

# A Physical-Level Study of the Compacted Matrix Instruction Scheduler for Dynamically Scheduled Superscalar Processors

Elham Safi, Andreas Moshovos, and Andreas Veneris

Electrical and Computer Engineering Department  
University of Toronto  
Toronto, Canada

{elham, moshovos, veneris} @eecg.toronto.edu

**Abstract**—This work studies hardware complexity (physical level characteristics) of the recently proposed compacted matrix instruction scheduler for dynamically scheduled, superscalar processors. Previous work focused on the matrix scheduler’s architecture and argued in support of its speed and scalability advantages; however, neither actual physical-level investigations nor models were reported for it. Using full-custom layouts in a commercial 90 nm fabrication technology, this work investigates the latency and energy variations of the compacted matrix and its accompanying logic as a function of the issue width, the window size and the number of global checkpoints. This work also proposes an energy optimization that throttles unnecessary pre-charges and evaluations. This optimization reduces energy by 10% and 18% depending on the scheduler size.

**Keywords:** *Compacted matrix schedulers, Physical level implementation, Latency, Energy.*

## 1. INTRODUCTION

Dynamically-scheduled processors exploit instruction level parallelism by buffering instructions and scheduling them potentially out of the program order. The *instruction scheduler*, responsible for the buffering and scheduling, typically comprises wakeup and select sides. Instructions wait in the wakeup stage to become *ready*, i.e., until all their input operands become available. The select side chooses a set of ready instruction to be sent for execution taking resource constraints into consideration. The scheduling loop formed between the wakeup and the select sides is critical for performance [6]. This scheduling loop’s delay is a function of the scheduler size (often synonymous to the window size) and to a lesser extent of the issue width.

Scheduler is a performance-critical component of processors. Large schedulers by issuing more instructions every cycle can improve the instruction per cycle (IPC) completion rate. However, commercial processors do not use large schedulers because they are slow, hence negating the performance benefits of higher IPC (e.g., the Intel and AMD desktop/server processors have integer scheduler sizes of 24 to 32 entries). Larger and hence slower schedulers decrease clock period, which offset any IPC benefits gained through using them, and hence lead to lower overall performance.

The instruction scheduler’s wakeup side serves three roles: i) it holds instructions waiting for their inputs to be produced, ii) it matches waiting instructions with incoming results, and iii) it identifies ready-for-execution instructions (all input operands are available). The matching functionality of the wakeup can be implemented using content addressable memories (CAMs) [1] [6] or dependency matrices [1] [3]. The latter is faster and consumes less power than the former. The *compacted matrix scheduler (CMS)* exploits typical program behavior to reduce the matrix width, and hence the wakeup latency [5]. CMS uses an indirection table, called wakeup allocation table or *WAT*, to manage the subscription of columns serving as communication channels where producer instructions wake up their consumer instructions.

Although arguments have been made in support of CMS’s speed and scalability advantages [5], neither actual physical level measurements nor models have been reported for it. Models can be used by computer architects to study the latency and energy of various alternatives during early stages of architectural level exploration where physical-level implementation is either impossible to develop, or unaffordable due to time and/or cost constraints. Accordingly, this work investigates the delay and energy variations of the CMS and WAT as a function of the issue width (*IW*), the window size (*WS*), and the number of speculation-recovery global checkpoints (*NoGCs*).

For this investigation, full-custom implementations were developed in a commercial 90 nm fabrication technology. To the best of our knowledge, this is the first work to study the physical-level implementation and optimizations of CMS and checkpointed WAT. A physical-level study is further useful in exposing design inefficiencies or optimization opportunities. Accordingly, this work proposes an energy optimization that throttles pre-charge and evaluation for unallocated matrix columns. This optimization reduces energy depending on the scheduler size (e.g., for 32×20 and 64×20 matrices, this optimization reduces energy by 10% and 18% respectively).

The rest of this paper is organized as follows: Section 2 reviews CMS and WAT. Section 3 and 4 discusses physical-level implementation and evaluation results respectively. Finally, Section 5 summarizes key findings.

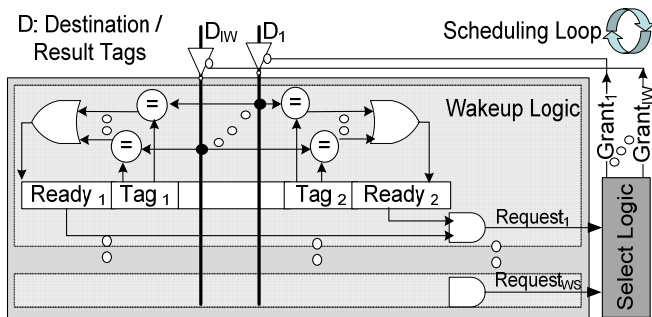


Figure 1: CAM-based wakeup

## 2. BACKGROUND

In typical modern processors, instructions first pass through register renaming, and then enter the instruction scheduler where they wait until they can proceed and execute. The instruction scheduler comprises wakeup and select sides. The wakeup side keeps track of the data dependencies among  $WS$  instructions (without loss of generality, we assume that the window size and the scheduler size are the same). An instruction remains dormant while some of its input/source operands are still outstanding. Once all of the source operands of an instruction become ready, the instruction is “woken up” and becomes ready to send execution request to the select logic. The wakeup side produces a vector of size  $WS$  showing the ready status of all of its instructions. The select side chooses at most  $IW$  instructions of those marked as ready in the request vector taking into consideration resource constraints (e.g., availability of functional units). The select side uses a priority encoder that typically chooses the oldest ready instruction to be sent for execution.

The rest of this section reviews three scheduler architectures: the CAM-based scheduler, the regular (uncompressed) matrix-based scheduler and the compacted (compressed) matrix-based scheduler.

### 2.1 CAM-Based Scheduler

Depicted in Figure 1, CAM-based scheduler (e.g., implemented in MIPS R10000) has  $WS$  entries. Each entry includes a tag field and a ready bit per source operand [6]. As up to  $IW$  instructions are selected for execution, their result tags (e.g., indexes of the physical registers or reservation stations) are broadcast on the result buses. Broadcasting of a result tag is delayed for an appropriate number of clock cycles, defined by the instruction’s execution latency. Each result bus is connected to comparators, one per source operand tag at each entry. The comparators match the waiting source operand

tags against the result tags. For  $X$  instructions, each having  $Y$  source operands, and with  $Z$  result broadcast buses,  $X \times Y \times Z \times \log_2(WS)$  single-bit comparators are required; assuming that window size ( $WS$ ) is equal to the number of physical registers. Once all source operands for an instruction become ready, it sends an execution request to the select logic.

### 2.2 Matrix-Based Schedulers

The conventional  $WS \times WS$  matrix scheduler has a row and a column per instruction [1] [3]; instruction’s column index and row index are the same as its scheduler entry index. A row records the instruction’s input dependencies whereas a column marks instruction’s true output dependencies. The matrix operates as follows. A matrix row  $c$  is allocated to an instruction when it enters into the scheduler. For each source operand that is still pending, if the instruction at the scheduler entry  $c$  consumes the result produced by the instruction at scheduler entry  $p$ , matrix cell  $(c,p)$  is set to one. Just before an instruction produces its result, it clears all cells along its associated column. In this way, the producer notifies all its consumers of its result availability. When a row is all zero, the corresponding instruction turns on its row-ready bit in order to send an execution request to the select logic.

Figure 2(a) to (c) illustrate subsequent snapshots of the wakeup matrix for an eight-entry instruction scheduler ( $Q$ ). In addition to a matrix row and a matrix column, a  $Q$  entry is associated with an instruction; each  $Q$  entry maintains a destination tag, source tags and a valid bit. These fields are used during executing instruction and clearing instruction’s associated column. The scheduling decisions are performed using the matrix cells alone. In this discussion,  $Q_x$  refers to the instruction stored at scheduler entry  $x$ . At time  $T=0$ ,  $Q$  has five valid entries. The cell  $(1,7)$  is set since  $Q_1$  depends on  $Q_7$ .  $Q_7$  is ready since its row is all clear. In superscalar processors, up to  $IW$  instructions can be issued (i.e., sent for execution) per cycle. Also, up to  $IW$  instructions can be renamed and allocated into  $Q$ . At time  $T=1$ ,  $Q_7$  issues and accordingly clears its column 7, resulting in  $Q_1$  and  $Q_4$  becoming ready. Also, at time  $T=1$ , two new instructions are simultaneously allocated into  $Q_0$  and  $Q_6$  respectively. When new instructions enter into  $Q$ , their dependencies are recorded in the matrix.  $Q_0$  uses  $R_{11}$ , produced by  $Q_1$ , hence cell  $(0,1)$  is set.  $Q_0$  also uses  $R_2$  that is ready. For  $Q_6$ , the cells  $(6,1)$  and  $(6,4)$  are set since  $Q_6$  uses  $R_{14}$  and  $R_{12}$  respectively. At time  $T=2$ ,  $Q_1$  and  $Q_4$  become ready, hence their columns and scheduler entries are cleared. As a result,  $Q_0$  becomes ready. However,  $Q_5$  and  $Q_6$  are still waiting for  $Q_2$  producing  $R_{14}$ .

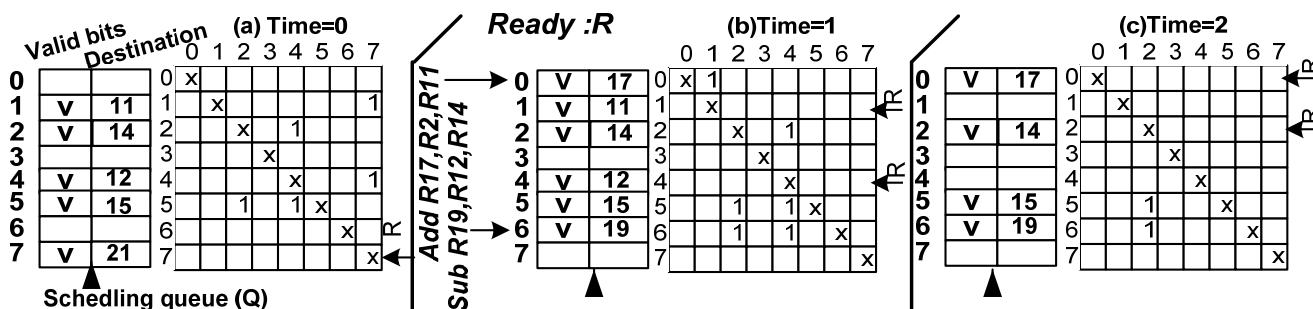


Figure 2: Scheduling with the dependency matrix.

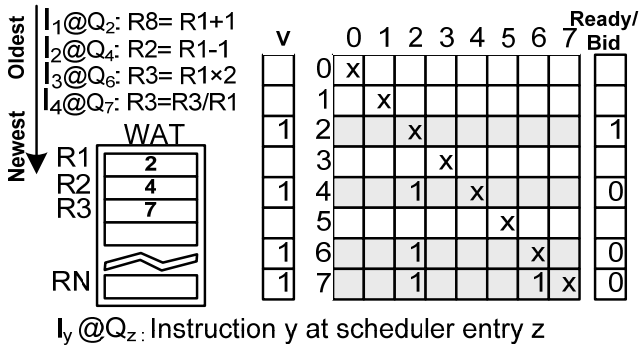
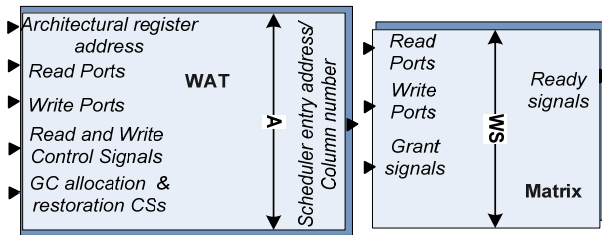


Figure 3: Instruction sequence example

### 2.3 Compacted Matrix Scheduler (CMS)

The size of the conventional matrix is proportional to  $WS^2$  making larger schedulers difficult to implement. However, in practice similar IPC performance can be achieved using a compacted matrix with fewer columns [5]. Several observations motivated compacted matrix schedulers: (i) irrespective of the scheduler size, about 70% of the instructions either have no consumer(s) in the scheduler or do not need to broadcast any result (e.g., branch or store); (ii) large schedulers are rarely full of producers and are often still refilling from pipeline flushes; and (iii) matrix snapshots during execution show that a few active dependencies exist per cycle, hence most of the WS matrix columns are unused at any given time and the matrix width can be reduced. Previous work demonstrates that sufficient number of matrix columns are respectively 12-16 for typical (e.g., up to 128) and 20 for larger scheduler sizes [5].

CMS takes advantage of the aforementioned observations and postpones allocating a column to an instruction until its first consumer enters the scheduler. Reducing the matrix width



Issue width:  $IW$   
 Instruction scheduler/ Window size:  $WS$   
 Number of checkpoints:  $NoGC$   
 Number of source registers per instruction:  $S$   
 Number of destination registers per instruction:  $D$   
 Number of architectural registers:  $A$   
 Number of physical registers:  $P = WS$   
 Number of columns:  $CN = 20$  ( $CN \leq \log_2(WS)$ )

Number of WAT entries:  $A$   
 Number of WAT read ports:  $(S+D) \times IW$   
 Number of WAT write ports:  $D \times IW$   
 WAT entry width:  $\log_2(WS) + 2 + NoGC$

Number of matrix entries:  $WS$   
 Number of matrix read ports:  $IW$   
 Number of matrix write ports:  $IW$   
 Matrix entry width:  $CN$

Figure 4: geometries and block diagrams for the CMS and the WAT

improves the speed of larger schedulers at the cost of a negligible IPC performance loss [5]. CMS uses a redirection table, called wakeup allocation table or WAT, to assign columns to producer instructions. WAT is required because in compacted matrix unlike the conventional matrix, the instruction's column index is not the same as its scheduler entry index. The WAT manages column subscriptions by finding producers' column indices for consumer's source operands. To assign a column to a producer, WAT assigns the column to the destination register of the instruction. A column free list (CFL) is also required to keep a list of free columns.

#### 2.3.1 Wakeup Allocation Table

Shown in Figure 4, the WAT has one entry per architectural register. Each WAT entry can be in one of the following four states: *unallocated* (the register value is unavailable and the producer instruction has not been assigned a column yet), *allocated* (a column is assigned to the producer, but producer's result, register value, is unavailable), *ready* (the register value is available), and *de-allocated* (the column was released after the producer and/or all of its consumers left the scheduler). The WAT entry width is  $\log_2(WS)+2$  as it stores either a row (or scheduler entry) index or a column index and two bits to encode one of the four WAT states. The WAT operations are WAT lookup (read), WAT update (write), GC allocation and GC restoration.

The rest of this section describes WAT operations and WAT outputs by means of an example. Using the instruction sequence shown in Figure 3, the rest of this section discusses the following scenarios: a producer entering the scheduler (*CASE 1*), the first consumer enters the scheduler (*CASE 2*), a subsequent consumer enters the scheduler (*CASE 3*), the producer executes (*CASE 4*), and the producer's associated column is released (*CASE 5*).

##### CASE 1 – Producer enters the scheduler (WAT Update):

When a new instruction enters the scheduler, the WAT entry corresponding to the destination architectural register is updated with the instruction's row index (or scheduler entry index); however, no column is allocated for the instruction. For example, when  $I_1$  with destination  $R_1$  enters  $Q_2$ , the WAT entry associated with  $R_1$  is set to "unallocated, 2", meaning that  $R_1$ 's value will be produced by  $I_1$  at  $Q_2$ .

In superscalar processors, up to  $IW$  scheduler entries may be concurrently allocated. Hence, up to  $IW$  WAT updates may proceed simultaneously. However, when write-after-write (WAW) dependencies exist, only the last update should proceed into the WAT. Hence, WAW dependencies must be detected among the  $IW$  co-renamed instructions; the destination of each instruction is compared with those of its preceding instructions. The WAT update is restricted to the last writer for a specific destination architectural register ( $I_4$  at  $Q_7$  updates the WAT entry for the destination  $R_3$ ).

##### CASE 2 – First consumer enters the scheduler (a WAT Lookup followed by a WAT Update):

Assume that  $I_2$  with source  $R_1$  is allocated into  $Q_4$ . The WAT lookup for source  $R_1$  returns "unallocated, 2", indicating that  $R_1$  is produced by  $I_1$  at  $Q_2$ . Since  $I_2$  is the first consumer of  $I_1$ , a free column (e.g., 3)

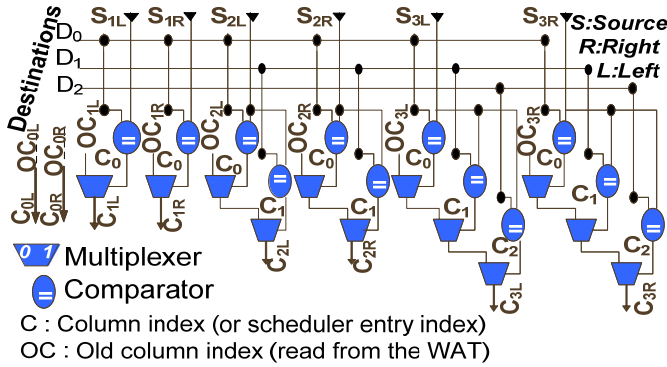


Figure 5: RAW dependency checking

is allocated for  $I_1$  by the CFL. The WAT entry for R1 is set to “allocated, 3”, meaning that as of now column 3 belongs to  $I_1$ , the producer of the architectural register R1.  $I_1$  must also be informed of its newly-assigned column index so that it can clear it when  $I_1$ ’s result becomes ready. An extra field per scheduler entry keeps the column index if any column is allocated. Finally, the matrix cell (4,3) is set to one to record the read-after-write (RAW) dependency of  $I_2$  on  $I_1$ .

**CASE 3 – Subsequent consumer enters the scheduler (WAT Lookup):**  $I_3$  at  $Q_6$  is a subsequent consumer of R1. The WAT lookup for source R1 returns “allocated, 3”, and the matrix cell (6,3) is set to record the RAW dependency of  $I_3$  on  $I_1$ . This and the preceding scenarios show that a WAT lookup for a source architectural register returns either the producer’s row index (CASE 2) or column index (CASE 3).

In superscalar processors, if RAW dependencies exist between a pair of IW co-renamed instructions, the producer’s columns index (e.g.,  $I_1$ ) may be assigned at the same cycle a consumer needs it (e.g.,  $I_2$  or  $I_3$ ). However, the WAT will not be updated by the end of the cycle. Hence, RAW dependencies among co-renamed instructions must be detected as depicted in Figure 5; the source of each instruction is compared against the destinations of all its co-renamed preceding instructions. The circuitry for detecting RAW and WAW dependencies can be shared between the renaming’ register alias table (RAT) and the WAT, since renaming also requires this information.

**CASE 4 – Producer completes execution (WAT Update):** When producer  $I_1$  starts execution, after an appropriate number of cycles, depending on  $I_1$ ’s execution latency, its column (column 3 has been assigned to  $I_1$ ) is cleared and matrix-based wakeup proceeds (Section 2.3.3). At this point, the R1’s WAT state changes to *ready*. If the producer clears its column at the same time a consumer enters the scheduler, the column reset is prioritized over the dependency-induced matrix cell set. Otherwise, deadlock can occur as the consumer would await a never-happening column reset.

**CASE 5 – Column is released (WAT Update):** A column is freed if (i) its producer and/or (ii) all its consumers have left the scheduler. The latter is an optimization for when a pipeline flush occurs between a register’s producer and consumer(s).

Even if the producer stays in the scheduler, the register state can be changed to *de-allocated* and its column can be freed. If the processor supports speculative scheduling (i.e., consumers of a load are granted execution predicting a hit in the L1 data cache), a column is not freed once the producer writes back its result. If the load misses, the consumers will be reset in the scheduler and wait for a second column broadcast from the load upon hit. Thus, an instruction can broadcast on the same column multiple times; in this case, only after the producer leaves the scheduler, its associated column will be freed.

### 2.3.2 Recovery from Mispeculation

The WAT like the RAT is a speculative structure and requires support to recover from mispeculations. For WAT, recovery can be done using global checkpoints (GCs); GCs are copies of the WAT content taken when a rollback due to mispeculation is possible (e.g., on predicted branches).

Checkpointing for the matrix is not required as long as the scheduler entries’ valid bits for squashed instructions are cleared. Some rows may keep obsolete information when pipeline flushes occur. However, obsolete information does not harm if the select logic is prevented from receiving false requests. To do so, for each entry, the valid bit is ANDed with the ready bit to produce the request to the select logic.

### 2.3.3 Dependency Matrix Operations

Shown in Figure 4, the compacted matrix has WS rows and 20 columns (Section 2.3). The matrix operates as follows: **A matrix row is updated** for a new instruction  $r$  by finding the column numbers for the producers of the instruction’s source operands from the WAT (e.g.,  $c_1$  and  $c_2$ ) and setting the matrix cells (e.g.,  $(r,c_1)$  and  $(r,c_2)$ ). These steps proceed in parallel with rename and dispatch (writing the instruction into the scheduler). Before an instruction completes execution, its matrix **column is cleared**. When the instruction’s matrix row cells are all zero, its **row-ready bit is set**, interpreted as a request by the select logic. Figure 6 summarizes the actions that take place in the CMS and WAT.

## 3. PHYSICAL-LEVEL IMPLEMENTATION

This section describes the physical level implementation of the WAT and CMS.

### 3.1 WAT

A non-checkpointed WAT is a multi-ported register file. Assuming a MIPS-like instruction set architecture, where the instructions may have at most two source operands and one destination operand, the WAT needs to support  $3 \times N$  reads and  $N$  writes per cycle, where  $N$  is the number of instructions required to be renamed per cycle.  $2 \times N$  read ports are used to rename the two source operands, and another  $N$  read ports are needed to read the current mappings of the destinations for the purpose of recovery using the reorder buffer [7].  $N$  write ports are also used to write new mappings for the destinations.

Renaming/Dispatch	Scheduling	Execution
WAT lookup for sources/ WAT lookup for destinations/ Matrix update	Matrix read Producing row-ready bits	Matrix column clearance/ WAT update for destination

Figure 6: Actions included in various pipeline stages for WAT and CMS



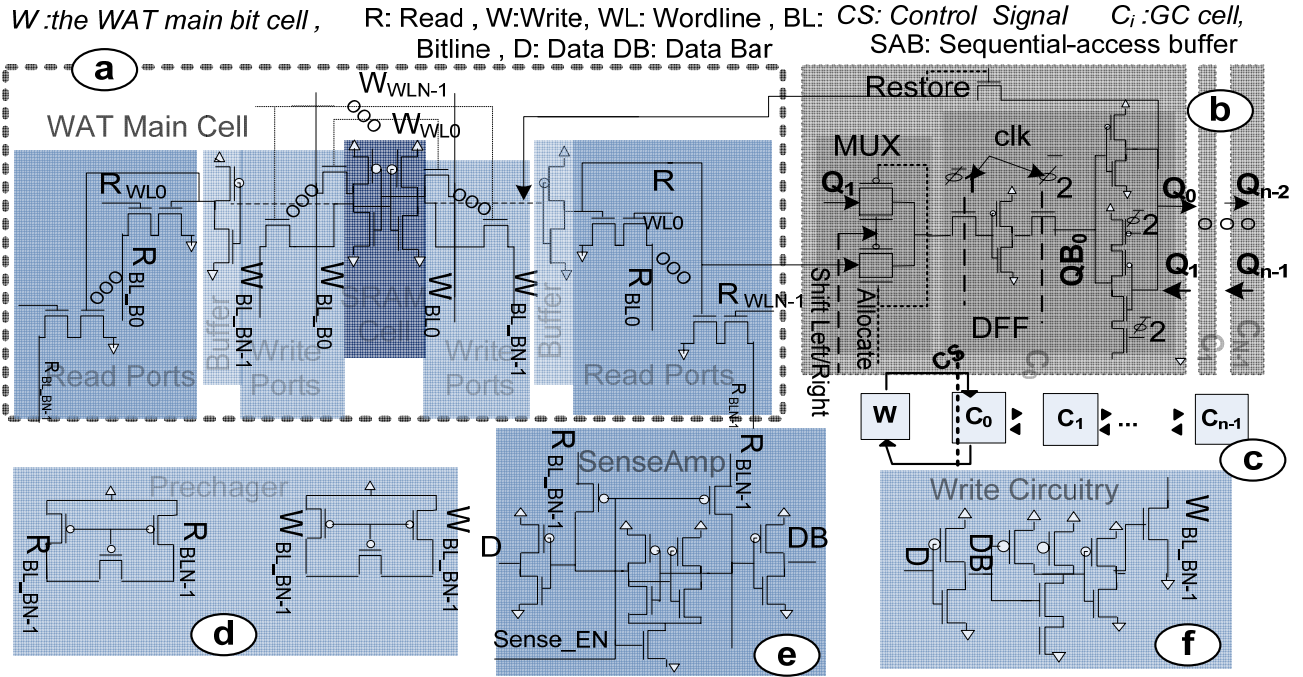


Figure 7: WAT building blocks (a) Multi-ported SRAM cell, (b) SAB GC cell, (c) SAB GC organization mechanism, (d) pre-chargers, (e) sense amplifiers and (f) write drivers

Depicted in Figure 7, a checkpointed WAT embeds GCs, copies of the WAT cells. Figure 7(a), shows the main WAT cell with multiple read and write ports. The GC cells, depicted in Figure 7 (b), are organized in a bi-directional shift register, shown in Figure 7 (c). Figure 7 (d), (e) and (f) respectively illustrate the precharge circuitry, the sense amplifier logic used during reads, and the write drivers used during writes.

This work uses the serial-access-buffer (SAB) GC organization mechanism as it is faster and simpler than the alternative [7]. SAB organizes the GCs in a bi-directional shift register with connections between adjacent cells; only one of the GCs is connected to the main WAT cell through pass gates [1] [7]. GC allocation is done by shifting the GC cells to the right, copying the WAT data cell to the adjacent vacant position. Restoring from a GC may require multiple steps since the appropriate GC must be shifted into the WAT main cell. A SAB GC cell consists of a register and a multiplexer controlling the shift direction. SAB uses two non-overlapping clocks and two external control signals irrespective of the number of GCs. Multiple reads may access the same WAT entry; hence, the main WAT cell must be capable of driving a capacitance proportional to the number of WAT ports and GC(s). To protect the cell's data during multiple accesses, decoupling buffers isolate the WAT data cell and the read ports [9] [1]. Differential read and write operations are used due to their superior latency, energy and noise margins. To reduce power, the following techniques were employed: (i) pulse operations for the wordlines, periphery circuits and sense amplifiers, (ii) multi-stage static CMOS decoding, and (iii) current-mode read and write operations.

### 3.2 Compacted Matrix

Figure 8(a) shows the architecture of the compacted matrix scheduler and its interface with the select logic. Figure 8(b) also illustrates the matrix column's transistor-level implementation. The matrix receives its inputs, the grant (or upcoming execution completion) signals, from the select logic, and produces the row-ready outputs (the data inputs are active-low). When a new instruction is allocated into a scheduler entry, its associated matrix row is updated with a bit pattern where the dependencies for instruction sources are set by the data decoders. Up to  $IW$  instructions may enter the scheduler concurrently; hence, up to  $IW$  matrix rows may be updated simultaneously. Therefore, the matrix includes  $IW$  data decoders that provide  $IW$  sets of bit patterns, each with as many bits as the number of matrix columns (20 in our implementation as per discussion of Section 2.3).

Each matrix cell consists of an SRAM cell. The write ports for update and read ports for producing row-ready outputs are single-ended. Up to  $IW$  instructions may write into  $IW$  different matrix rows simultaneously. Hence, each matrix cell needs  $IW$  write ports. Figure 8(b) shows the transistor-level implementation of a column. The grant signals entering from the right side become vertical and are sent over the corresponding columns. The vertical broadcast signals clear the corresponding columns. A bitwise-NOR of all matrix cells along a row generates the row's row-ready output (single-ended read ports implement the NOR gate). Up to  $IW$  different columns may be cleared simultaneously. The row-ready signals are first pre-charged to high. A single set matrix cell per row (in this implementation a set cell keeps the value zero)

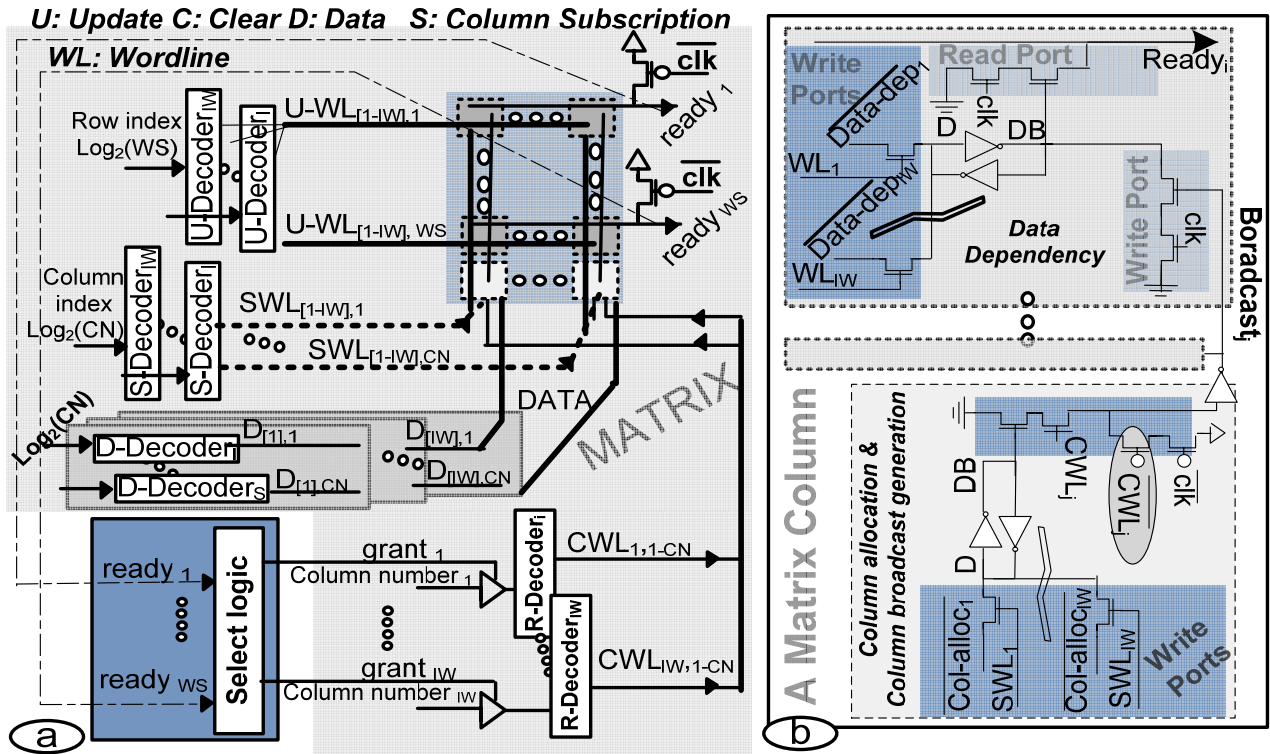


Figure 8: (a) Matrix overall organization and (b) Matrix column's transistor level implementation

can pull-down the row-ready to zero. If all row cells are *unset* (i.e., all cells are set to one), the row-ready stays high, interpreted as a request for execution by the select logic. Producing all row-ready output signals proceeds in parallel.

For column subscription, an additional SRAM cell per column is set (to zero) to indicate that the column is currently allocated. Up to  $IW$  columns may be allocated per cycle during renaming; hence, each *column-subscription* cell, which exist per matrix column, needs  $IW$  read/write ports. These ports are also used for column de-allocation (set column-subscription cell to one). Up to  $IW$  columns may need to be cleared during result write-back. For each column, the column-subscription cell data is ANDed with the input grant to determine if broadcast on the column is permitted. In compacted matrix, compared to a conventional matrix, the row-ready critical path has one extra gate delay penalty due to the column-subscription technique; however, the compacted matrix's reduced latency is supposed to offset this penalty.

If the grant signal for a column is active, the normal precharge and evaluation operations proceed; otherwise, they are unnecessary. To prevent unnecessary activities, and hence to reduce the energy consumption, an extra switch (pass transistor) per column, shown circled in Figure 8(b), is added. For  $32 \times 20$  and  $64 \times 20$  matrices, this optimization reduces total energy by 10% and 18% respectively.

#### 4. EVALUATION

This section presents the latency and energy measurements from the physical implementation for CMS and WAT. Section 4.1 details the implementation and measurement

methodologies. Sections 4.2 and 4.3 present the latency and energy measurements as well as analysis for the CMS and WAT respectively. Finally, Section 4.4 presents empirical models for quick estimation of delay and energy.

##### 4.1 Design and Measurement Methodology

We restrict our attention to WATs with 0, 4, 8 or 16 GCs because previous work demonstrated that for the SRAM-based RATs, similar in structure to the SRAM-based WATs considered in this work, 16 GCs or less are sufficient to achieve performance close to what is possible with infinite GCs [4]. The number of architectural registers is either 32 or 64 corresponding to typical processors without or with floating-point register sets. The number of physical registers varies from 32 to 256 in power of two steps. We developed full-custom layouts for all components using the Cadence tool set in a 90 nm commercial fabrication technology with a 1.2V supply voltage. For circuit simulations, Spectre<sup>TM</sup> was used, and worst-case latency and energy values are reported.

Circuit designs can be tailored to achieve different latency and energy tradeoffs. In an actual commercial design, a target latency and/or energy is decided and used as a specification for tuning components. In lieu of an actual specification for the target operating frequency, we used CACTI 4.2 [8], a tool providing latency, power and area estimations for caches and SRAMs, to obtain upper bounds on the latencies of SRAMs that would be similar in size to the WAT [8]. Using CACTI, for the base 4-way superscalar WAT, we determined an upper bound on the critical path delay by estimating the delay of a 64-bit, 64-entry SRAM with 12 read and four write ports. This

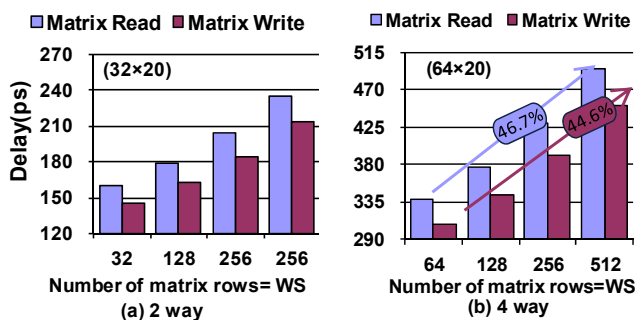


Figure 9: Matrix read and write latency for (a) 2-way and (b) 4-way matrix schedulers.

upper bound is reasonable since non-checkpointed WATs/RATs are identical to multi-ported SRAMs.

#### 4.2 Compacted Matrix

Figure 9 (a) and (b) report compacted matrix read and write latencies for 2-way and 4-way superscalar schedulers respectively. Figure 10 shows the corresponding energy measurements. The number of matrix rows (equal to WS) varies from 32 to 512. Each matrix has 20 columns (Section 2.3) since using column subscription technique 20 columns are sufficient for large scheduler sizes.

The matrix read delay (wakeup delay) is the time period between the select logic's grant activation and row-ready discharge. Increasing the number of matrix rows increases matrix read latency as it increases the size, and hence the latencies of the decoder and select logic's priority encoder. For example, when WS increases by 8x from 64 to 512, read latency increases by 46.7%. The scheduler delay is the sum of the matrix read delay (wakeup delay) and the select logic delay. The scheduler delay increases logarithmically by increasing the number of matrix rows (equal to WS) when IW is fixed. The matrix write delay is the time period between the dependency-data preparation and the matrix cell's value change. Matrix read latency is longer than write latency. For a fixed IW, matrix write delay and energy increase logarithmically for larger entry counts (equal to WS).

Increasing the number of matrix columns (matrix width) increases row-ready NOR gate's size, and hence the wakeup delay. Increasing the matrix width also increases the load on the row clock driver, and hence the driver's delay. By transistor resizing and output buffering, significant delay increase can be avoided at the cost of more energy. While we

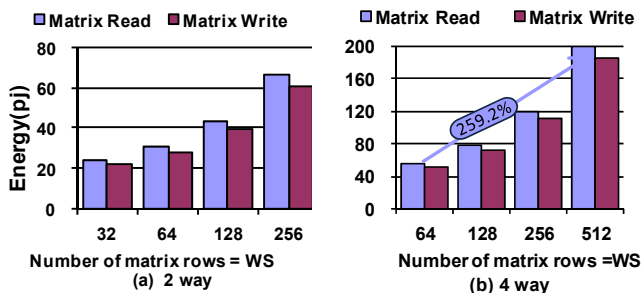


Figure 10: Matrix read and write energy for (a) 2-way and (b) 4-way schedulers.

do not show these results in the interest of space, for a fixed WS and IW, both read and write latencies increase linearly with a very small slope for larger matrix widths. For example, for the 64x20 matrix, the read delay increases by 21.6% as the number of column increases from 16 to 20.

Previous work demonstrate the speed and power efficiency of the conventional (uncompressed) matrix scheduler compared to the CAM-based scheduler [1]. The same results are applicable to the compacted matrix schedulers.

#### 4.3 WAT

Figure 11 report WAT read and write latency as the entry width varies from 7 to 11 (corresponding to schedulers of 32 to 512 entries). For each entry width, four stacked bars are presented each for WATs with 0, 4, 8, and 16 GCs.

For a fixed IW and NoGCs, WAT latency increases logarithmically with the entry width. For example, WAT read and write latencies increase by 13% and 11.6% as the entry width increases from 8 to 11 (WS increases from 64 to 512) for the 4-way WAT with no GCs.

For a fixed WS and IW, both latency and energy increase exponentially with increasing NoGCs. For example, for the 4-way WATs, writes are 1.1%, 2.5% and 6.7% slower with 4, 8 and 16 GCs compared to the non-checkpointed WAT. Previous work shows that for RAT using very few GCs (e.g., four) leads to optimal overall performance (execution time) because of the impact of increasing NoGCs on renaming latency [6]. Our results show a similar trend for the WAT.

Figure 12 shows the WAT energy as a function of the entry width ( $\log_2(WS)+2$ ) and NoGCs. Energy increases slightly by increasing the entry width due to the additional bitlines and sense amplifiers, and longer wordlines. The GC allocation's and restoration's latency and energy are not shown in the interest of space. These percentage of these operations is negligible compared to that of the WAT reads and writes.

A comparison of the 2-way and 4-way delay measurements for the CMS and WAT suggests that latency increases quadratically as the issue width (IW) increases. Increasing IW increases the number of read and write ports for both CMS and WAT; hence, it significantly increases latency and energy.

#### 4.4 Empirical Models

This work also developed empirical delay and energy models for the CMS and WAT. Projections from circuit level delay and energy characterization data available from physical level implementations can be used in formulating empirical models. In other words, our empirical models were developed by extrapolating on the measurements of the organizations that we implemented in 90 nm technology. These models can predict CMS and WAT delay as a function of several architectural parameters. These models can be used by computer architects during architectural level explorations where many different configurations are considered while developing physical level implementations for them is unaffordable due to time and cost constraints. As an example, we present latency models for the 4-way schedulers. Equations (1) and (2) estimate matrix read and write latencies, while equations (3) and (4) estimate WAT read and write latencies.

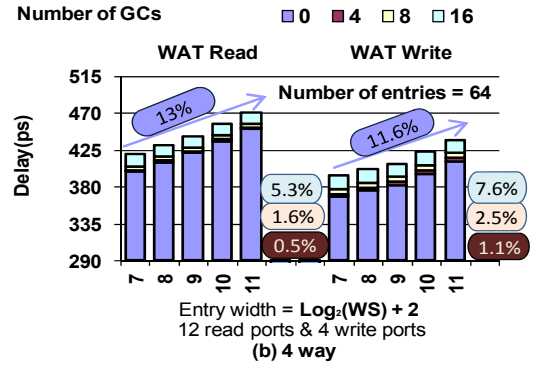
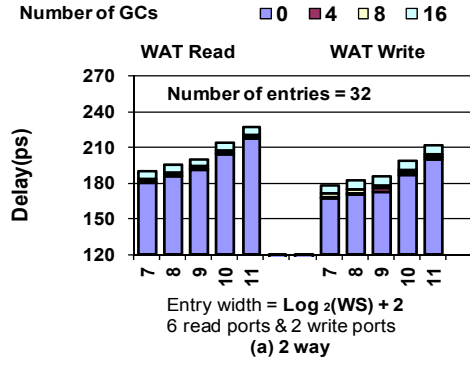


Figure 11: WAT read and write latency for (a) 2-way, and (b) 4-way schedulers

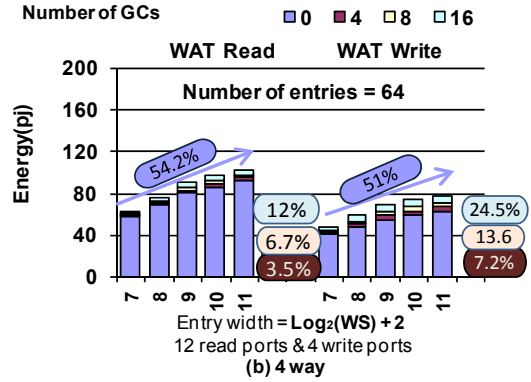
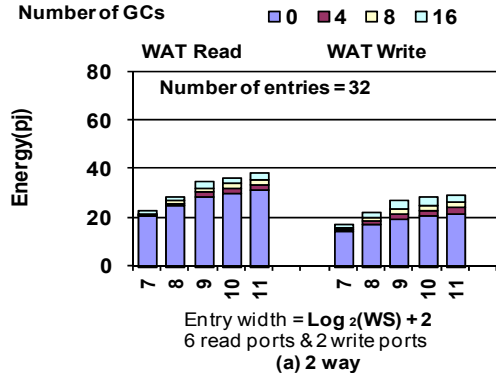


Figure 12: WAT read and write latency and energy for (a) 2-way, and (b) 4-way schedulers.

$$\text{Matrix}_{\text{Read Delay}} (\text{ps}) = 4.1128 \times \text{WS}^{0.6159} \quad (1)$$

$$\text{Matrix}_{\text{Write Delay}} (\text{ps}) = 3.8119 \times \text{WS}^{0.6159} \quad (2)$$

$$\text{WAT}_{\text{Read Delay}} (\text{ps}) = 1.7543 \times e^{0.1524 \times \text{NoGCs}} + 22.77 \times \ln(\text{WS}) + 330.42 \quad \text{NoGCs} \geq 0, \text{WS} \geq 128 \quad (3)$$

$$\text{WAT}_{\text{Write Delay}} (\text{ps}) = 5.5818 \times e^{0.1322 \times \text{NoGCs}} + 13.946 \times \ln(\text{WS}) + 330.42 \quad \text{NoGCs} \geq 0, \text{WS} \geq 128 \quad (4)$$

## 5. CONCLUSIONS

This work investigates the delay and energy variations of the recently proposed CMS and WAT, its accompanying logic. Previous work discussed the speed and scalability advantages of CMS; however, neither actual physical-level investigations nor models were reported for it. Using full-custom layouts in a commercial 90 nm fabrication technology, this work investigated the latency and energy variation of CMS and WAT as a function of the issue width, the window size and the number of global checkpoints. An energy optimization for the matrix was also proposed that reduces energy by 10% or 18% depending on the scheduler size.

Our results show that for a fixed issue width and global checkpoint count, CMS delay and energy increase logarithmically with increasing the number of matrix rows (or the window size). For a fixed window size, issue width and global checkpoint count, the matrix delay increases linearly

with increasing the number of columns. For a fixed issue width and global checkpoint count, WAT latency increases logarithmically for larger window sizes. For a fixed window size and issue width, both WAT latency and energy increase exponentially with increasing checkpoint count. The results of this work support the previously claimed latency and energy scalability of the compacted matrix schedulers.

## REFERENCES

- [1] M. Goshima, K. Nishino, T. Kitamura, Y. Nakashima, Sh. Tomita, S. Mori, "A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors", International Symposium on Microarchitecture, 225-236, Dec. 2001.
- [2] R. Heald et al., "A Third-Generation SPARC V9 64-b Microprocessor", IEEE Journal of Solid-State Circuits, 35(11): 1526-1538, Nov. 2000.
- [3] A. Henstrom, "Scheduling Operations Using a Dependency Matrix", United States Patent 655709, Apr. 2003.
- [4] A. Moshovos, "Checkpointing Alternatives for High Performance, Power-Aware Processors", International Symposium on Low Power Electronics and Design, 318-321, Aug. 2003.
- [5] P.G. Sassone, J. Rupley, II, E. Brekelbaum, G.H. Loh and Bryan Black, "Matrix Scheduler Reloaded", International Symposium on Computer Architecture, 335-346, May 2007.
- [6] S. Palacharla, "Complexity-effective Superscalar Processors", Ph.D. Thesis, University of Wisconsin-Madison, 1998.
- [7] E. Safi, A. Moshovos and A. Veneris, "On the Latency and Energy of Checkpointed, Superscalar Register Alias Tables", To appear in IEEE Transactions on VLSI.
- [8] D. Tarjan, S. Thoziyoor and N. P. Jouppi, CACTI 4.0, HP Labs Technical Report HPL-2006-86, 2006.
- [9] V. Zyuban, "Inherently Lower-Power High-Performance Superscalar Architectures", PhD Thesis, University of Notre Dame, Jan. 2000.