High-Performance Low-Vcc In-Order Core

Jaume Abella, Pedro Chaparro, Xavier Vera, Javier Carretero, Antonio González
Intel Barcelona Research Center, Intel Labs Barcelona - UPC
jabella@ac.upc.edu, xavier.vera@intel.com,
javier.carretero.casado@intel.com, antonio.gonzalez@intel.com

Abstract

Power density grows in new technology nodes, thus requiring Vcc *to scale especially in mobile platforms where energy is critical.*

This paper presents a novel approach to decrease Vcc *while keeping operating frequency high. Our mechanism is referred to as immediate read after write (IRAW) avoidance. We propose an implementation of the mechanism for an Intel® Silverthorne^{TM} in-order core. Furthermore, we show that our mechanism can be adapted dynamically to provide the highest performance and lowest energy-delay product (EDP) at each* Vcc *level.*

Results show that IRAW avoidance increases operating frequency by 57% at 500mV and 99% at 400mV with negligible area and power overhead (below 1%), which translates into large speedups (48% at 500mV and 90% at 400mV) and EDP reductions (0.61 EDP at 500mV and 0.33 at 400mV).

1 Introduction

New CMOS process generations enable smaller transistors, thus increasing integration densities. Extra transistors in chips are used to increase features and reduce form factors of portable devices (mobile phones, mobile internet devices, netbooks, etc.). However, those trends reduce drastically battery lifetime due to extra energy required for new features and reduced battery size and capacity. Thus, technology scaling enforces the use of a lower *Vcc* to keep and even increase battery life in portable devices.

Those processors in the mobile market segment make an aggressive use of Dynamic Voltage and Frequency Scaling (DVFS) techniques to adapt their *Vcc* and frequency to the current workload and battery state [10, 24] for the sake of energy efficiency. Thus, flexible and energy-efficient designs are required to enable high performance at both high and low *Vcc* operation.

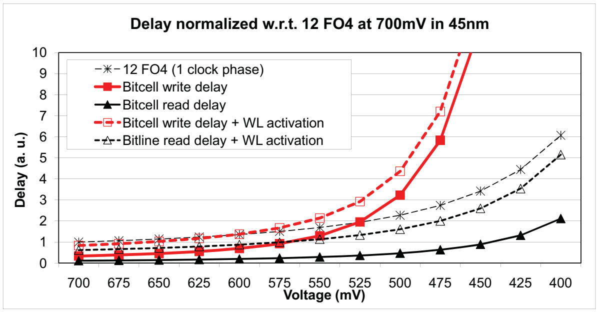

Figure 1. Delay of a clock phase (estimated as 12 FO4 gates), and write and read delay for an 8-T SRAM bitcell [19]. WL stands for wordline

Cycle time is determined by the slowest path in the processor. Such critical path is typically part of the logic and its delay can be modeled accurately as a chain of N fanout-of-4 (FO4) inverters. However, write operations in SRAM arrays become the most critical path at low *Vcc* because their latency grows exponentially when *Vcc* is decreased. We illustrate this trend in Figure 1 (further details on the experiment are provided later).

Bitcell write latency (the main component of write latency) can be somewhat mitigated by process optimization and bitcell redesign (e.g., using cells other than conventional 6-T [9, 17–19]). However, such techniques come at some area cost for SRAM blocks. Moreover, even if bitcell redesign provides write latency reductions, write latency is still the slowest path at low *Vcc* levels (e.g., 525mV in our experiment) and drastical operating frequency reductions are still required at low *Vcc*.

To the best of our knowledge this paper presents the first processor design that enables high operating frequency at low *Vcc* by overriding SRAM write delay constraints. Some state-of-the-art solutions work for particular structures, but they cannot be applied to all SRAM blocks in a core. This paper addresses this issue with a flexible solution and its

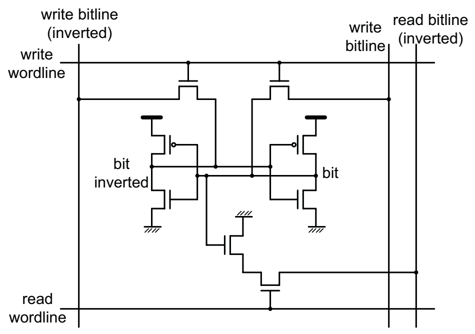

Figure 2. Scheme of an 8-T bitcell [19]

implementation for an Intel® Silverthorne™ in-order core that operates at high frequency even at low *Vcc*. Based on the observation that write operations can finish properly if they are interrupted early and given that most of the data written in SRAM structures are rarely read immediately after being written, we interrupt write operations before bitcells reach a readable state. This way operating frequency is increased. Those bitcells stabilize and reach a readable state across several cycles by avoiding conflictive read operations. We refer to our strategy as Immediate Read After Write (IRAW) avoidance. Infrequent stalls introduced by IRAW avoidance are largely offset by the performance increase caused by the operating frequency boost.

The rest of the paper is organized as follows. Section 2 presents a detailed analysis of the impact of low *Vcc* in delay and existing solutions. Section 3 introduces IRAW avoidance strategies. Section 4 describes the implementation of the IRAW avoidance method for the main blocks of an Intel® Silverthorne™ core. Results are presented in Section 5. Section 6 reviews some related work. Finally, Section 7 summarizes this paper.

2 Low Vcc Operation Overview

This section describes the impact of low *Vcc* operation in cycle time and reviews existing solutions to increase operating frequency at low *Vcc* operation.

2.1 Impact of Vcc in the Cycle Time

Effective bitcell write operations are performed during half of the clock cycle (a clock phase). During the first phase of the cycle decoding and bitline set up are performed. During the second clock phase some time is devoted to activate the wordline and most of the time is used to effectively update the bitcell. In Figure 1 we show the latency for bitcell write and read operations with and without considering the wordline activation time, as well as the latency for a chain of 12 FO4 gates (typical latency for combinational circuits during a single clock phase). Read and write delays depicted in Figure 1 correspond to an 8-T bitcell like the one shown in Figure 2 with double-bitline write ports and single-bitline read ports. We have chosen such cell because it is the one used for the SRAM blocks of the Intel® Silverthorne™ (including register files and cache memories). *Vcc* and operating frequency trends are based on electrical simulations for the *Vcc* range [700mV,400mV] in an Intel® simulator. The process technology used to gather such data is 45nm considering process variations in the geometry of the transistors and the threshold voltage. The level of process variations considered is 6σ, which is a reasonable margin to ensure fault-free operation (only one critical path per billion would not fit the cycle time). Threshold voltage (*Vth*) has been scaled based on trends for super-*Vth* and near-*Vth* operation [8]. Any delay is measured until the observed signal completes 80% of its swing (e.g., *Vcc* is 600mV and the signal falls below 120mV or raises above 480mV). Note that read and write delays are shown with and without wordline activation delay. Such delay depends on the particular characteristics of the SRAM array (mainly the number of bits per wordline). Our experiments correspond to an array with 1,024 entries, 32 bits per entry and 8 bits attached to each wordline (wordlines have been partitioned into 8 bit groups to optimize their delay).

As shown in the figure, most of the delays grow almost linearly, however this is not the case for write operations whose delay grows exponentially. Write latency is the most critical path below 525mV if wordline activation is neglected. If we consider wordline activation latency then write operations become the most critical path at 600mV. For instance, frequency must be decreased down to 77% of the frequency allowed by the logic at 550mV due to write delay constraints, and down to only 24% at 450mV. Wordline activation delay is low and its slope resembles that of the 12 FO4 chain. Read delay remains below that of the 12 FO4 inverters even if wordline activation delay is considered (thick dotted lines) because 8-T cells allow sizing properly those transistors feeding the read bitline without harming write delay.

We can conclude that bitcell write delay is the most critical path at low *Vcc* and impacts cycle time dramatically.

2.2 State-of-the-Art on Overriding SRAM Write Delay

Some techniques can be used to operate at a higher frequency than that dictated by write delay (i.e., up to the frequency dictated by the chain of FO4 inverters). Based on the fact that delay for each bitcell differs due to process variations, an alternative may consist of reducing the number of σ considered to determine the cycle time (e.g., using 4σ instead of 6σ). Such a solution will increase the num-

	Works for all SRAM blocks	Adapts to multiple Vcc	Hw. ovh.	Large IPC impact	Hard to test
Faulty Bits	NO	YES (costly)	LOW	YES	YES
Extra Bypass	NO	NO	HIGH	YES	NO

Table 1. Characteristics of state-of-the-art techniques to override SRAM write delay

ber of faulty bits and hence, faulty bits (or groups of bits) should be disabled. This technique has been proposed for cache memories [1, 22, 26]. We refer to it as *Faulty Bits*. A different alternative consists of increasing frequency above the one required to perform write operations and pipelining write operations across more than one cycle. If cycle time has been shortened, bitcell update does not finish during a single clock phase and it must last one or more cycles. We refer to this approach as *Extra Bypass*. Table 1 describes both approaches based on the following characteristics:

- **Work for all SRAM blocks**. *Faulty Bits* cannot be used for those structures that require all entries to operate as for instance the register file for an in-order core. *Extra Bypass* requires knowledge of whether data in the bypass will be used, which is not be feasible for cache-like structures where addresses may be known too late.

- **Adapt to multiple Vcc**. *Faulty Bits* works for multiple *Vcc* levels if fault maps can be updated properly. However, either SRAM blocks must be tested at every *Vcc* level change or extra storage is required for as many maps as *Vcc* levels allowed. *Extra Bypass* is not flexible because bypasses must be in place (and their costs paid) at any *Vcc* level.

- **Hardware overhead**. *Faulty Bits* require fault maps and rather small logic. The cost of the fault maps may not be negligible. *Extra Bypass* has prohibitive costs [3, 4, 20] due to the extra wires and latches (up to 128 or 256-bit latches for SIMD data). Moreover, those bypasses affect critical paths.

- **Large IPC impact**. *Faulty Bits* disables faulty storage, and hence, miss rates increase for caches leading to lower performance. Similary, *Extra Bypass* extends write operations across multiple cycles, causing significant write port contention. Extra write ports can be set up, but their cost in power, area and delay is huge.

- **Hard to test**. *Faulty Bits* introduces indeterminism during post-silicon testing. This issue is especially relevant in multi-cores where testing costs are reduced by running the same patterns in several cores and comparing their outputs periodically. Whenever a pair of cores does not provide the same outputs simultaneously there is no way to identify whether the discrepancy is due to an error or due to performance variations introduced by disabled hardware in one of the cores. *Extra Bypass* does not introduce new testing issues.

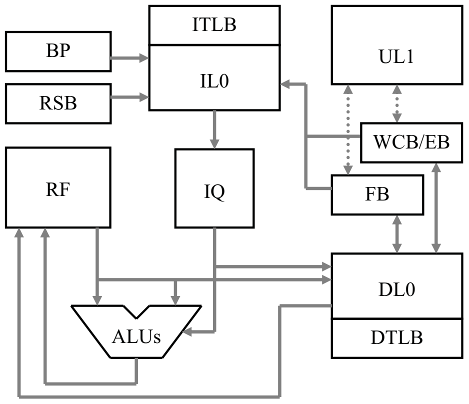

Figure 3. Schematic of the main blocks of an Intel® Silverthorne^{TM} Microarchitecture [6]

Summing up, mechanisms are required to increase operating frequency at low *Vcc* for all SRAM blocks, providing flexibility to the different *Vcc* levels, with reduced hardware overhead and IPC degradation, and without affecting post-silicon testing.

3 Strategies for IRAW Avoidance

This section presents our strategies to increase operating frequency at low *Vcc* beyond the frequency determined by write delay. First, we classify SRAM structures of an in-order core, and then, we present the IRAW avoidance strategy for each category.

3.1 In-Order Core SRAM Structures

The processor we consider in this work is the Intel® Silverthorne^{TM} [6] although our implementation works for any in-order core. The schematic of the main blocks of this core is depicted in Figure 3. As shown, we consider first level data cache (DL0), first level instruction cache (IL0) and second level cache (UL1) memories, data and instructions TLBs (DTLB and ITLB respectively), joint write combining and eviction buffers (WCB/EB), fill buffers (FB), instruction queue (IQ), register file (RF), branch predictor (BP) and return stack buffers (RSB).

In order to present strategies and implementations for the different blocks of the core, we classify them into

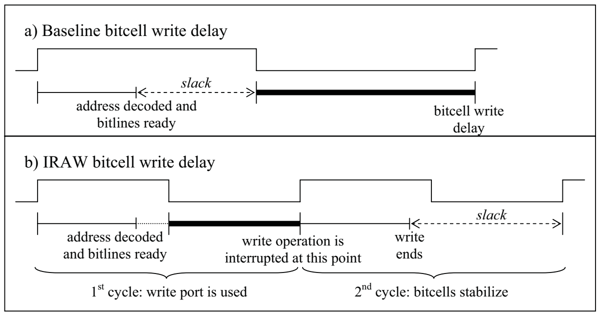
a) Baseline bitcell write delay
slack
address decoded and bitlines ready
bitcell write delay
b) IRAW bitcell write delay
slack
address decoded and bitlines ready
write operation is interrupted at this point
write ends
1st cycle: write port is used
2nd cycle: bitcells stabilize

Figure 4. Example of timing of a conventional bitcell write operation (top plot) and an interrupted one (bottom plot)

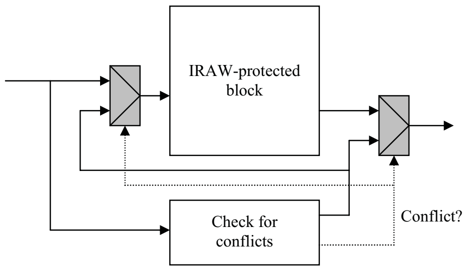

Figure 5. Example where reads of conflicting locations are not avoided but repaired

the following categories: (1) Register file; (2) Instruction queue; (3) Unfrequently written cache-like blocks: IL0, UL1, DTLB, ITLB, WCB/EB and FB; (4) Frequently written cache-like blocks: DL0; and (5) Prediction-only cache-like blocks: BP and RSB. For this last type of blocks it is not relevant whether they are written often or not.

3.2 IRAW Avoidance Strategies

We propose a set of IRAW avoidance strategies, which rely on the fact that write operations can be interrupted before bitcells reach a readable state by disabling wordlines. Figure 4 illustrates how cycle time can be shortened. The top plot shows the baseline case where the cycle lasts until write operations complete. At low *Vcc* bitcell write delay is the cycle time limiter and therefore, there is large slack during the first clock phase to decode the address and prepare bitlines. However, bitcell write delay requires the full second clock phase. The bottom plot shows our approach where write operations are interrupted early by deactivating wordlines. Some extra time is required to reach a readable state (almost a full clock phase in the example), and the total bitcell update delay (effective bitcell write plus stabilization) may increase with respect to the baseline case because bitcells must complete their flip by their own (there is no further help from the bitlines). However, as we show later, performance gains due to operating frequency increase largely offset performance decrease caused by delaying by one cycle data consumers.

Write interruption does not affect correctness as long as some properties are guaranteed: (i) bitcells written have their wordlines activated during some time, (ii) bitlines reach the proper *Vcc* level before wordlines are deactivated and keep such level until wordlines are deactivated, and (iii) bitcells flip their contents to some extent before the wordline is deactivated. Although bitcells may not be readable when wordlines are deactivated (e.g., internal nodes have not completed 80% of their swing), their contents have flipped enough so that they will stabilize and reach a readable state.

By interrupting write operations in this way we achieve the following: cycle time can be shortened (and operating frequency raised) and write ports are used only during one cycle so no new resource conflicts are caused. A mechanism is required to avoid read operations on not-yet stabilized data because otherwise data retrieved could be wrong and bitcell contents could be destroyed. Such mechanism must delay conflicting read operations by one or few cycles. However, new write operations can start every cycle because the write port is used during a single cycle.

Mechanisms to avoid IRAW depend on the type of structure. The strategy to follow for each type of structure is as follows:

- **Register file**. Source registers are known before instructions are issued. Such information can be used to avoid issuing any instruction one of whose sources would be read when it is stabilizing in the register file. Note that given that back-to-back execution is still allowed because values are obtained from the existing bypass network if they are consumed immediately.

- **Instruction queue**. Instructions are allocated and issued in-order in in-order cores. Thus, we only need to guarantee that instructions are not issued right after they have been allocated. By doing so we avoid any IRAW in the IQ.

- **Unfrequently written cache-like blocks**. Given that these blocks are written rarely, the simplest solution consists of delaying read accesses until contents have stabilized because few stalls will arise.

- **Frequently written cache-like blocks**. Locations to be read may be obtained immediately before the access must be performed. Thus, it is very unlikely we can avoid IRAW without delaying many read operations that would not access not-yet stabilized data but whose address is not known in advance. Similarly, delaying read operations until all contents have stabilized introduces many stalls because such blocks are written frequently. Thus, we propose tracking those few values not-yet stabilized and checking *a posteriori* or in parallel whether an IRAW happened. If that is the case, then simple recovery actions can be taken. Figure 5 depicts this mechanism.

- **Prediction-only cache-like blocks**. The same solution employed for frequently written cache-like blocks would work for prediction-only blocks. However, if determinism is not required, we can simply ignore IRAW violations because those violations may alter predictions, and hence performance, but not correctness.

4 IRAW Avoidance Implementation

This section describes particular implementations of the IRAW avoidance strategies for the main blocks of the in-order core studied. Beyond particular implementations, the purpose of this section is showing that our strategies can be applied to any SRAM block in an in-order core.

4.1 Implementation for Register Files

This section presents an implementation of our IRAW avoidance method for the RF. By introducing minor modifications in the issue logic we avoid issuing instructions in those cycles when they would consume those not yet stabilized registers. Since instructions causing an IRAW are infrequent (13.2% of the total instructions), the issuance of those instructions is delayed and thus, performance degradation is low. Moreover, the IRAW avoidance mechanism can be easily deactivated during high *Vcc* operation to prevent any performance degradation.

This section describes the issue logic for in-order cores and the modifications required in the issue logic to implement the IRAW avoidance method.

4.1.1 Readiness Control Logic in the Instruction Queue

Typically, readiness of registers is tracked by means of a centralized scoreboard. Each entry in the scoreboard is devoted to a logical register, as shown in Figure 6. Each entry consists of a shift register. The purpose of the shift register is dealing with delayed wake-up. The most significant bit of the shift register indicates whether the logical register is ready. For instance, if the shift register has 5 bits, an instruction whose latency is 3 cycles will set the shift register of its destination logical register to **00011**. Shift registers

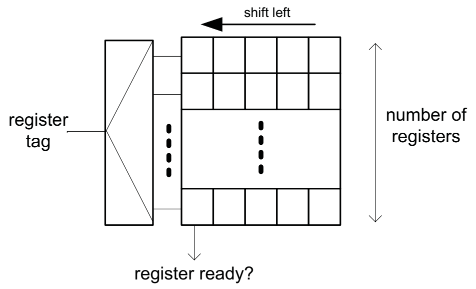

Figure 6. Readiness control logic of a register in the scoreboard of an in-order processor

are shifted every cycle one position keeping the same value in the least significant bit. This way, the shift register will store **00111** after one cycle, **01111** after two cycles, and **11111** after three cycles indicating that any consumer does not need to wait anymore for this logical register because the most significant bit is **1**. In general, shift registers of B bits can deal with the delayed wakeup of any instruction whose latency is up to $B - 1$ cycles. Those instructions update the shift register of its destination logical register as soon as they issue.

In our example only those instructions whose latency is greater than 4 cycles will need an additional mechanism to indicate readiness of their destination logical registers. The destination logical register of a long-latency instruction will keep its shift register set to **00000** when such instruction is issued. Long-latency instructions must be tracked in a separate scoreboard (e.g., FP division) or an event will be generated when they finish their execution (e.g., a load miss). At some point in time the shift register of the destination logical register is updated. Typically, it is updated when the value becomes available (the shift register is updated as if the producer was a single cycle instruction, **11111** in the example), or when the value is expected to be available in less than B cycles (the shift register is updated properly to indicate that the logical register will be ready in few cycles).

4.1.2 Modifications in the Control Logic

The purpose of our mechanism consists of delaying the issuance of those instructions causing an IRAW violation. We will use the example in Figure 7 to describe the IRAW avoidance mechanism. The example corresponds to a pipeline with 1 bypass level where IRAW must be avoided for 1 cycle (N=1). As shown in Figure 7, right after the value is produced there is no conflict with the consumers because such value is available through the bypass network. This is the case for *Consumer 1*, which obtains the value through the bypass network in place. Conversely, *Consumer 2* would read the value from the register file right after it has

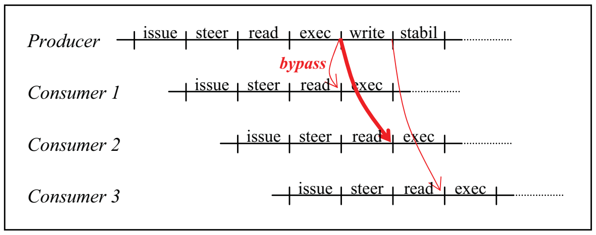

Figure 7. Example of consumers obtaining inputs from their producer in different cycles

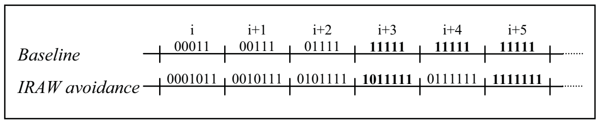

Figure 8. Ready vector for an input operand. Bold values indicate that the operand is ready

been written. In order to enable low *Vcc* operation we must prevent instructions from reading values immediately after they have been written, which is the case of *Consumer 2* in the example. Finally, from *Consumer 3* onwards there is no conflict because the register has stabilized.

In order to avoid IRAW violations we delay the issue time of conflictive instructions. Our proposal is based on extending the shift registers in the scoreboard with few more bits. In particular, the number of extra bits required corresponds to the number of bypass levels (1 bypass level in our example) plus the number of cycles required for the bubble (1 cycle in our example) to avoid reading values in conflicting cycles (total of 2 bits in the example in Figure 7). Whenever the producer instruction is issued, the shift register is initialized as follows from left to right: (I) as many zeros as cycles takes the producer to execute (like in the baseline case), (II) as many ones as bypass levels are available, (III) as many zeros as cycles we want to delay read operations (N), and (IV) as many ones as needed to fill the remaining bits (like in the baseline case).

As shown, the only difference in the shift register between our proposal and the baseline is the fact that we include the bits described in (II) and (III). Note that such bits are set automatically like the rest of the bits of the shift register because both the number of bypass levels (case II) and the size of the bubble (case III) have been set at design time. In our example in Figure 6 and assuming one level of bypass in place, a 3-cycle instruction will initialize the shift register of its destination logical register with **0001011** because the producer takes 3 cycles to execute (**000**), there is 1 bypass level (**1**), the bubble is 1 cycle (**0**), and after that the value is always available (**11**). As shown in Figure 8, our approach prevents consumers to issue in the cycle $i + 4$ when they would read the operand from the register file immediately after it has been written.

The proposed mechanism requires rather modest hardware overhead and does not need expensive mechanisms to detect/reissue wrongly issued instructions because instructions are not allowed to issue in conflicting cycles. This is particularly important in in-order processors because such kind of mechanisms are not in place. Other approaches perform similar modifications in the shift registers of the issue logic to simplify bypass networks and use incomplete networks [3]. Since the modifications required by both approaches, our IRAW avoidance mechanism for the register file and the one based on incomplete bypass networks, are pretty similar, they can be combined sharing the overheads in a synergistic manner. Note that incomplete networks [3] cannot increase operating frequency by themselves, whereas our technique based on interrupting write operations early enables such operating frequency boost.

4.1.3 Multiple Vcc Operation

Our IRAW avoidance mechanism must be reconfigured dynamically to maximize performance at any *Vcc* level. Thus, the number of IRAW cycles required must be updated according with the *Vcc* level when it changes. The way to adapt our IRAW avoidance mechanism to the different scenarios is fairly simple since it is just a matter of setting the bits of the shift registers to the right value. For the sake of clarity we illustrate this explanation with the example of the 3-cycle instruction in a processor with one level of bypass and shift registers considering up to 4-cycles instructions. For the different *Vcc* levels scoreboard entries are initialized as follows:

- **575mV or lower.** The shift register is set to **0001011**. The zeros in the fifth position corresponds to the cycle when the register is stabilizing.
- **600mV or higher.** The shift register is set to **0001111**. No extra cycle for stabilization is required and hence, IRAW avoidance is deactivated by setting properly the shift register.

4.2 Implementation for Instruction Queues

Once instructions are decoded, they are allocated to the IQ where they remain until they are issued. The IQ of an in-order core considers only the oldest instructions for issuance. For instance, Intel® Silverthorne™ [6] considers the 2 oldest instructions. To simplify the design of the IQ, those entries are read every cycle independently of whether they contain ready instructions or even they are valid.

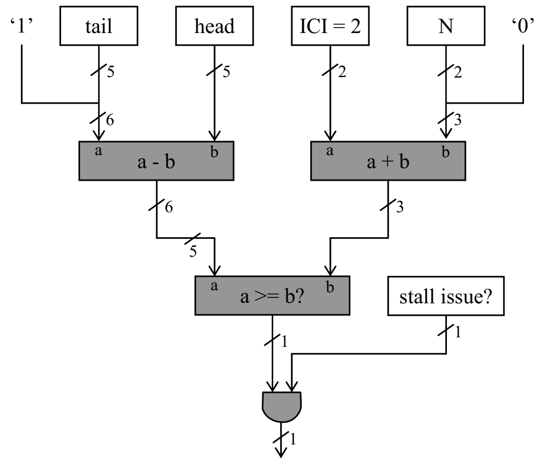

Figure 9. Logic to avoid IRAW in the IQ for a Intel® Silverthorne™ Microarchitecture [6]

The number of instructions considered for issuance is referred to as ICI in the rest of the section. Instructions are allocated to the IQ at a given rate of AI (allocated instructions) per cycle. The number of cycles required by the instructions to stabilize after they have been written is N (typically 1 cycle). Based on these parameters, ICI, AI and N, our implementation to avoid any read from not-yet stabilized entries of the IQ is based on allowing issuance of instructions if and only if the following condition holds:

$$occupancy \geq ICI + AI \cdot N \quad (1)$$

Occupancy stands for the number of instructions in the IQ, which is obtained substracting the tail and the head.

As shown, our mechanism allows issuance if there are at least $ICI + AI \cdot N$ instructions in the IQ, thus ensuring that even if there are $AI \cdot N$ not-yet stabilized instructions in the IQ (the maximum number of them), the ICI oldest instructions have stabilized. However, some performance may be lost because it may happen that there are at least ICI stabilized instructions and issuance is not allowed. As shown later, performance impact is low.

The logic required to implement the mechanism is depicted in Figure 9. The leftmost part of the plot shows how to compute the occupancy of the IQ, whereas the rightmost part shows how to compute the occupancy threshold. In order to obtain the occupancy, we increase the $tail$ by $IQsize$ (32 in our particular implementation), which is done by appending a '1' to the left of the $tail$. Then, we discard the uppermost bit of the substraction to compute the modulus operation. Appending a '0' to the right of N corresponds to multiplying N by AI because AI is 2 in our particular implementation. The *stall issue?* signal is set to '0' only when write operations fit into a single cycle, and therefore, the IRAW avoidance mechanism must be disabled. Note that the occupancy threshold is recomputed only at each *Vcc* level change because it is the only time when N is changed. As shown in the picture, whenever *Vcc* is changed, only N and *stall issue?* signal must be updated to fit the best configuration of the IQ.

Whenever the pipeline must empty (e.g., due to an exception), $AI \cdot N$ NOOP instructions are injected in the IQ to ensure all instructions are issued.

4.3 Implementation for Unfrequently Written Cache-like Blocks

Cache memories have limited capacity and associativity, and hence, some misses may happen. Whenever a miss happens some contents must be replaced. Since write operations may need several cycles to complete, any access to cache during those cycles may destroy the contents of the recently filled entry. Contents may be destroyed even if the contents requested correspond to a different entry because of the particular implementation of some caches. In general, whenever we access a given address in a set-associative cache, all entries in the corresponding set are accessed simultaneously to reduce latency. Should an entry in such set be stabilizing, its contents and tag may be destroyed even if its address does not match the address of the on-going cache access.

Given that cache misses are infrequent in most cache memories (e.g., IL0, UL1, DTLB, ITLB), the solution we adopt is fairly simple: *in case of a fill we stall any access to cache*. Whenever the corresponding entry is filled, further accesses are not allowed during the number of cycles required to stabilize. Typically, preventing further accesses during some cycles is as easy as keeping the ports busy to prevent the port arbiter from issuing new accesses.

The same principle can be applied to the WCB/EB and the FB, which deal with data comunicated between DL0/IL0 and UL1.

Reconfiguring the mechanism to stall accesses a different number of cycles depending on the current *Vcc* level is fairly simple. Ports must be stalled during a number of cycles indicated by a counter, whose initial value is updated whenever the *Vcc* level is changed. Note that the overhead of such counters is negligible given the fact that they may need to work with small values (as shown later, 1 or 2 bits suffice).

4.4 Implementation for Frequently Written Cache-like Blocks

The only frequently written cache-like block in the in-order core considered is DL0. Similarly to the other cache-like structures, DL0 is written on cache line fills, which

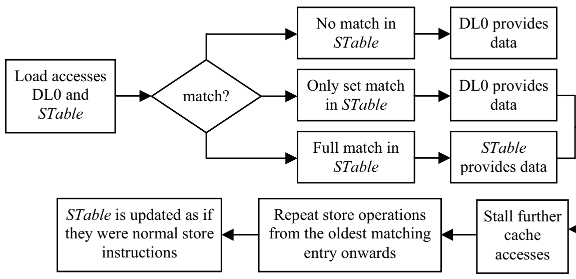
Load accesses DL0 and *STable*
match?
No match in *STable*
DL0 provides data
Only set match in *STable*
DL0 provides data
Full match in *STable*
STable provides data
Stall further cache accesses
Repeat store operations from the oldest matching entry onwards
STable is updated as if they were normal store instructions

Figure 10. Actions taken for load instructions

are unfrequent. The way to avoid IRAW in this scenario is the same as for the unfrequently written cache-like structures: stalling further accesses while write operation finishes. However, DL0 is also modified by store instructions, which are much more frequent than cache misses, and therefore, simply stalling cache accesses for few cycles after store instructions access DL0 would cause a significant performance drop. Thus, we have devised a particular mechanism for DL0 caches to manage store operations, which can be used for any other frequently written cache-like block.

Our mechanism requires a table (Store Table or *STable* for short). Such table tracks the address and data stored in DL0 by any access different to cache line fills. For instance, if only one store instruction can commit per cycle and write operations require 2 cycles to stabilize, the *STable* has 2 entries. Each entry holds a valid bit, a memory address and space for the maximum data size that can be stored. The *Stable* is implemented by means of latch cells to guarantee that it can operate in a single cycle at low *Vcc*. The *STable* is updated as follows: Every cycle as many entries as store instructions can commit are replaced with address and data of those stores writing in cache in the current cycle (if any). If new store instructions do not exist, the corresponding entries are simply invalidated. Entries are selected in a round-robin fashion in such a way that those entries corresponding to store instructions that have just stabilized are the ones replaced.

Load instructions access both the DL0 and *STable* in parallel, and two different situations may arise (see Figure 10):

- Load address does not match any address of any valid entry in *STable* (this is the most common case). Further actions are not required.

- Load address matches either at least one address or only the set of valid entries in *STable*. Full address matches correspond to load instructions reading data that has been recently stored, whereas set-only matches correspond to load instructions reading some data located in the same set as some recently stored data. Since all cache lines in a set of a set-associative DL0 cache are read simultaneously to reduce latency, not-yet stabilized data can be read and destroyed even if such data are not requested by the load instruction. If load address matches the full address, the *STable* provides data to the load instruction. Otherwise, DL0 provides data. Then, further accesses to cache are stalled, and store instructions in the *STable* are repeated from the oldest matching entry onwards to restore correct state. Those repeated store actions further update *STable* to keep it consistent. Note that partial and full address matches are very unlikely because they only arise when a load instruction is issued right after a store instruction accessing the same cache set.

Store instructions only update *STable* at commit time, but they do not cause any other issue because they simply read tags, which are stabilized for sure (no store instructions could modify tags), and write data. Even if the data in the updated location were still stabilizing, correctness is guaranteed because data are not read but updated.

Reconfiguring the mechanism to the current *Vcc* level is fairly simple. The *STable* must have the size required by the largest number of IRAW cycles allowed. The *Vcc* controller sets the number of entries that must be checked (as many as IRAW cycles for the current *Vcc* level). The remaining entries are disabled.

As explained in Section 2 alternative mechanisms based on trading off frequency for faulty bits can be used for this type of blocks [1, 22, 26]. However, it is unclear how to reconfigure those approaches for multiple *Vcc* levels and how to deal with undeterminism during testing. Moreover, there is some performance loss due to extra cache misses. Note that our approach for the DL0 causes small performance degradation (0.3% as shown later). Nevertheless, both IRAW avoidance and allowing faulty bits can be combined to further increase DL0 operating frequency if required.

4.5 Implementation for Prediction-only Cache-like Blocks

Some blocks such as the BP and RSB hold predictions, which can be wrong. Thus, we may neglect IRAW effects and allow read operations to access freely these structures even if they access not-yet stabilized entries, because correctness is not affected. In the case of the RSB, it is written whenever a function call is performed, and read when returning from such function. Therefore, only if call and return happen within a very short number of cycles (e.g., between 1 and 2 cycles) the predicted return address could be corrupted. However, we did not find any short function meeting those conditions. Similarly, a BP entry is updated on every branch execution and read on every branch prediction. Since the number of entries in the BP is large and only those entries whose uppermost bit is flipped could be corrupted, having a conflict in few cycles is extremely unlikely (we observed a negligible 0.0017% average potential extra misprediction rate). Overall, no changes are required to keep the BP and the RSB working properly at low *Vcc* levels.

The only issue related to those errors in the BP and RSB is the fact that they introduce undeterminism in execution, which is detrimental for testing. For instance, if two cores in a multi-core are tested simultaneously to compare their progress, undeterminism in the BP and RSB could make both cores to progress differently, and therefore, their state could not be compared at a given cycle. If determinism is mandatory, the RSB should be stalled after a call instruction, which is very unlikely to delay any instruction. Similarly, the BP should incorporate some hardware to track recent updates as it is the case for the DL0.

5 Evaluation

This section evaluates our in-order core with our IRAW avoidance approach in place. We introduce the evaluation framework and present some results on performance, energy and area impact for our technique. The IRAW avoidance mechanism is compared against the scenarios where frequency is scaled down to allow write operations to happen in a single cycle.

5.1 Evaluation Framework

Performance results have been collected from trace-driven Intel® production simulators. Our workload consists of 531 traces of 10 million consecutive instructions each, which were obtained from different wide variety of programs (Spec2006, Spec2000, kernels, multimedia, office, server, workstation, etc.).

Leakage for the whole processor has been set to 10% of the total energy consumption at 600mV, which is consistent with the trends shown in [8]. Area overhead has been estimated based on the size of the extra bits required to implement our IRAW avoidance scheme assuming latch-size bits [16, 23]. Based on the fact that the extra hardware is rather small (below 0.1% as shown later), power overhead has been estimated simply assuming a pessimistic 20X activity factor for the extra hardware.

5.2 Performance Impact

Figure 11 (a) (squares) shows that cycle time grows quickly as *Vcc* decreases due to write delay increase. Note that differently to Figure 1 we depict cycle time (24 FO4 full clock) instead of a clock phase (12 FO4). For instance, cycle time almost doubles at 500mV with respect to the case where the cycle time is not constrained by write operations. Conversely, IRAW avoidance enables much higher operating frequency (e.g., 57% increase at 500mV and up to 99%) as shown in Figure 11 (b). Such operating frequency boost leads to a significant performance increase (e.g., 48% at 500mV and up to 90%) as depicted in Figure 11 (b).

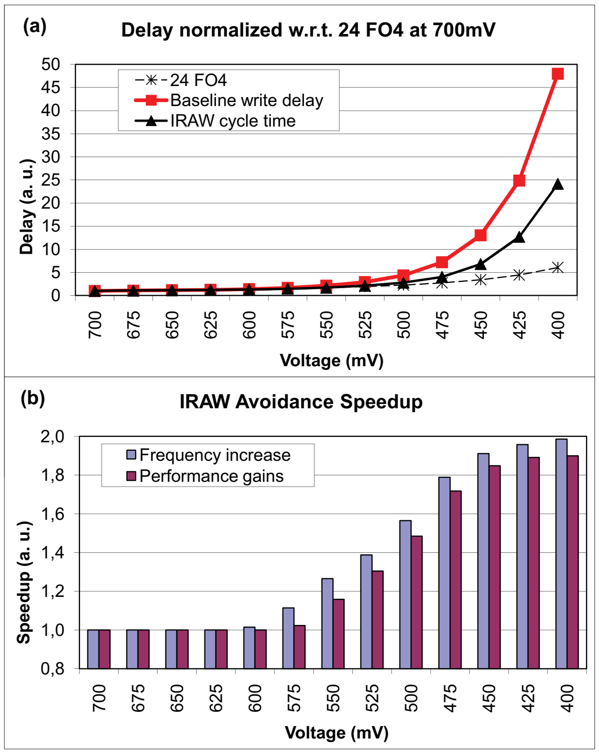

Figure 11. (a) Normalized Cycle time and (b) frequency and performance gains w.r.t. the baseline

In our experiments IRAW avoidance must be deactivated above 600mV and one stabilization cycle suffices below 600mV. As explained before, IRAW avoidance works for different number of IRAW cycles and can be reconfigured dynamically. Thus, our mechanism would work also for different technology nodes or *Vcc* ranges where the number of IRAW cycles was larger.

Performance increase is slightly lower than frequency raise because (i) off-chip memory latency remains constant and (ii) our IRAW avoidance mechanism introduces some performance degradation due to those stalls caused in the different structures. Performance degradation due to stalls ranges between 8% and 10% at different *Vcc* levels. Such performance degradation is mostly caused by issue stalls required by the IRAW avoidance mechanism in the register file. For instance, performance drop at 575mV is 8.86% and distributes as follows: 8.52% due to issue stalls re-

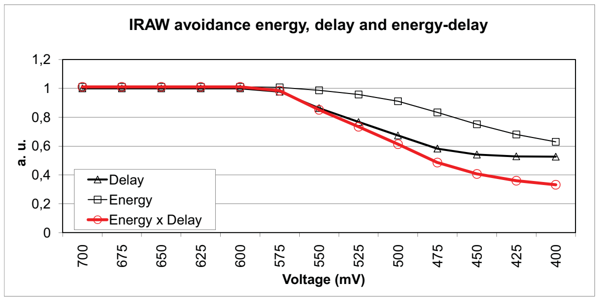

Figure 12. Energy, delay and EDP

quired to avoid IRAW in the register file, 0.30% due to DL0 IRAW avoidance, and the remaining 0.04% due to IRAW avoidance in the remaining blocks. Due to this overhead IRAW avoidance is not used at 600mV because operating frequency could be raised by a modest 1%, which would be largely offset by the stalls caused to avoid IRAW violations. Note that the compiler could help removing some of the register file induced stalls by scheduling instructions properly. However, such compiler optimizations are out of the scope of this paper.

Register file delays the issuance of instructions relatively often. Those instructions are delayed because they would read not-yet stabilized values otherwise. This is so because issuing instructions in order implies that most of them are critical in the overall execution time, and thus, even if stalls are not very frequent, any delay in the issue stage is very likely to impact performance. As stated before, 13.2% of the instructions are delayed by one cycle to avoid IRAW violations, and such stalls cause a performance degradation between 8% and 10%. Our implementation of IRAW avoidance for the DL0 degrades performance to some extent mainly due to those stalls required after a cache line fill.

5.3 Power and Area Impact

Energy and area overheads for the extra hardware required are neglibible (below 1% extra energy and 0.03% extra area) despite our pessimistic assumptions.

Results in terms of energy, delay and energy-delay product for the IRAW avoidance scheme and the realistic baseline are compared in Figure 12. All parameters are compared at each *Vcc* level. IRAW is slightly worse at high *Vcc* (between 700mV and 575mV) because its energy consumption is almost 1% higher than that of the baseline. As stated before such overhead is probably too pessimistic. Delay remains the same in the 700mV - 575mV voltage range. However, as *Vcc* decreases, energy and delay for the IRAW avoidance scheme grow much more slowly than those of the baseline. Thus, IRAW avoidance provides lower delay, energy and energy-delay product (EDP for short) than the baseline. For instance, the relative EDP of IRAW avoidance with respect to the baseline is as low as 0.61 at 500mV and 0.41 at 450mV. Note that energy gains of IRAW avoidance with respect to the baseline come from the fact that execution time, and thus leakage, are lower for our IRAW avoidance approach. Leakage per time unit grows around 10% per 25mV decrease whereas dynamic energy depends quadratically on *Vcc*, which is consistent with the *Vcc* levels considered [8]. Thus, at lower *Vcc* leakage has higher contribution to the total energy. For instance, if we assume 5J energy consumption at 450mV and the cycle time was not constrained by write delay, 1.24J would correspond to leakage. However, write delay decreases operating frequency in the baseline case, and therefore, it would spend 8.50J (4.74J due to leakage). Conversely, IRAW avoidance increases performance and hence, shortens execution time leading to lower energy: 6.40J (2.64J due to leakage) to perform the same task.

Recalling Table 1 we can conclude that IRAW avoidance enables higher operating frequency, works for all SRAM blocks of an Intel® Silverthorne^{TM} core, adapts to multiple *Vcc* levels, its hardware overhead is small, does not introduce new testing issues and IPC degradation is low.

6 Related Work

Reducing the latency of the SRAM arrays has been a concern in the last years. This matter is especially important at low *Vcc* operation, where SRAM write delay grows exponentially. Several approaches have addressed this issue for register files from the microarchitecture standpoint by reducing the size and number of ports [14, 21, 25], which is also achieved indirectly with clustered microarchitectures [5, 28]. Multiple-banked designs also reduce read and write delay in register files [2, 7, 21, 25, 28] and cache memories [11], as well as multi-level register files [2,4] and cache memories [15]. However, even if those techniques are in place, *Vcc* scales only to some extent for register files and caches, and mechanisms as the IRAW avoidance proposed in this paper are needed for the SRAM blocks.

Techniques for low *Vcc* operation in SRAM structures have been proposed in the past. The drowsy cache allows retaining contents at a very low *Vcc* [12, 13, 27]. Whenever some data must be accessed the *Vcc* is raised in the proper bank. This technique saves power, but does not reduce SRAM write delay.

Our approach is orthogonal to all techniques above and can be easily combined with any of them. Moreover, our approach is not constrained to few blocks. Instead, IRAW avoidance can be applied to all SRAM blocks of an in-order core.

7 Conclusions

Lower *Vcc* is required due to energy constraints in the mobile market segment. However, decreasing *Vcc* increases SRAM write delay dramatically. Hence, operating frequency is severely affected by *Vcc* scaling. Existing solutions either have high overhead or cannot be used for all SRAM blocks.

In this paper we present the first approach to tolerate high SRAM write delay at low *Vcc* in an Intel® Silverthorne™ in-order core by interrupting write operations, stabilizing contents across several cycles and avoiding any read operation of those not-yet stabilized values.

IRAW avoidance increases operating frequency by 57% at 500mV and 99% at 400mV with negligible area and power overhead. Such frequency boost translates into 48% speedup at 500mV and 90% at 400mV, and low EDP (0.61 EDP at 500mV and 0.33 at 400mV).

In summary, our IRAW avoidance approach enables high operating frequencies when scaling down *Vcc* for in-order cores at very low cost and with high flexibility.

8 Acknowledgements

We would like to thank Steven Hsu, Amit Agarwal, Ram Krishnamurthy and Tiju Jacob for their feedback on the circuit level impact of *Vcc* scaling and their support for electric simulations. This work has been partially supported by the Spanish Ministry of Education and Science under grant TIN2007-61763 and the Generalitat de Catalunya under grant 2009SGR1250.

References

[1] J. Abella, J. Carretero, P. Chaparro, X. Vera, and A. Gonzalez. Low vccmin fault-tolerant cache with highly predictable performance. In *MICRO*, 2009.

[2] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *MICRO*, 2001.

[3] M. Brown and Y. Patt. Using internal redundant representations and limited bypass to support pipelined adders and register files. In *HPCA*, 2002.

[4] J.-L. Cruz, A. González, M. Valero, and N. Topham. Multiple-banked register file architectures. In *ISCA*, 2000.

[5] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The multicluster architecture: reducing cycle time through partitioning. In *MICRO*, 1997.

[6] G. Gerosa et al. A sub 1W to 2W low power IA processor for mobile internet devices and ultra mobile PCs in 45nm Hi-K metal gate CMOS. In *ISSCC*, 2008.

[7] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14):1–6, 1996.

[8] S. Hanson, B. Zhai, K. Bernstein, D. Blaauw, A. Bryant, L. Chang, K. Das, W. Haensch, E. Nowak, and D. Sylvester. Ultralow-voltage, minimum-energy CMOS. *IBM Journal of Research and Development*, 50(4/5):469–490, 2006.

[9] M. Ishida et al. A novel 6T-SRAM cell technology designed with rectangular patterns scalable beyond 0.18 μm generation and desirable for ultra high speed operation. In *IEDM*, 1998.

[10] A. Iyer and D. Marculescu. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *ICCAD*, 2002.

[11] T. Juan, J. Navarro, and O. Temam. Data caches for superscalar processors. In *ICS*, 1997.

[12] N. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy instruction caches: leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *MICRO*, 2002.

[13] N. Kim, K. Flautner, D. Blaauw, and T. Mudge. Single-V_{DD} and single-V_T super-drowsy techniques for low-leakage high-performance instruction caches. In *ISLPED*, 2004.

[14] N. Kim and T. Mudge. The microarchitecture of a low power register file. In *ISLPED*, 2003.

[15] J. Kin, M. Gupta, and W. Mangione-Smith. The filter cache: an energy efficient memory structure. In *MICRO*, 1997.

[16] A. KleinOsowski et al. Latch design techniques for mitigating single event upsets in 65 nm SOI device technology. *IEEE Transactions on Nuclear Science*, 54(6):2021–2027, Dec. 2007.

[17] J. Kulkarni, K. Kim, and K. Roy. A 160 mv, fully differential, robust schmitt trigger based sub-threshold SRAM. In *ISLPED*, 2007.

[18] Z. Liu and V. Kursun. High read stability and low leakage cache memory cell. In *ISCAS*, 2007.

[19] Y. Morita et al. An area-conscious low-voltage-oriented 8T-SRAM design under DVS environment. In *VLSI*, 2007.

[20] S. Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, University of Wisconsin - Madison, 1998.

[21] I. Park, M. Powell, and T. Vijaykumar. Reducing register ports for higher speed and lower energy. In *MICRO*, 2002.

[22] D. Roberts, N. Kim, and T. Mudge. On-chip cache device scaling limits and effective fault repair techniques in future nanoscale technology. In *DSD*, 2007.

[23] M. Saint-Laurent, B. Mohammad, and P. Bassett. A 65-nm pulsed latch with a single clocked transistor. In *ISLPED*, 2007.

[24] G. Semeraro et al. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *MICRO*, 2002.

[25] J. Tseng and K. Asanović. Banked multiported register files for high-frequency superscalar microprocessors. In *ISCA*, 2003.

[26] C. Wilkerson et al. Trading off cache capacity for reliability to enable low voltage operation. In *ISCA*, 2008.

[27] K. Zhang et al. A 3-GHz 70MB SRAM in 65nm CMOS technology with integrated column-based dynamic power supply. In *ISSCC*, 2005.

[28] V. Zyuban and P. Kogge. The energy complexity of register files. In *ISLPED*, 1998.