W. Sathaye, Kishore N. Menezes, and Thomas M. Conte, "MPS: Miss-Path Scheduling for Multiple-Issue Processors". *IEEE Trans. Computers,* Vol. 47, No. 12, December 1998. pp 1382-1397.

[13] Diep, T.A., Nelson,C. and Shen, J.P. "Performance Evaluation of the PowerPC 620 Microarchitecture". *Proc. Of 22nd Annual Intern. Symposium on Computer Archit. (ISCA '95)*, pp.163-174. Italy.

[14] Folegnani, D. and González, A. "Energy-Effective Issue Logic". *Proceedings of the 28th Annual Intern. Symposium on Computer Architecture,* p.230-239, July 2001, Sweden.