High-Performance Instruction Scheduling Circuits for Superscalar Out-of-Order Soft Processors

Henry Wong, University of Toronto
Vaughn Betz, University of Toronto
Jonathan Rose, University of Toronto

Soft processors have a role to play in simplifying FPGA application design as they can be deployed only when needed, and it is easier to write and debug single-threaded software code than create hardware. The breadth of this second role increases when the performance of the soft processor increases, yet the sophisticated out-of-order superscalar approaches that arrived in the mid-1990s are not employed, despite their area cost now being easily tolerable. In this paper we take an important step toward out-of-order execution in soft processors by exploring instruction scheduling in an FPGA substrate. This differs from the hard-processor design problem because the logic substrate is restricted to LUTs, whereas hard processor scheduling circuits employ CAM and wired-OR structures to great benefit. We discuss both circuit and microarchitectural trade-offs, and compare three circuit structures for the scheduler, including a new structure called a *fused-logic matrix scheduler*. Using our optimized circuits, we show that four-issue distributed schedulers with up to 54 entries can be built with the same cycle time as the commercial Nios II/f soft processor (240 MHz). This careful design has the potential to significantly increase both the IPC and raw compute performance of a soft processor, compared to current commercial soft processors.

CCS Concepts: •**Computer systems organization** → **Superscalar architectures;** •**Hardware** → **Reconfigurable logic applications;** Sequential circuits;

1. INTRODUCTION

The design effort required to build large modern FPGA systems has become a key focus of the industry. Many of the approaches to reduce design time take the form of transforming software directly into hardware. An alternative is to simply implement that software on a processor, and the modern hard processors in FPGAs can take on some of that role. However, various subsystems may require their own processor for performance, security or design isolation reasons, and the limited number of hard processors may not suffice. In that case, the ability to deploy a soft processor is important, and the performance of the soft processor is key to determining how much of the subsystem can be implemented in software. High performance soft processors may be a better vehicle to attach custom-hardware accelerators, given their inherent flexibility.

Despite this, there are still no commercial out-of-order superscalar soft processors, yet there is clear evidence from the hard processor arena that the move to out-of-order results in a significant performance increase. This is illustrated in Table I, which provides SPECint scores between pairs of historical hard processor architectures that moved from in-order to out-of-order microarchitectures. The ratio of each pair of performance numbers in that table are normalized to the same operating frequency to isolate instructions per cycle (IPC) from clock frequency improvements. The table shows

Authors' addresses: {henry, vaughn, jonathan.rose}@ece.utoronto.ca; Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada.

DOI: http://dx.doi.org/10.1145/3093741

that performance increases by a factor of 1.6 to 2 times moving to out-of-order. This performance improvement largely arises from exploiting instruction-level parallelism and tolerating the multicycle latency of memory operations.

If these cycle-count performance gains can be obtained without sacrificing operating frequency ($f_{\max}$), then soft processors can achieve significant performance gains.

This paper focuses on a key component of an out-of-order processor, the instruction scheduler, and explores the microarchitecture and design of scheduler circuits that yield high IPC and large gains in $f_{\max}$ operating frequency. Prior work has often failed to achieve reasonable $f_{\max}$ [Mesa-Martínez et al. 2006; Schelle et al. 2010; Rosière et al. 2012], although at least one prior work has shown out-of-order instruction schedulers on an FPGA at reasonable $f_{\max}$ [Aasaraai and Moshovos 2010]. Our new circuit designs improve on these earlier results, achieving 60% greater $f_{\max}$ for the same size of scheduler, as shown in an earlier version of this work [Wong et al. 2016]. In this paper we significantly expand the work to build *multi-issue* distributed schedulers that have slightly higher $f_{\max}$ than single-issue scheduler circuits of the same total scheduler capacity. We designed four-issue distributed compacting matrix schedulers that can run above 240 MHz (the same as a Nios II/f on a Stratix IV FPGA) at up to 54 entries, although the multiple-issue schedulers cost more than twice the area of a single-issue scheduler of the same capacity.

This paper begins with an overview of instruction scheduling trade-offs in Section 2 followed by a description of the classical scheduling circuits in hard processors in Section 3. Section 4 discusses FPGA circuit designs that are evaluated in Section 6. Section 7 then uses these circuits to build multiple-issue schedulers. Section 9 discusses further optimizations for future scheduler designs.

2. REVIEW OF INSTRUCTION SCHEDULING IN OUT-OF-ORDER PROCESSORS

The key attribute of out-of-order processors is that they execute instructions in *dataflow* order (based on data dependencies) rather than program order. In typical processor pipelines, this dataflow ordering occurs after the instructions are fetched, decoded, and register renamed in program order. They are then inserted into the instruction scheduler, which executes instructions as they become ready. Instructions leave the scheduler when completed. Finally, completed instructions are committed in program order.

The instruction scheduler is responsible for tracking the readiness of every not-yet-completed instruction and for choosing which ready instruction should be executed each cycle. An instruction is ready to execute when all of its source operands are available, having been computed by previously executed instructions.

An instruction scheduler holds a pool of instructions that are waiting to be executed. The *wakeup* portion of the scheduler is responsible for determining when a waiting

Table I. Comparison of SPECint scores between in-order and out-of-order processors and frequency-normalized ratio

Vendor	SPEC Version	In-order		Out-of-Order		Ratio
		Processor	SPEC score	Processor	SPEC score	
MIPS [SPEC 2000]	SPECint 95	R5000 180 MHz	4.8	R10000 195 MHz	11.0	2.1
Alpha [SPEC 2000]	SPECint 95	21164 500 MHz	15.0	21264 500 MHz	27.7	1.9
Intel [SPEC 2000]	SPECint 95	Pentium 200 MHz	5.5	Pentium Pro 200 MHz	8.7	1.6
Intel [Kuttanna 2013]	SPECint 2006	Atom S1260 2 GHz	7.4	Atom C2730 2.6 GHz	15.7	1.6

instruction is ready for execution. It does this by observing which instructions are completing in each cycle and comparing their outputs with the required inputs for each waiting instruction. The *selection* logic is responsible for selecting one of the ready instructions for execution.

Multi-issue schedulers are built around the same wakeup and select circuits. The wakeup and select logic can be extended to handle multiple operations per cycle, or more commonly, the wakeup and select logic can be replicated to achieve the needed instruction execution throughput. In a superscalar processor, the peak throughput of the instruction schedulers usually exceeds that of the overall processor (fetch, decode, and commit), as instruction execution tends to have lower utilization due to data dependencies and the execution units being specialized for particular types of operations (e.g., branches, memory, or integer).

2.1. Scheduler Trade-offs

Processor design is all about trading off IPC, f_{max}, and design complexity. Here we discuss three major design decisions that affect this trade-off.

First, the number of scheduler entries affects how far ahead in the instruction stream instructions can be searched to find instruction-level parallelism (ILP). Larger schedulers (with more entries) increase ILP and IPC, but require more area and tend to have lower f_{max}. For example, Figure 1(a) shows how IPC improves with scheduler size on the two-issue superscalar system we explore in this paper. More scheduler entries improve IPC but eventually give diminishing returns, because other parts of our processor limit the number of in-flight instructions.

Second, the selection policy — how to decide which of several ready instructions should execute — has an impact on IPC. Oldest instruction first is a known good heuristic as it is more likely that an older instruction blocks execution of later dependent operations, but requires tracking the age of entries in the scheduler, which has a hardware cost. Figure 1(b) shows the impact of an oldest-first selection policy compared to random selection. The IPC impact is small for small schedulers because the chance of having more than one ready instruction is lower, but the impact grows to over 15% for large schedulers. Prior out-of-order processors have mostly employed age-based selection [Golden et al. 2011; Farrell and Fischer 1998; Vangal et al. 2002; Gwennap 1997].

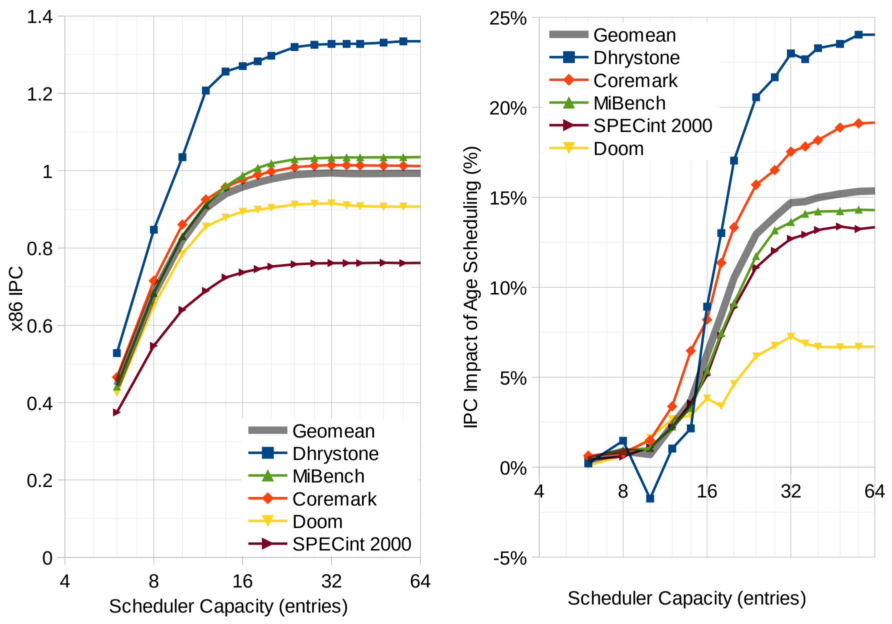
The third key decision is whether wakeup and selection operations complete in a single cycle, which allows execution of dependent operations in consecutive cycles, but makes circuit timing more challenging. A processor will suffer a roughly 10% IPC penalty for adding just one extra cycle of scheduling latency [Brown et al. 2001; Stark et al. 2000].

The trade-offs in multi-issue schedulers are the same, but have many more degrees of freedom. In a distributed multi-issue scheduler, each cluster can potentially have its size, selection policy and latency chosen independently from the others.

In this work, which focuses on fast circuits for high-performance soft processors, we make the following two up-front design decisions: 1) a requirement of single cycle wakeup and 2) an oldest-first selection policy (although we will measure the impact of omitting this for one case). For all scheduler designs we explore, we will measure the impact of a wide range of the number of entries.

3. BACKGROUND ON SCHEDULER CIRCUITS

As described above, schedulers have two key components: wakeup logic to determine which instructions are ready and selection logic to choose among the ones that are ready to execute in the next cycle. In this section we describe how classical hard pro-



(a) IPC vs. Scheduler Capacity (b) Benefit of Age Priority vs. Random

Fig. 1. IPC sensitivity to scheduler capacity and age-based selection policy. The simulated processor has 1 each of branch, ALU, AGU, and store-data execution units, 64 reorder buffer entries, and a peak IPC of 2.

cessor *CAM*-based and *matrix* [Goshima et al. 2001] schedulers perform these two functions.

3.1. Wakeup Logic

CAM-based schedulers track operand dependencies using physical register numbers (after register renaming). Each entry in the scheduler's wakeup array holds an instruction's two source operand register numbers and two comparators that compare them to the destination register number of instructions completing each cycle. A source operand is available after its register number has been broadcast on a result bus, and an instruction is ready when all source operands are ready.

Matrix-based schedulers track dependencies by the position of producer instructions in the scheduler. Each entry (row) of the wakeup array contains a bit vector indicating which *instructions* in the scheduler will produce the source operands. The result bus bit vector indicates which *instructions* are granted execution each cycle, and an instruction is ready when all producer *instructions* have completed.

3.2. Selection Logic

The selection logic is responsible for choosing one instruction for execution from a set of ready instructions. The simplest and fastest selection logic uses fixed priority, prioritizing instructions based only on an instruction's position in the scheduler. However, age-based selection heuristics improve IPC (Section 2.1). Age-based selection can be achieved by maintaining age ordering of scheduler entries, or allowing random ordering of instructions in the scheduler and augmenting the selection logic with age information.

Compacting schedulers insert new instructions at the top, and shift scheduler entries down to fill holes left behind by instructions that have completed execution. Compaction allows a fast fixed-priority selection circuit to be used. The main drawback is in the power consumption of shifting the scheduler entries and the delay of the multiplexer required for shifting.

Explicitly tracking instruction age makes selection logic more complicated due to dynamic priority. There are many methods to track age, including precise and approximate methods (e.g., [Vangal et al. 2002; Golden et al. 2011]). Our matrix scheduler uses an age matrix, a precise method that uses a matrix where the bits in each row indicate which instructions are older than the instruction occupying the row [Sassone et al. 2007].

3.3. Fusing Wakeup and Selection

Instruction schedulers are usually implemented with separate wakeup and select circuits, performed sequentially. For matrix schedulers on FPGAs, the wakeup logic's wide OR gate reductions and the selection logic's pick-first-ready scan logic are both implemented as trees of LUTs. In some circuits, it is possible to reformulate the logic function to combine two reduction or scan operations into one, improving delay. The sum-addressed decoder is one well-known example of this kind of transformation [Lynch et al. 1998].

Inspired by this strategy, we present a new scheduler circuit, the fused-logic matrix scheduler, that combines both the wakeup wide-OR and select linear scan operations into a single tree of LUTs. This circuit is faster than both the CAM and age-based matrix schedulers for most scheduler sizes.

4. DETAILED CIRCUIT DESIGNS

This section discusses the circuit designs of the three scheduler circuits we implemented on the Stratix IV FPGA: a compacting CAM scheduler, a non-compacting matrix scheduler, and our new fused-logic matrix.

4.1. CAM

Our CAM scheduler implementation, shown in Figure 2, uses compaction to maintain age ordering and allows back-to-back scheduling of dependent operations. In each cycle, ready bits are used to select an instruction for execution. The selected instruction's destination tag is then broadcast on the result bus, and consumers of the newly-produced register are woken up.

4.1.1. Wakeup. Each entry in the CAM wakeup logic has two source operand tags and an associated pair of comparators. The comparators monitor the result bus for a physical register number that indicates when an operand becomes available. An instruction is ready when all operands are available and has not already been selected for execution. The register number is assumed to be large enough to hold at least twice the scheduler capacity, so comparators compare two $log_2 N + 1$ bit numbers. Each 6-LUT can do three bits of a comparison, which is followed by an AND tree, so the total logic depth for two comparators is roughly $log_6(4(log_2 N + 1))$. The ready bit of every entry, forming a *ready vector*, is sent to the selection logic.

4.1.2. Compaction. Our CAM wakeup logic can shift down to eliminate up to one hole per clock cycle. This is enough because only one instruction can be selected for execution each cycle, so new holes are created no faster than one per cycle. Compacting by one position occurs through 2-to-1 multiplexers immediately before the set of pipeline registers.

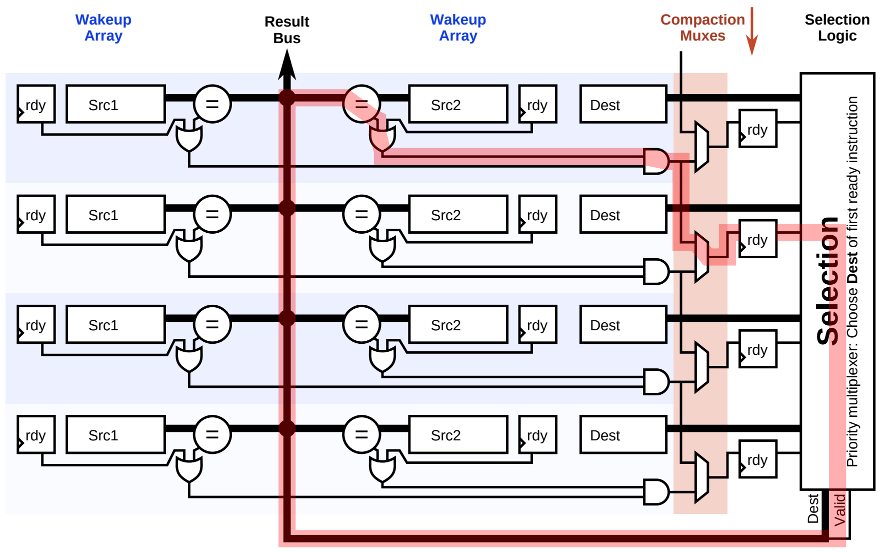


Fig. 2. CAM wakeup circuit. Entries compact downwards. An example critical path is highlighted in red.

The control logic to decide whether each entry should shift down is a prefix OR operation, computing for each entry whether there is a vacant entry at or below the current position. This prefix OR function is implemented using a tree of LUTs with logic depth $log_6(N)$ using a radix-6 Han-Carlson prefix tree with sparsity 6 [Harris 2003]. The radix and sparsity were chosen to suit a 6-LUT FPGA architecture.

4.1.3. Selection. The CAM scheduler's selection logic performs two functions. It must grant execution to the oldest ready instruction, and it must also select that instruction's destination register number and broadcast it on the result bus to wake up dependent operations.

One grant signal per entry indicates whether that entry has been selected for execution. Oldest-ready grant logic is implemented using the same radix-6 Han-Carlson prefix tree used for computing the wakeup compaction multiplexer control signals.

Generating the destination register is done with a priority multiplexer that selects the destination register number field of the oldest ready instruction. The priority multiplexer has a logic depth of $log_4 N$ LUTs, implemented as a radix-4 tree using 7-input ALMs[1]. Figure 3 shows this circuit.

4.2. Matrix

The matrix scheduler implementation tracks dependencies of instructions using a wakeup matrix of dependency bits. We evaluated the matrix scheduler both with and without age-based selection. In each cycle, the ready bits (and age matrix) are used to select an instruction for execution, and grant signals are broadcast into the wakeup matrix to wake up dependent instructions.

4.2.1. Wakeup. The wakeup array, shown in Figure 4, consists of a matrix of dependency bits, used to track when each instruction is ready. The dependency bits are cleared by the grant signals as instructions are granted for execution. When all of

[1]A Stratix IV ALM can implement 7-input functions that can be expressed as a 2-to-1 multiplexer selecting between two 5-input LUTs that share 4 inputs [Altera 2011].

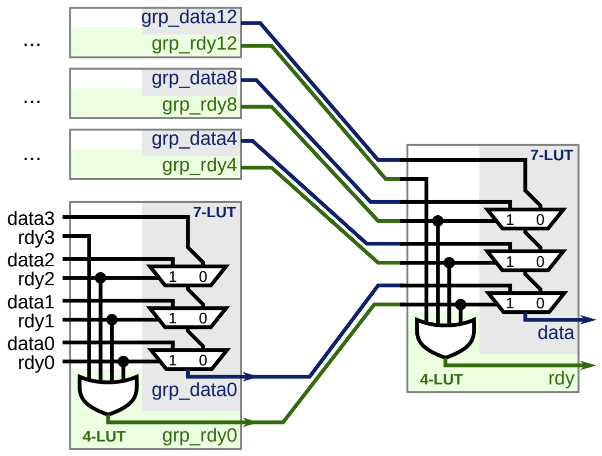


Fig. 3. Priority multiplexer built from radix-4 blocks, each of which is a size 4 priority multiplexer. This figure shows how a depth-2 tree can implement a 16-entry priority multiplexer. **grp_data** and **grp_rdy** fit in a single 7-input and 4-input LUT, respectively. Used for CAM selection and fused-logic matrix selection.

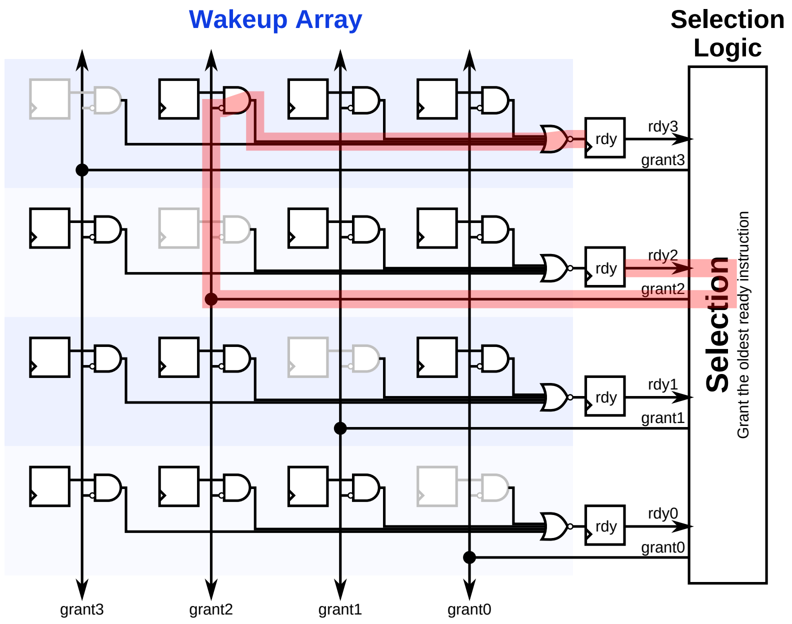


Fig. 4. Matrix wakeup circuit. Diagonal is omitted as an instruction does not depend on itself. An example critical path is shown.

the bits in a row are ready or are just granted this cycle, the ready bit for the row is set, resulting in a N-wide NOR of required-and-not-granted functions. An N-wide NOR of two-input functions ($2N$ inputs) can be computed with a tree of 6-input LUTs with depth $log_6(2N)$.

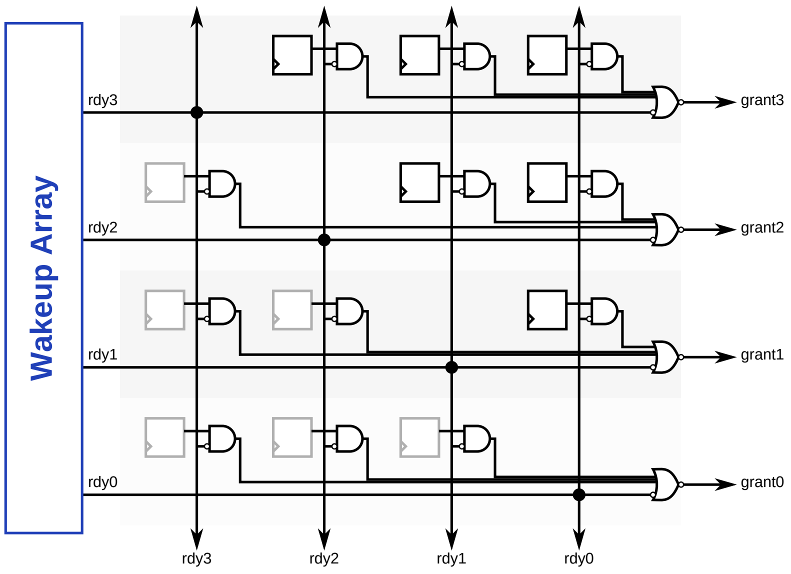


Fig. 5. Age matrix selection circuit. An entry is granted if it is ready and the grant is not "killed" by a higher-priority grant. Lower triangle registers are omitted as it is the complement of the upper triangle.

4.2.2. Position-Based Selection. The position-based select logic grants a ready instruction if there are no other ready instructions before it. As scheduler position does not correlate with instruction age, position priority is essentially random priority. It is implemented using the same radix-6 Han-Carlson prefix tree as the CAM selection grant logic (Section 4.1.3).

4.2.3. Age-Based Selection. The age-based selection logic uses an age matrix to dynamically specify age priority, as the scheduler entries are not ordered by age. An age matrix specifies for each row which other instructions are older than itself. A ready instruction is granted execution if there are no older ready instructions, which is a N-wide NOR of ready-and-older functions. This is computed with a radix-6 tree with logic depth $log_6 2N$, shown in Figure 5. The age matrix has symmetry (if instruction A is older than B, then B must be younger than A), so we omit half of the registers to reduce area.

Compared to the compacting CAM scheduler, the 2-to-1 compaction multiplexer and radix-4 priority multiplexer are removed from the critical loop. Dynamic-priority grant logic is slower than fixed-priority grant logic, with depth $log_6(2N)$ rather than $log_6(N)$. The CAM and matrix wakeup delays scale differently with scheduler size, favouring matrix wakeup for small sizes, but CAM wakeup for large sizes.

4.3. Fused-Logic Matrix

With separate wakeup and selection circuits, both the CAM and matrix schemes contained two reduction trees of LUTs in their critical path: one for the wakeup logic, and one for selection. To further improve speed, we endeavoured to create a scheduler with a critical loop containing only *one* reduction tree that would perform *both* wakeup and select functions.

The resulting design is a compacting matrix scheduler with fused wakeup and select logic. Dependency information is expressed as a matrix of dependency bits like the matrix scheduler, but select *and* wakeup are computed using a single radix-4 tree of

LUTs. Conceptually, instead of having one instance of selection logic broadcasting its result to per-entry wakeup logic, the selection logic is also replicated per entry and merged with the wakeup logic. Figure 6 shows this arrangement.

4.3.1. Wakeup and Select. Scheduler entries are ordered by age using compaction, so the selection uses fast fixed-priority selection. Each row has a combined select-and-wakeup circuit. The two inputs to each instance of the select-wakeup logic are a ready vector indicating which instructions are ready for execution, and a dependence vector indicating whether the instruction in the current row is dependent on each instruction. The select-wakeup logic computes whether the current instruction depends on the oldest ready (i.e., selected) instruction. If so, this means one dependency has been satisfied, and a two-bit counter storing the number of outstanding dependencies is decremented. An instruction is ready when the counter reaches zero. Grant logic is still used to generate grant signals to clear dependency bits in the matrix, but is now moved off the critical path.

The select-wakeup logic is equivalent to a priority multiplexer, implemented using the circuit in Figure 3, which is a radix-4 tree of 7-input ALMs with a logic depth of $log_4 N$. The priority multiplexer finds the first ready instruction and selects the one bit of data indicating whether the instruction depends on the selected (oldest ready) instruction.

There is more preprocessing that needs to be done than for the matrix scheduler. In addition to encoding dependencies as positions in the scheduler, we also need to count *how many* dependencies are outstanding, which is a population count of the dependence vector. In this implementation, the single-cycle preprocessing is a critical timing path. However, because it is outside the wakeup-select loop, it should be possible for future implementations to further pipeline preprocessing without giving up the ability to schedule dependent instructions in back-to-back cycles.

4.3.2. Compaction. Compaction of a matrix is more complex than for a CAM-based scheduler. In a matrix scheduler, the bits in each row indicate the scheduler position of the parent instructions, whose positions will also change due to compaction. As scheduler entries are compacted downwards in a matrix scheduler, the dependency bit vectors are also compacted horizontally to track the changing instruction positions as they shift down the scheduler. Fortunately, the extra compaction logic is off the critical wakeup and select loop.

5. EVALUATION METHODOLOGY

The main objective of this work is to evaluate area and f_{max} of different circuit-level implementations of broadcast-based instruction schedulers. We build optimized circuits for the circuits described in the previous section (CAM, non-compacting matrix, and fused-logic matrix) targeting a Stratix IV FPGA (smallest, fastest speed grade, EP4SGX70-C2) using Quartus 15.0 using default settings, but register retiming was enabled where it made a difference).

We sweep scheduler capacity (entries) and observe area and f_{max} scaling as the scheduler size varies. Circuit results are the mean of 100 random seeds. We focus on area and delay here because all of the scheduler circuits have nearly the same cycle-by-cycle behaviour: they wake up all ready instructions every cycle and select either a random instruction or the oldest ready instruction for execution.

Although the focus of this paper is on scheduler circuit design, we need to evaluate IPC on at least one processor microarchitecture. In this paper, we model a fairly typical two-issue out-of-order x86 microarchitecture. Our simulator is based on the Bochs [Lawton 1996] functional model of an x86 system, modified with a new cycle-level model of an x86 soft processor microarchitecture we are currently developing. The

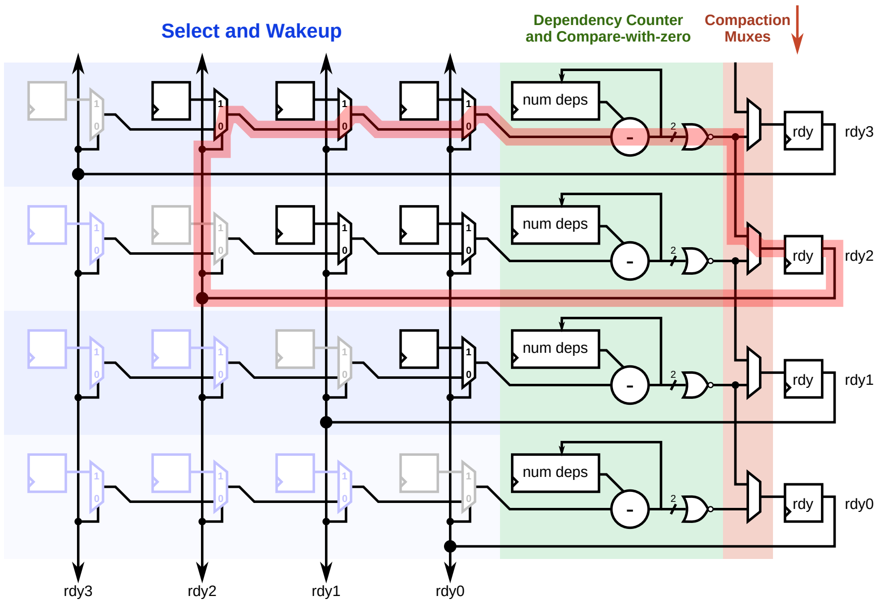


Fig. 6. Fused-logic matrix circuit. Selection and wakeup are merged and implemented as a one-bit wide priority multiplexer and a two-bit counter. Lower triangle is omitted as instructions do not depend on itself or future instructions.

processor front-end decodes x86 instructions into micro-ops. The instruction scheduler sees only micro-ops and is unaware of x86 instructions. Thus, instruction scheduler design is mostly independent of instruction set, other than in details such as the number of source operands per micro-op. We used a set of workloads totalling 25 billion x86 instructions. These include billion-instruction samples from SPECint2000 (reference inputs), CoreMark, Dhrystone, the MiBench suite [Guthaus et al. 2001], and Doom (first-person shooter game running under DOS).

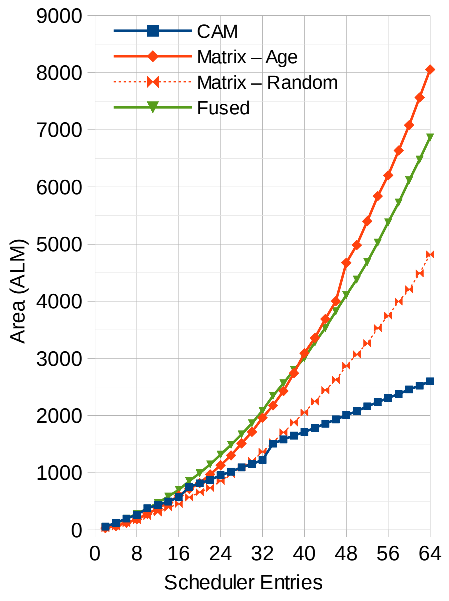
6. CIRCUIT DESIGN RESULTS

This section presents area and f_{max} results of implementing CAM, matrix, and fused-logic matrix scheduler circuits on a Stratix IV FPGA.

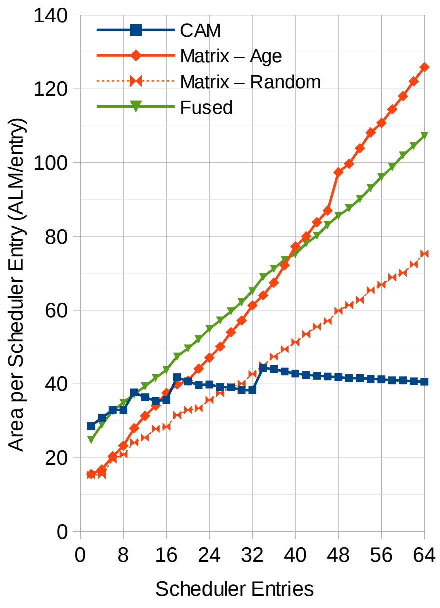
6.1. Area

Figure 7 compares the area of the three scheduler circuit types as scheduler size changes. The two matrix schedulers scale similarly, as the size of the matrix grows quadratically with the number of scheduler entries, but the matrix with position-based selection is smaller as it does not have an age matrix. CAM schedulers have better area at large sizes, as the size of comparators increases logarithmically (register number width) but the size of each matrix row's OR gate increases linearly. This can be seen more clearly when plotting area per entry, in Figure 7(b).

For out-of-order FPGA soft processors, we are mainly interested in small schedulers, generally below 20 entries, where area does not differ greatly between scheduler types. The poor area scaling of matrix wakeup logic for larger schedulers was also true in custom CMOS. [Goshima et al. 2001].



(a) Area vs. Capacity



(b) Area per Scheduler Entry

Fig. 7. Area of four scheduler types. Area per entry gives insight into scaling trends with scheduler size.

6.2. Delay

Figure 8 shows the achieved f_{max} for the three scheduler circuit types as scheduler capacity is varied. The general trend, unsurprisingly, is that larger schedulers are slower. The delay for the matrix schedulers increase faster than CAM schedulers at large sizes. On an FPGA where there are no fast wired-OR circuits, we see smaller improvements vs CAM than those reported for custom CMOS implementations [Goshima et al. 2001].

Among the three age-based schedulers, our new fused-logic matrix scheduler is the fastest option beyond 6–10 entries, though at very large sizes, excessive area causes poor routing delays. CAM schedulers are slow at small sizes, only being faster than the age-based matrix scheduler beyond 24 entries. The position-based matrix scheduler ("Matrix — Random") is the fastest, but gives up age-based scheduling.

At small sizes (below 20 entries), both types of matrix scheduler are faster than CAM schedulers, with little difference in area. The largest age-based scheduler that can match the clock speed of a Nios II/f on the same FPGA (240 MHz or 4.2 ns) is around 20 entries for CAM, 22 entries for age-based matrix, and 42 entries for the compacting fused-logic matrix. A 44-entry position-based matrix scheduler also fits in a 4.2 ns period, but has limited usefulness at this size given the large IPC degradation of the selection policy. For comparison, current high-end x86 processors have 40–60 scheduler entries [Golden et al. 2011], while earlier out-of-order processors have far less (20 for Alpha 21264 [Farrell and Fischer 1998], 16 for Pentium 4 [Vangal et al. 2002]). This suggests that moderately aggressive out-of-order designs are feasible on FPGAs even when targeting the same frequency as simple single-issue in-order soft processors.

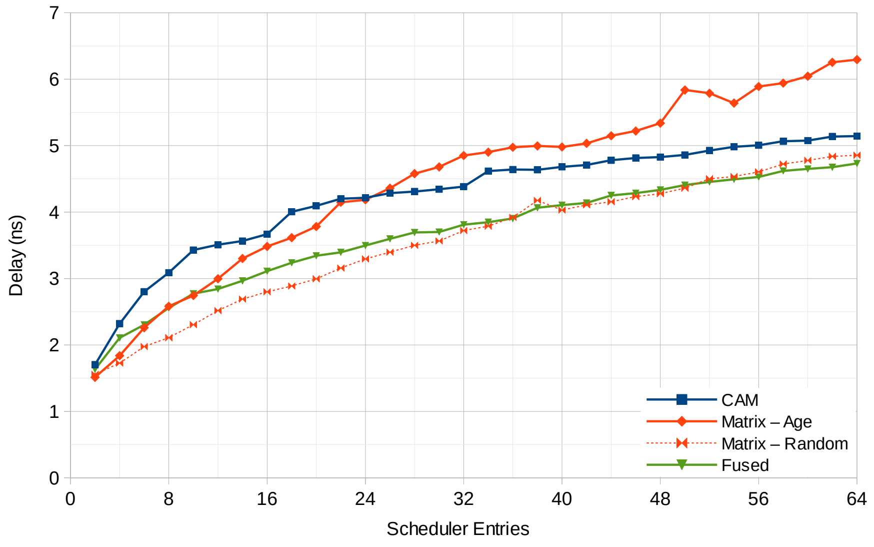


Fig. 8. Delay of four scheduler types at varying capacities on a Stratix IV FPGA

7. MULTIPLE-ISSUE SCHEDULERS

In the previous sections, we presented circuit designs for single-issue schedulers. Using one of these circuits alone could form the scheduler of a single-issue out-of-order processor. Superscalar processors that execute more than one instruction per clock cycle can provide large improvements in IPC, but require larger schedulers to find instruction-level parallelism to keep execution units utilized. For example, in our simulated processor design, a dual-issue pipeline has about 70% higher IPC than a pipeline with a single-issue scheduler, but needs about 4 times as many scheduler entries to reach that performance (Figure 12(b)).

To build schedulers for multi-issue soft processors, the circuits presented earlier serve as fundamental building blocks. Single-issue schedulers need to select one ready instruction for execution and (usually) wake up one instruction per clock cycle. Extending this to multiple issue is, in principle, fairly straightforward. In a multi-issue scheduler, several ready instructions are selected each cycle (one per execution unit), and instructions can be woken up by any of the several instructions that are completing each cycle. In practice, doing multiple actions (wakeup and select) within a clock cycle can easily result in slow circuits. The design space that needs searching is also potentially much larger.

Extending wakeup from one per cycle to several per cycle requires each scheduler entry to monitor multiple result buses, and wake up if *any* of the result buses are broadcasting the desired source operand (for CAM-based) or instruction (for matrix-based). For CAM-based schedulers, wakeup occurs using register number tag comparisons between each source operand and all result buses. Multiple result buses thus requires multiple comparators per source operand to detect whether the desired source operand was produced by *any* result bus. This can be simplified to a multiplexer and one comparator if it is known which result bus produces the necessary source operand. For matrix-based schedulers, no modification to the wakeup array is needed to support multiple issue, nor to support the use of instructions with more source operands. When multiple instructions execute in the same clock cycle, multiple grant wires are asserted, which can clear multiple dependency bits in each row in the same cycle. For our fused-logic scheduler, the dependency-tracking mechanism requires decrementing a counter that counts the number of outstanding dependencies. Extending this to support multiple issue requires either the counter to support decrementing by more than one each clock cycle, or multiple counters (one per result bus) and waking up the instruction when all counters reach zero. Both options increase the logic depth by at least one LUT logic level, making the fused-logic circuit unattractive for multiple issue.

A multiple-issue scheduler requires the selection logic to be extended to choose multiple ready instructions for execution each cycle. There are several common options to implement multiple issue that trades flexibility (higher IPC) with circuit complexity. If the execution units in the processor are identical, one option is to use selection logic that picks the first *several* ready instructions for execution. This option is usually impractical because selection circuit delays grow with the issue width, as selecting each subsequent instruction requires knowing which other instructions have also been previously selected this clock cycle. By assigning each instruction to an execution unit before scheduling (possibly by necessity, if execution units are specialized), the dependency between each instruction selection circuit is broken, and each selection circuit searches the scheduler for the first ready instruction that has been pre-assigned to the corresponding execution unit. This arrangement is often called a *unified* scheduler, as all instructions occupy the same pool of scheduler entries. The circuit can be further simplified by partitioning the scheduler entries between the execution units, so that each execution unit is attached to its own scheduler. This *distributed* scheduler further simplifies circuits but makes less efficient use of scheduler entries. Because there are multiple schedulers that are sized independently, a distributed scheduler has a larger design space to search.

The rest of this section describes our design of a compacting-matrix-based distributed scheduler for use in a two-issue superscalar x86 soft processor that has four different execution units. We compare the IPC between unified and distributed designs, choose capacities for each scheduler in a distributed design, then present area and cycle time results of the four-way distributed scheduler circuit.

7.1. Unified vs. Distributed Scheduler

A unified scheduler has a single pool of instructions from which ready operations are chosen, while distributed schedulers use one private scheduler per execution unit. Unified schedulers make more efficient use of scheduler entries because the relative demand for each execution unit varies dynamically, and a distributed scheduling scheme stalls the pipeline whenever *any* of its schedulers are full even if not *all* schedulers are full.

However, circuits for unified schedulers are slower than distributed, as each execution unit's selection logic must search the entire pool of waiting instructions rather than just the scheduler attached to its own execution unit [Brown et al. 2001]. The wakeup logic is similar for both distributed and unified designs: every executed instruction must broadcast to every scheduler entry.

For our x86 design using complex micro-ops, another complexity of using unified schedulers is the required peak *enqueue* throughput. Renaming and issuing two micro-ops per cycle into the schedulers can produce up to 6 independently scheduled operations (2 micro-ops, each with a load, arithmetic, and store). With a distributed scheduler, these 6 operations are guaranteed to be distributed among at least three different schedulers, so each scheduler only needs to enqueue up to 2 operations per cycle. Using a unified scheme would require the unified scheduler to enqueue up to 6 operations per cycle.

For circuit complexity reasons, we prefer a distributed scheduler design. However, we still wish to know the IPC cost of choosing a distributed design. A proper compari-

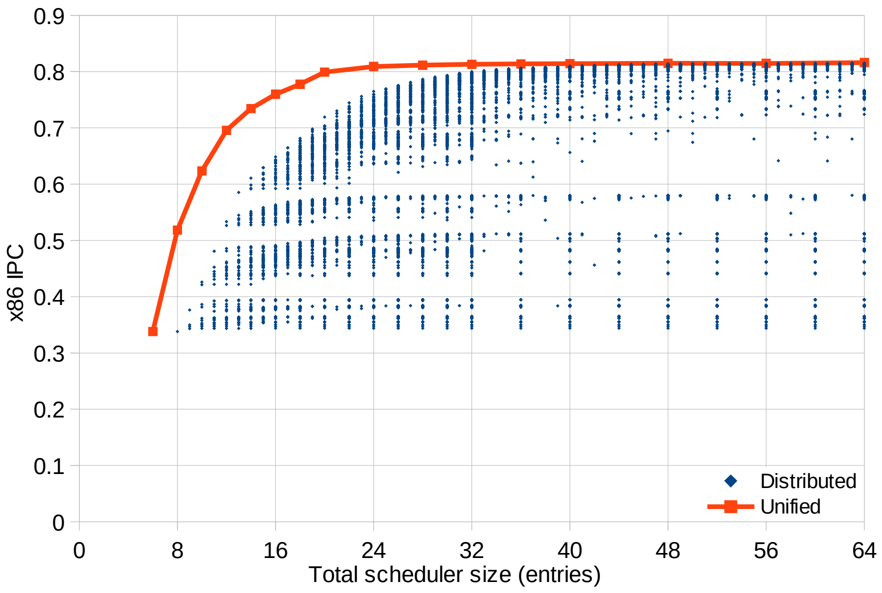


Fig. 9. Searching the four-component distributed scheduler design space. For distributed schedulers, each data point is one configuration in the four-dimensional design space.

son requires exploring the distributed scheduler design space, which will be discussed in the next section. Figures 9 and 12(b) show the final results on two different benchmark sets. The charts show that distributed schedulers need more total entries to achieve the same IPC (e.g., 20 unified and 32 distributed entries have similar IPC).

7.2. Tuning the distributed scheduler

Our processor has four execution units (branch and complex operations, integer arithmetic, address generation and loads, and store-data), so a distributed scheduler has four independently-sized components, one for each execution unit. Because each execution unit is specialized for a particular operation type, some of which occur more frequently than others, it is not immediately obvious how to determine the size of each component that maximizes performance for a given total size. Thus, we need to search the four-dimensional design space, then use the results to guide how each individual scheduler should be sized.

It is easy to measure the IPC for unified schedulers as there is only a single parameter (total capacity), but a distributed scheduler has, for us, 4 independent size parameters. If we limit the maximum total scheduler size to 64 entries, the design space contains just under half a million design points, each requiring simulation. To make this search tractable, we explored the design space using a randomized search, optimizing for maximum IPC at each total scheduler capacity. This is of course a simplified cost function, as it omits details such as cycle time (which is also influenced by the largest scheduler component and not just the total capacity) and that some operation types cost more than others (e.g., stores do not need to wake up dependents). For each design point, we obtained IPC using a cycle-level simulation with a reduced benchmark set, which consists of 279 million instructions taken from five benchmarks: Dhrystone (70M instructions), Mibench (dijkstra 41M and mad 27M), and SPECint2000 (gzip 70M, mcf 70M). This is a reduction from the workload of 25 billion instructions used in the rest of the paper.

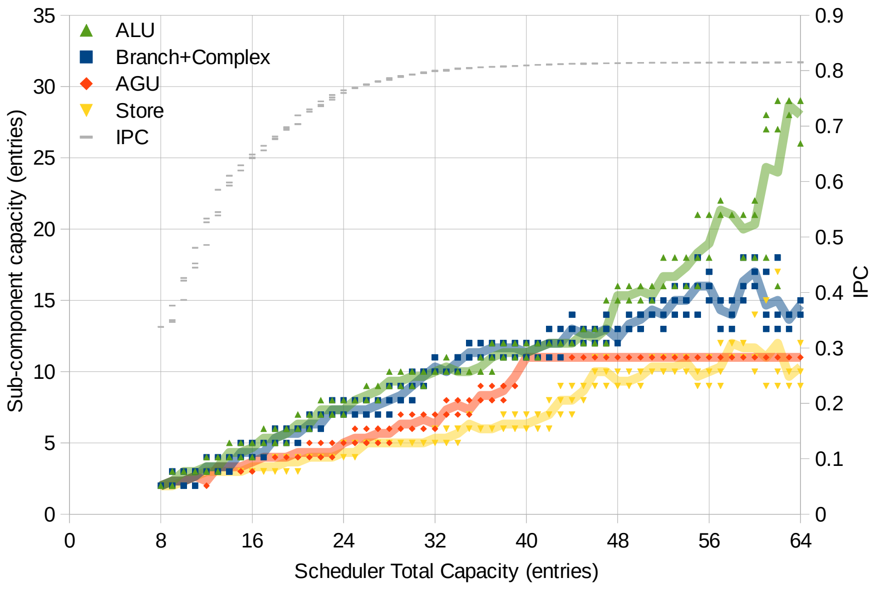


Fig. 10. Best four-component distributed scheduler parameters vs. total scheduler capacity

We first coarsely explored the design space (points where component sizes differ by a multiple of 4, 3060 points). Then, for each scheduler size, we pick several design points at random, strongly skewed toward the best points for that size, and create a new design point for each by perturbing its four parameters randomly. We repeat this process until we were satisfied that the search space was adequately explored. In total, we explored 10 635 points using 150 CPU-days, or about 2% of the total design space of 487 635 points. Figure 9 shows the IPC of every design point we simulated. Points closer to the pareto-optimal curve are more densely explored. Also plotted on the chart for comparison is the performance of unified schedulers on the same reduced benchmark set. As expected, unified schedulers achieve the same IPC with fewer total entries.

To use this data to guide how to size each scheduler component, we take a subset of the data points (only the top-performing three points) at each total size from Figure 9 and examine the four parameters that produced each data point. The four lines in Figure 10 show the average size of the sub-scheduler used by each execution unit type (y-axis) for each total capacity (x-axis) of the distributed scheduler. The IPC of this subset of data points is also plotted again for reference. Not surprisingly, the best schedulers do not have equally-sized components. However, the relative sizes of the four schedulers do not match their relative utilization (17%, 35%, 37%, 11% for Branch/Complex, integer ALU, AGU, and stores, respectively). Perhaps most surprising is that branch/complex operations "prefer" scheduler sizes similar to the integer ALU, despite being utilized only half as often. We speculate this is a side effect of our processor design that executes branches and some complex operations in-order, which causes operations to occupy the scheduler longer than a fully out-of-order ALU would.

For the remainder of this section, we choose the sizes of each scheduler component based on a linear fit to the portion of the plot with less than 40 total entries. The portion greater than 40 entries behaves somewhat erratically, likely because random noise dominates when the schedulers have more entries than required to achieve maximum IPC.

To verify that our reduced benchmark set was reasonably representative of the full 25-billion instruction workload, we simulated scheduler sizes from 8 through 64 on our full benchmark set, using our linear-fit rule for sizing each execution unit's scheduler. Figure 12(a) shows these results. Figure 12(b) compares this 4-way distributed scheduler with the IPC of a 4-way unified scheduler (from Figure 1) and also to a single-issue out-of-order processor with a 1-way unified scheduler. Two-issue (with 4-way scheduling) superscalar execution has a large 70% IPC increase over single-issue, but requires larger schedulers to achieve that performance, while a single-issue out-of-order processor is largely insensitive to scheduler capacity. Single-issue performs better for very small schedulers due to a side effect of our design: The scheduler is considered "full" if there are not enough free entries to accommodate the worst-case output from the renamer, which is 6 operations for dual-issue but 3 operations for single-issue (thus, a dual-issue processor effectively loses 2–3 scheduler entries compared to a single-issue processor).

7.3. Proposed Scheduler Microarchitecture

In addition to supporting multiple issue with a distributed scheduler as discussed in the preceding section, we also extended our scheduler to implement details of our target instruction set (x86), such as supporting different instruction types (e.g., arithmetic vs. load), more source operands per operation (up to 3), and different register types (e.g., general-purpose vs. condition codes).

Our processor can decode, rename, and commit two complex micro-ops per cycle, and has at most 64 micro-ops in flight (reorder buffer size). Each complex micro-op is then broken up into up to three operations (load, arithmetic, store) that are executed by four different types of execution unit, with a peak execution throughput of four operations per cycle. Deciding on the overall processor issue width and arrangement of execution units is outside the scope of instruction scheduler design. Each scheduler (one per execution unit) can enqueue at most two operations per cycle, and select and dispatch one operation per cycle. In aggregate, the four schedulers can enqueue 6 operations (two complex micro-ops with three operations each) and select and dispatch 4 operations per cycle (one of each type). Figure 11 shows a high-level schematic of how our four-component distributed scheduler is built out of smaller matrix scheduler circuits.

Unlike for a single-issue processor, the wakeup matrix for each scheduler is much wider than it is tall, as each entry in the scheduler can wait for operands arriving from any scheduler, but the aggregate dimensions of the wakeup matrix is roughly square. It is not precisely square because stores do not wake up any dependent instructions, as they only write data into the store queue.

To help with cycle time, we relaxed the wakeup latency from our earlier assumption of single-cycle latency for some of the less critical paths. Complex ALU operations and memory loads never complete in a single cycle, so the branch/complex and AGU schedulers do not need single-cycle wakeup of its dependents. We also added an extra cycle for operands going to the branch/complex ALU.

According to Figure 12(a), schedulers with greater than 32–40 entries are wasteful, while those much smaller than 16 have large IPC losses. The final choice of scheduler capacity in the range of 16–32 entries depends on other factors such as the frequency target of the processor and the area budget.

7.4. Circuit Design

At this point, we have decided that we should build a distributed scheduler with four sub-schedulers of different sizes (Figure 11). We chose to build the scheduler from four *compacting matrix* scheduler circuits. This design draws from lessons learned from both the fused-logic and matrix schedulers discussed earlier in Section 4.

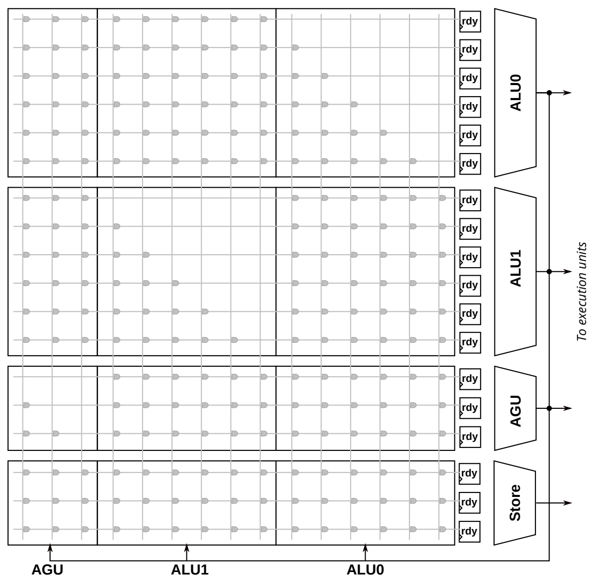


Fig. 11. Block diagram of a 4-way compacting matrix distributed scheduler. Each cluster can be a different size. Stores do not need to wake up dependent operations.

A compacting matrix uses wakeup and selection logic found in a matrix scheduler, but tracks instruction age by compaction as used by fused-logic scheduler instead of an age matrix. This design attempts to work around shortcomings in each design.

We saw in Figure 8 (Section 6.2) that the matrix scheduler using an age matrix had higher delay than the fused-logic scheduler. The same figure also suggested that much of the disadvantage for the age-based matrix scheduler may have been the use of an age matrix, as a matrix scheduler without the age matrix (random priority) had lower delay. The choice of using matrix wakeup logic is due to its more straightforward design making it easier to extend to allow more source operands and multiple wakeups per cycle. The fused-logic wakeup logic's use of a counter to count the number of remaining operands becomes problematic because the number of dependencies in our x86-based micro-ops can be up to 3, and up to three dependencies can be satisfied each cycle (instead of one), which complicates the counter circuit. Extending the counter design will add at least one LUT logic level to the critical path. In contrast, no changes need to be made to the matrix scheduler's wakeup logic to support more source operands or multi-issue. Using a compacting matrix circuit design attempts to gain some of the fused-logic design's speed (by using compaction instead of an age matrix) without inheriting its extra complexity (by using the simpler matrix wakeup logic).

Like all matrix schedulers, we need preprocessing logic to map source register numbers to its producer operation's location in the matrix scheduler. We use an array of one-hot bits to indicate which scheduler operation produces a given *logical* (not physical) register. Because querying this mapping happens in-program-order partially in parallel with register renaming, we can map logical registers directly to scheduler location. This produces a smaller mapping table because there are fewer logical registers than physical registers. This mapping table compacts along with the compaction of scheduler entries. The preprocessing logic is a substantial portion of the scheduler (about 60% of the scheduler's area), and is a significant disadvantage of matrix-based designs. All of our f_{max} and area numbers include this overhead.

7.5. Circuit Results

We synthesized distributed schedulers from size 8 through 64 for a Stratix IV FPGA, where the component sizes were determined following the guideline from Section 7.1. Figure 13 plots the area and delay of these designs. Area and f_{max} results are the median of 50 random seeds.

In Figure 13(a), the cycle time of the CAM and matrix (random priority) are plotted again (from Figure 8) for comparison. One interesting observation is that the distributed multi-issue scheduler has a slightly *faster* cycle time than a single-issue scheduler of the same total capacity, because a distributed scheduler's individual components are smaller. For example, the 32-entry distributed scheduler has components of at most 10 entries each (10, 10, 7, 5) that operate in parallel.

The area of the multi-issue scheduler is substantially greater, as seen in the plot of area per scheduler entry in Figure 13(b). For comparison, the area per entry of single-issue CAM and matrix (random) circuits (from Figure 7(b)) are also plotted. A large contributor to this increase is the need to map up to 16 (instead of 2) source operands per cycle to the scheduler entry of the instruction that produced the register's value. The wakeup and select logic area alone is similar to the single-issue fused-logic scheduler and slightly higher than the matrix (random priority) due to the addition of compaction.

As mentioned in the previous section, 16–32 scheduler entries is a reasonable range to build from an IPC perspective. Using our compacting matrix circuit results in frequencies of 325–280 MHz (faster than single-issue schedulers of the same size) and area of 1750–3750 ALM (substantially worse than a single-issue scheduler).

7.6. Overall Performance

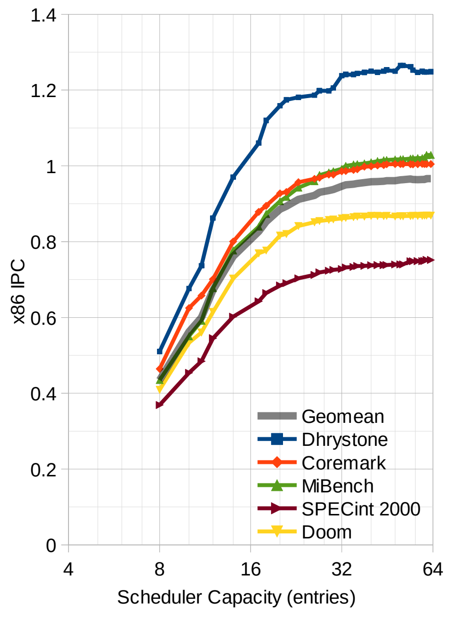
Figure 14 combines the IPC (Figure 12(a)) and cycle time (Figure 13(a)) results into a single plot, showing the trade-off between IPC and frequency. The curved grid lines mark instruction throughput in MIPS, which is the product of IPC and frequency in MHz. Each point on the plot shows the IPC and frequency for the scheduler of a particular capacity. For example, the 17-entry scheduler has 0.83 IPC, 324 MHz, and 268 MIPS, while a 32-entry scheduler achieves almost the same throughput (264 MIPS) but does so with a higher IPC (0.95) and lower frequency (279 MHz). While these two design points have similar overall performance, the latter is easier to build as it imposes a less stringent timing constraint on the rest of the processor.

There is a fairly large region with similar MIPS between 14 and 33 entries (275–347 MHz, 0.95–0.76 IPC, 260–275 MIPS). This may allow a fairly large range of designs to suit the frequency target of the rest of the processor without major performance compromises. Of course, larger schedulers do incur a higher area cost.

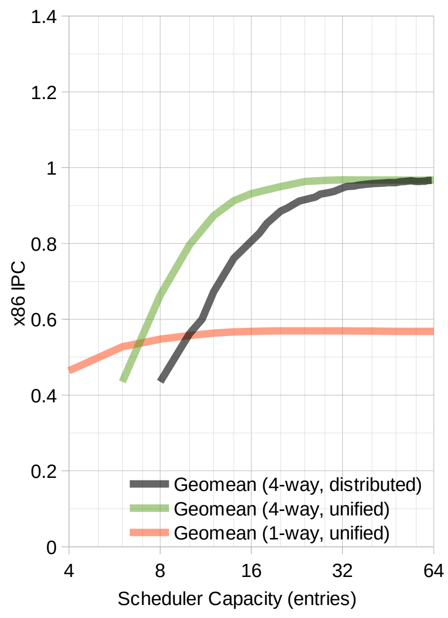
We expect soft processor designs to have frequencies toward the lower end on this chart, as the Nios II/f only runs at 240 MHz [Altera 2015] on a Stratix IV FPGA, and we expect a more complex out-of-order processor would not be able to exceed the Nios II/f's frequency by much. A lower processor frequency target suggests the use of larger, higher IPC schedulers, and motivates more area-efficient designs even if it means some frequency loss for the instruction scheduler.

8. COMPARISONS TO PREVIOUS WORK ON FPGAS

Direct comparisons with prior work are difficult to make due to differences in scheduler microarchitecture, but the approximate comparisons can still demonstrate our improvements. In most cases, to match the chip used in prior work, we re-synthesized our scheduler circuits on a different Altera FPGA than the one our circuits were designed



(a) Distributed scheduler IPC



(b) Distributed, unified, and 1-way scheduler comparison

Fig. 12. IPC comparison between 4-way unified, 4-way distributed, and 1-way unified schedulers.

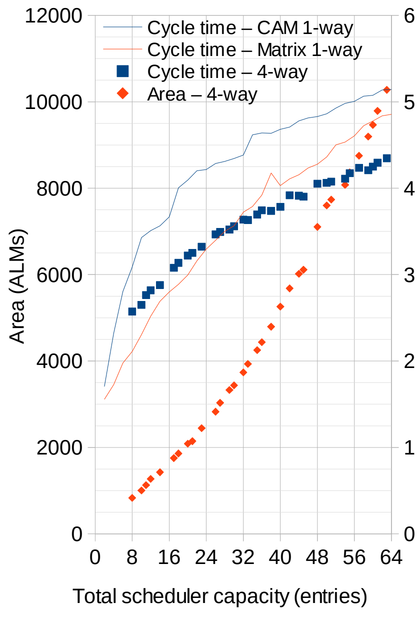
for. Our instruction scheduler circuits achieve faster cycle times than schedulers in the literature, in some cases by substantial amounts.

8.0.1. Single-issue CAM. Aasaraai and Moshovos [Aasaraai and Moshovos 2010] presented a design space exploration of traditional single-issue CAM schedulers on Stratix III FPGAs. The microarchitecture of their scheduler circuits match well with our CAM (single issue, compacting age-priority, two operands) allowing for a reasonably fair comparison. On the same Stratix III FPGA, we achieve higher frequencies with our CAM scheduler circuit (+40% at 16 entries). Matrix and fused-logic matrix schedulers get additional gains (+47% and +60% at 16 entries, respectively).

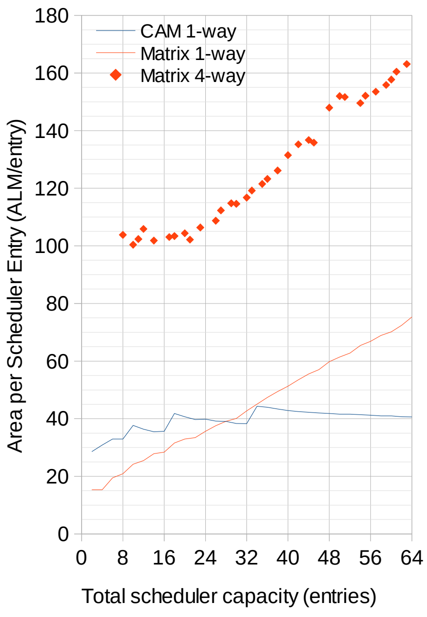
It is interesting that Aasaraai and Moshovos recommend using a small 4-entry scheduler because they observe less than 10% change in IPC on their single-issue out-of-order processor between 2 and 32 entries. We also observed insensitivity of IPC to scheduler size for *single-issue* designs, but *dual-issue* demands a much larger scheduler (Figure 12(b)), which makes the ability to build high-capacity schedulers important.

8.0.2. Dual-issue CAM and Matrix. Johri compared two-issue CAM and matrix schedulers on FPGAs [Johri 2011]. Our single-issue schedulers achieved twice the frequency at 16 entries for both CAM and matrix schedulers, but they use 3 source operands per instruction on a Virtex-6, while we use 2 source operands per instruction on a Stratix IV.

8.0.3. OpenRISC OPA. The OpenRISC OPA out-of-order processor merges the reorder buffer (ROB) and scheduler into a single unit, allowing some circuit simplifications and good f_{max} [Terpstra 2015]. On the same Arria V FPGA, our fused-logic matrix scheduler in isolation achieves about 30% higher frequency than their complete processor at



(a) Area and Delay vs. Capacity



(b) Area per Scheduler Entry

Fig. 13. Area and delay of distributed 4-way compacting matrix scheduler. One-way CAM and matrix (random selection) circuits (from Figures 7 and 8) are also plotted in solid lines for comparison.

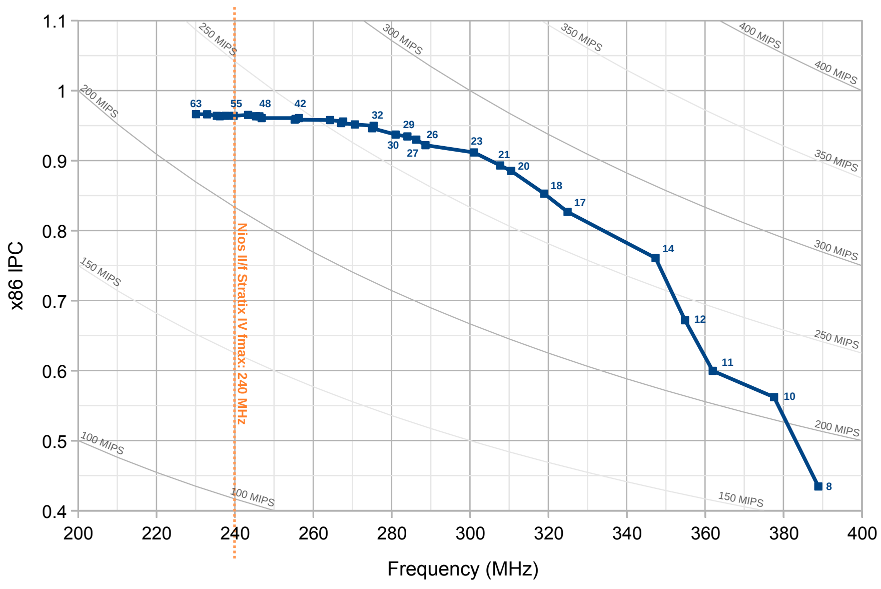


Fig. 14. IPC vs. frequency of 4-way distributed schedulers from 8 to 64 entries. Peak of 275 MIPS occurs at size 20, but MIPS varies little between 14 and 33 entries.

both 18 and 27 entries. Its main drawback is that a merged ROB and scheduler is wasteful of scheduler capacity. Schedulers only need to be 30–50% of the ROB size with almost no loss in IPC, which is also seen in our processor design with 64 ROB entries (Figure 1(a)).

8.0.4. Combined ROB and Scheduler. Rosière et al. presented a combined ROB and scheduler [Rosière et al. 2012]. The microarchitecture appears highly unbalanced, with a large (128–512 entry) ROB, but only the oldest few instructions (4–16) are considered for scheduling. On a Virtex-5, they reported slow f_{max} (4.7$\times$ slower than our fused-logic matrix at 16 entries), and did not report absolute IPC numbers.

8.0.5. Non-Broadcast Scheduler. SEED is a scheduler designed to avoid broadcast behaviour [Mesa-Martínez et al. 2006]. On the same Stratix II FPGA, our fused-logic matrix scheduler achieves 1.9–2.4$\times$ higher f_{max} over their broadcast-free scheduler, and 6.5–5.5$\times$ higher f_{max} over their baseline, an Alpha 21264-like compacting CAM scheduler, for 16 to 64 entries.

9. FUTURE WORK

There has been much processor microarchitecture research that improves on the fundamental scheduler circuits. Most of these proposals still use the same circuit structures at their core, but trade some amount of IPC to improve area, speed, or power [Sassone et al. 2007; Ernst and Austin 2002; Kim and Lipasti 2003; Chen and Hsiao 2007; Palacharla et al. 1997; Canal and González 2000; Michaud and Seznec 2001; Brown et al. 2001; Stark et al. 2000]. The majority of these techniques can still be used on FPGA designs.

While this work aimed for, and achieved, a high speed multi-issue scheduler design, this came with a fairly high area cost. For processors where clock frequency is less critical, whether to reduce area use for less aggressive designs or for more aggressive designs that target higher IPC (and larger schedulers) at lower clock speeds, CAM-based schedulers deserve further exploration for use in multi-issue distributed schedulers. CAM-based schedulers may still be fast enough for a processor with a modest frequency target, potentially at a significant area savings.

10. CONCLUSIONS

We compared optimized circuit structures used in broadcast-based instruction schedulers. We also presented an improved age-based fused-logic matrix circuit that is faster at age-based scheduling than traditional CAM- or matrix-based schedulers ($\sim$20% faster at 22–36 entries, or twice the capacity at 240 MHz), yet is functionally equivalent. Our careful circuit implementations are substantially faster ($\sim$1.4–6$\times$) than prior work.

Our results show that moderately-aggressive out-of-order soft processors with single-issue schedulers of up to 40 entries are feasible on FPGAs at no frequency loss compared to the small, simple, highly-optimized Nios II/f.

For multi-issue processors, we explored the design space of a 4-way distributed scheduler, and demonstrated that a 6-enqueue/4-dequeue distributed scheduler for complex x86 micro-ops runs at a *higher* frequency than a single-issue scheduler for simpler two-operand instructions of the same total capacity, achieving 240 MHz for schedulers with up to 54 entries.

The IPC and performance benefit of out-of-order processors is expected to be large, on the order of 2$\times$ for a first implementation, and opens the door to even more aggressive designs in the future.

REFERENCES

K. Aasaraai and A. Moshovos. 2010. Design space exploration of instruction schedulers for out-of-order soft processors. In *Proc. FPT*.

Altera. 2011. *Stratix IV Device Handbook Volume 1*.

Altera. 2015. *Nios II Performance Benchmarks, DS-N28162004*.

Mary D. Brown, Jared Stark, and Yale N. Patt. 2001. Select-free Instruction Scheduling Logic. In *Proc. MICRO*.

Ramon Canal and Antonio González. 2000. A Low-complexity Issue Logic. In *Proc. Supercomputing*.

Chung-Ho Chen and Kuo-Su Hsiao. 2007. Scalable Dynamic Instruction Scheduler through Wake-Up Spatial Locality. *IEEE Trans. Computers* 56, 11 (Nov 2007), 1534–1548.

D. Ernst and T. Austin. 2002. Efficient dynamic scheduling through tag elimination. In *Proc. ISCA*.

J.A. Farrell and Timothy C. Fischer. 1998. Issue logic for a 600-MHz out-of-order execution microprocessor. *IEEE JSSC* 33, 5 (May 1998), 707–712.

M. Golden, S. Arekapudi, and J. Vinh. 2011. 40-Entry unified out-of-order scheduler and integer execution unit for the AMD Bulldozer x86-64 core. In *Proc. ISSCC*.

Masahiro Goshima, Kengo Nishino, Toshiaki Kitamura, Yasuhiko Nakashima, Shinji Tomita, and Shinichiro Mori. 2001. A High-speed Dynamic Instruction Scheduling Scheme for Superscalar Processors. In *Proc. MICRO*.

M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Workshop on Workload Characterization*. 3–14.

Linley Gwennap. 1997. MIPS R12000 to Hit 300 MHz. *Microprocessor Report* 11, 13 (Oct 1997).

D. Harris. 2003. A taxonomy of parallel prefix networks. In *Proc. Signals, Systems and Computers.*, Vol. 2.

Abhishek Johri. 2011. *Implementation of Instruction Scheduler on FPGA*. Master's thesis. University of Tokyo.

I. Kim and M.H. Lipasti. 2003. Half-price architecture. In *Proc. ISCA*.

Belli Kuttanna. 2013. Technology Insight: Intel Silvermont Microarchitecture. IDF 2013, https://software.intel.com/sites/default/files/managed/bb/2c/02_Intel_Silvermont_Microarchitecture.pdf. (2013).

Kevin P. Lawton. 1996. Bochs: A Portable PC Emulator for Unix/X. *Linux J.* 1996, 29es, Article 7 (sep 1996).

W.L. Lynch, G. Lautterbach, and J.I. Chamdani. 1998. Low load latency through sum-addressed memory (SAM). In *Proc. ISCA*.

Francisco J. Mesa-Martínez, Michael C. Huang, and Jose Renau. 2006. SEED: Scalable, Efficient Enforcement of Dependences. In *Proc. PACT*.

Pierre Michaud and André Seznec. 2001. Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors. In *Proc. HPCA*.

Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. 1997. Complexity-effective Superscalar Processors. In *Proc. ISCA*.

M. Rosière, J.-L. Desbarbieux, N. Drach, and F. Wajsburt. 2012. An out-of-order superscalar processor on FPGA: The ReOrder Buffer design. In *Proc. DATE*.

Peter G. Sassone, Jeff Rupley, II, Edward Brekelbaum, Gabriel H. Loh, and Bryan Black. 2007. Matrix Scheduler Reloaded. In *Proc. ISCA*. 335–346.

Graham Schelle, Jamison Collins, Ethan Schuchman, Perry Wang, Xiang Zou, Gautham Chinya, Ralf Plate, Thorsten Mattner, Franz Olbrich, Per Hammarlund, Ronak Singhal, Jim Brayton, Sebastian Steibl, and Hong Wang. 2010. Intel Nehalem Processor Core Made FPGA Synthesizable. In *Proc. FPGA*. 3–12.

SPEC. 2000. SPEC CPU95 Results. https://www.spec.org/cpu95/results/. (2000).

Jared Stark, Mary D. Brown, and Yale N. Patt. 2000. On Pipelining Dynamic Instruction Scheduling Logic. In *Proc. MICRO*.

Wesley Terpstra. 2015. OPA: Out-of-Order Superscalar Soft CPU. In *ORCONF*.

S. Vangal, M.A. Anders, N. Borkar, E. Seligman, V. Govindarajulu, V. Erraguntla, H. Wilson, A. Pangal, V. Veeramachaneni, J.W. Tschanz, Yibin Ye, D. Somasekhar, B.A. Bloechel, G.E. Dermer, R.K. Krishnamurthy, K. Soumyanath, S. Mathew, S.G. Narendra, M.R. Stan, S. Thompson, V. De, and S. Borkar. 2002. 5-GHz 32-bit integer execution core in 130-nm dual-VT CMOS. *IEEE JSSC* 37, 11 (Nov 2002), 1421–1432.

H. Wong, V. Betz, and J. Rose. 2016. High Performance Instruction Scheduling Circuits for Out-of-Order Soft Processors. In *Proc. FCCM*. 9–16.