HAIR: Halving the Area of the Integer Register File with Odd/Even Banking

Pierre Michaud, Anis Peysieux

HAIR: Halving the Area of the Integer Register File with Odd/Even Banking

Pierre Michaud and Anis Peysieux

Inria, Univ Rennes, CNRS, IRISA

{pierre.michaud,anis.peysieux}@inria.fr

July 5, 2022

Abstract

This paper proposes a new microarchitectural scheme for reducing the hardware complexity of the integer register file of a superscalar processor. The register file is split into two banks holding even-numbered and odd-numbered physical registers, respectively. Each bank provides one read port to each two-input integer execution unit. This way, each bank has half the total number of read ports, and the register file area is roughly halved, which reduces the energy dissipated per register access and the register access time. However, a bank conflict occurs when both inputs of a two-input micro-operation lie in the same bank. Bank conflicts hurt performance and we propose a simple solution to remove most bank conflicts, thus recovering most of the lost performance.

1 Introduction

Since the mid 2000s, CPU performance has kept increasing while the thermal design power (TDP), that is, the maximum power that the CPU can dissipate in a sustained manner, remained limited by the cooling capacity [27]. Twenty years ago, it was possible to sacrifice energy efficiency for maximizing performance. Ever since however, energy efficiency has been a necessary condition for high performance, even for desktop and server CPUs. This fact is manifest in the substantial gap existing in today's CPUs between the TDP and the maximum instantaneous power, making the base clock frequency significantly lower than the maximum turbo frequency.

The energy efficiency problem can been tackled at several levels, one of them being the microarchitecture [8, 23, 73]. Energy and execution time tend to be anticorrelated quantities: microarchitecture modifications that reduce one of these quantities often increase the other. A modification of the microarchitecture is not considered to improve energy efficiency unless the energy vs. performance tradeoff it offers is better than that provided by voltage/frequency scaling [38, 76]. This limits the set of microarchitecture features that can be implemented in practice, particularly when performance is limited by the TDP. As can be seen from the following relation between power, energy per instruction (EPI) and instructions executed per second (IPS),

$$\text{power} = \text{EPI} \times \text{IPS},$$

if performance is constrained by a fixed power limit, the IPS cannot increase unless the EPI is reduced. Today, net performance gains may be more easily obtained by simplifying the microarchitecture than

by making it more complex. An example of microarchitectural simplification is the presence of "small" cores in heterogeneous multicore chips [32, 25, 55].

The HAIR scheme proposed in this paper is a simplification of the integer register file (IRF) intended to reduce the energy dissipated in the IRF. The CPU-wide energy reduction provided by HAIR is modest, as the IRF represents only a fraction of the whole CPU energy.[1] Nevertheless, temperature on the chip is not uniform, and it is more rewarding, temperature-wise, to remove one watt from a hot region of the chip than from a cold region. The IRF is typically a hot spot [62, 14]. When only a few cores are active, we expect the relative-to-ambient IRF temperature to be reduced more, in proportion, than the total chip power.

HAIR splits the IRF into two banks holding even-numbered and odd-numbered integer physical registers, respectively. Each bank provides one read port to each ALU (arithmetic logic unit) and each AGU (address generation unit). This way, each bank has half the total number of read ports, which roughly halves the total IRF area and reduces substantially the energy per IRF access and the IRF access time. This simplification comes at the cost of a loss of IPC (average number of instructions executed per clock cycle) due to bank conflicts occurring when both source operands of a 2-input micro-operation (μop) lie in the same bank, as such μop needs one extra cycle to read both operands. We propose a simple solution to remove most bank conflicts, thus recovering most of the lost IPC.

The paper is organized as follows. Section 2 describes the baseline microarchitecture model and the simulation methodology. Section 3 describes the proposed HAIR banking scheme. We show that bank conflicts hurt performance on some workloads. We propose a simple solution to the bank conflict problem in Section 4 and we show that this solution recovers most of the lost performance. We also provide an analytical model to explain how and why the proposed solution works. Finally, the related work is discussed in Section 5.

2 Simulated baseline microarchitecture model

For this study, we used an in-house trace-driven simulator. Traces are generated with Intel's Pin dynamic binary instrumentation framework [37]. The traces are listed in Table 1. Each trace was obtained by running the SPECrate 2017 benchmarks with the reference inputs [11]. Benchmarks were compiled for an Intel Cascade Lake microarchitecture with `gcc 9.3` using the `-O3` optimization level. Each trace consists of 20 samples stitched together, each sample representing 50 million consecutive dynamic instructions, for a total of 1 billion instructions per trace. The 20 samples are taken uniformly throughout the whole benchmark execution. The simulated baseline microarchitecture is described in Table 2. The superscalar core is loosely modeled after the AMD Zen 3 [2]. Only a single core is simulated, with a scaled down L3 cache. Table 1 provides the baseline IPC[2] for each trace. All the performance numbers presented in this paper are normalized by dividing the IPC for a given trace by the baseline IPC from Table 1.

2.1 Simulator approximations

Our simulator does not simulate operating system activity. It does not simulate wrong-path execution either. Front-end pipeline stages are somewhat idealized. In particular, the branch target buffer (BTB) and the μop-cache are not simulated. Move elimination (AKA zero-latency move) is not simulated. We simulate only 20 distinct μop categories. Most dynamic μops executing on the integer execution unit

[1] Improving energy efficiency in a substantial manner at the microarchitecture level requires modifying many different parts of the microarchitecture, as no single part is responsible for more than a fraction of the whole CPU energy. Hence, apart from straightforward downscaling of the main parameters of the microarchitecture, microarchitectural solutions for reducing energy are generally incremental.

[2] Each iteration of a REP prefixed string instruction is counted as a distinct instruction.

trace	benchmark	input	IPC	%int	%two	trace	benchmark	IPC	%int	%two
500a	perlbench	checkspam.pl	2.96	89.5	7.2	505	mcf	0.92	89.3	13.8
500b	perlbench	diffmail.pl	3.54	89.6	5.4	507	cactuBSSN	1.43	46.4	24.9
500c	perlbench	splitmail.pl	2.50	90.2	4.0	508	namd	3.31	50.9	20.9
502a	gcc	gcc-pp.c -O3	1.78	89.1	4.6	510	parest	1.81	73.3	21.3
502b	gcc	gcc-pp.c -O2	1.36	88.8	5.2	511	povray	3.10	72.7	2.8
502c	gcc	gcc-smaller.c -O3	1.36	87.4	7.3	519	lbm	0.83	26.0	0.8
502d	gcc	ref32.c -O5	1.00	91.7	4.4	520	omnetpp	0.78	88.8	6.4
502e	gcc	ref32.c -O3	0.95	97.8	0.8	521	wrf	1.47	54.1	28.3
503a	bwaves	bwaves_1.in	1.10	55.4	45.1	523	xalancbmk	1.24	82.2	17.3
503b	bwaves	bwaves_2.in	1.07	56.3	48.3	526	blender	2.41	59.6	12.6
503c	bwaves	bwaves_3.in	1.05	56.8	50.1	527	cam4	2.16	63.2	19.1
503d	bwaves	bwaves_4.in	1.04	55.1	44.5	531	deepsjeng	2.46	91.9	14.8
525a	x264	--pass 1 --frames 1000	3.61	93.9	31.4	538	imagick	4.16	67.8	7.3
525b	x264	--pass 2 --frames 1000	4.09	92.9	34.2	541	leela	1.99	90.6	20.3
525c	x264	--seek 500 --frames 1250	3.99	93.5	33.5	544	nab	2.25	55.1	18.8
557a	xz	cld.tar.xz	0.85	90.9	24.6	548	exchange2	4.21	89.5	8.7
557b	xz	cpu2006docs.tar.xz	2.13	86.3	37.5	549	fotonik3d	0.48	51.4	40.0
557c	xz	input.combined.xz	1.70	88.5	31.2	554	roms	0.83	55.1	41.5

Table 1: Our 36 traces. Benchmarks for which no input is specified (right side) use the reference inputs. For each trace, the IPC of the baseline microarchitecture (Table 2) is provided, followed by the percentage of μops that execute on the integer execution unit (%int) and the percentage of integer μops that have two inputs (%two). A μop reading the same register twice is not counted as a 2-input μop.

general	Intel x86-64, fixed clock, single core, no SMT, OoO issue, 64-byte cache line, 4 KB page
branch prediction	67 KB TAGE-GSC [58]; 6 KB ITTAGE [57]; ideal BTB; branch mispredict penalty 12 cycles (minimum); redirect instruction fetch at branch execution
front-end	I-fetch 1 line, 1 taken branch/cycle; 32 KB instruction cache; 6-entry FTQ [53]; decode: 8 instructions/cycle; rename: 12 μops/cycle; retire: 12 μops/cycle
X-pipes	IX: 4 ALU, 3 AGU, 2 STD; FPX: 4 FP, 2 STD
μops window	ROB 512 μops; LDQ 128 loads; STQ 64 stores; IX scheduler 96 μops; FPX scheduler 64 μops; FPX NSQ (non-scheduling queue) 64 μops[2]
registers	160 int physical registers (16 reads, 7 writes); 160 FP physical registers (12 reads, 7 writes)
store sets [10]	SSIT 2048 entries, 4-way associative
TLB	ITLB1: 64 entries 4-way associative; DTLB1: 64 entries 16-way associative; TLB2: 2048 entries 16-way associative; 6 page walks
L1 data cache	32 KB, 8-way associative, LRU, write back, 8 banks, 8 bytes/bank, 3 accesses/cycle (2 stores max), int latency 3 cycles, FP latency 5 cycles; MSHR 32 lines; stride prefetcher 64 entries
L2 cache	512 KB, 8-way associative, LRU, write back, latency 8 cycles; BOP [42]
L3 cache	4 MB, 16-way associative, DIP [52], write back, latency 34 cycles
DRAM	2 channels; each channel: 32-read queue, 32-write queue, 64-bit bus, 1 rank, 8 chips/rank, 8 banks/chip, row buffer 1KB/chip; FR-FCFS [54]; bus cycle = 4 core cycles; timing (bus cycles): tCL=11, tRCD=11, tRP=11, tRAS=33, tCWL=8, tRTP=6, tWR=12, tWTR=6, tBURST=4 (8 beats)

Table 2: Simulated baseline core microarchitecture

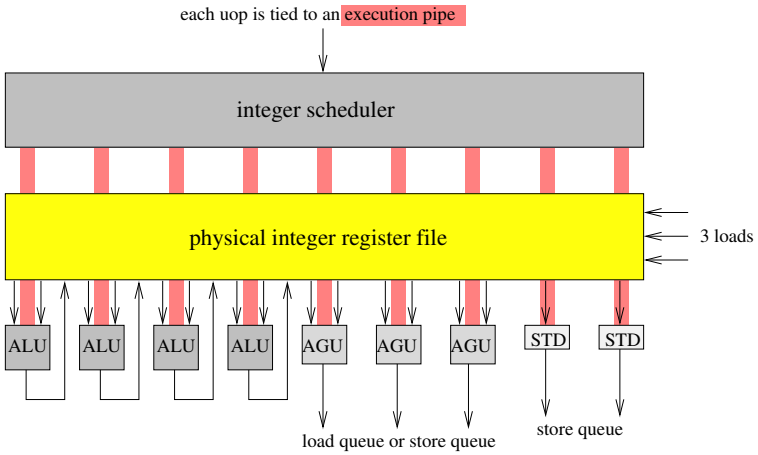

Figure 1: The integer execution unit (IX) modeled in our simulator. Each μop is tied to an X-pipe before entering the scheduler buffer. An X-pipe consists of an execution unit, some dedicated register read ports and write ports, and a picker that selects from the scheduler buffer the oldest ready-to-execute μop attached to this X-pipe. The IX features 9 X-pipes. The integer register file has 16 read ports and 7 write ports. The bypass network sending the outputs of the execution units back to the inputs is not shown.

(IX) have a 1-cycle latency. Most dynamic μops executing on the floating-point execution unit (FPX) have a 4-cycle latency. Distinct μops are generated for all address computations, including loads. Our Pin-based tracing tool (Pin 3.18) does not trace scattered vector memory accesses, and the corresponding instructions are treated like NOP instructions. The only benchmark with a non-negligible amount of scattered vector accesses is 549.fotonik3d, where these instructions represent about 2.6% of all dynamic instructions. We assume that flags and AVX-512 mask registers have dedicated physical register files, which we do not simulate. Nevertheless, we do simulate the performance effect of data dependencies through the flags and mask registers. We simulate the *macro-fusion* of a conditional branch instruction with the immediately preceding instruction when this instruction updates the flags [28]. However, we do not simulate AMD Zen's macro-ops [2], nor Intel's *micro-fusion* [23, 28]. To compensate for this approximation (plus our simulator considering address computations as distinct μops), we simulate a 12-μop wide rename stage, a 12-μop wide retire stage, and a 512-μop reorder buffer (ROB) [2]. We do not simulate the effect on performance of the μops issued speculatively in the shadow of a load that misses the L1 data cache but was predicted to hit [50]. Nevertheless, we simulate data-cache bandwidth contention from load misses. We simulate virtual-to-physical address translation by applying a randomizing hash function on the virtual page number. Page table walks are approximated: upon a TLB2 miss, we assume perfect caching for intermediate levels of the page table [5], and we "magically" access only the last level, i.e., the page table entry (PTE). We read the PTE through the L2 cache (an L2 miss brings several adjacent PTEs at once).

2.2 The integer execution unit (IX)

This study focuses on the integer execution unit (IX), which executes μops doing operations on integer registers. Integer registers are general-purpose, scalar registers. They are distinct from floating-point (FP) vector registers, which are located in the floating-point execution unit (FPX). All the address computations are done by the IX.

Figure 1 depicts the IX modeled in our simulator. The IX is somewhat similar to that of the AMD Zen 3 microarchitecture [2]. As in most modern superscalar CPUs, the IX is structured as multiple execution pipes (X-pipes for short, AKA execution ports) for simplifying the scheduler. An X-pipe consists of an execution unit (e.g., an ALU), some dedicated register read ports and write ports, and

a *picker* that selects from the scheduler buffer the oldest ready-to-execute μop attached to this X-pipe. All the X-pipes in the IX share a single scheduler buffer.[3] Before entering the scheduler buffer, μops are tied to an X-pipe by the steering stage. The four X-pipes denoted "ALU" in Figure 1 each contain an ALU. However, they are not all identical. As in most modern superscalar CPUs, the X-pipes are specialized. In our simulated model, two ALU X-pipes can execute branches, the other two can execute long operations such as multiplication, division, population count, etc. The steering stage must try to equalize the utilization of X-ports. A simple round-robin policy cannot be used here, because of X-pipe specialization. Instead, when a μop can be executed by several X-pipes, it is steered to the X-pipe least-recently steered to.[4]

The three AGU X-pipes are for address computations and have no register write ports. The two STD X-pipes are for store-data μops and have no register write ports either. Unlike other X-pipes, the STD X-pipes have a single register read port. As in the Zen 3, up to three loads can execute simultaneously, hence there are three register write ports for loads. Overall, the IRF has 16 read ports and 7 write ports (16R7W).

3 The HAIR odd/even banking scheme

The IRF dissipates a significant amount of power, for multiple reasons. The IRF is read and written very frequently, every clock cycle for certain workloads. Moreover, each X-pipe has dedicated register file ports. Superscalar cores have multiple X-pipes to exploit instruction-level parallelism (ILP). However, the register file area increases quadratically with the number of ports [26]. Power dissipation increases with the number of ports not only due to more simultaneous accesses, but also because of the larger array which makes bitlines longer, increasing the energy per access. Furthermore, the instruction window of superscalar processors keeps growing for exploiting more ILP and supporting more simultaneous threads. This requires more physical registers, which increases the register file area and power. Also, there is an indirect energy cost induced by the IRF. Accessing a large multi-ported register file may take several clock cycles. However, pipeline stages between issue and execute are detrimental to performance, as they contribute not only to the branch misprediction penalty but also to the load misscheduling penalty [7, 50]. Moreover, the bypass network complexity increases with the number of register access stages, and the effective scheduler buffer capacity is reduced as μops speculatively issued after a load stay in the scheduler until the load is known to be a cache hit [7]. Hence, while pipelining may allow to use slower, more energy efficient logic in certain parts of the microarchitecture, there is not much room in the IRF for such tradeoff.

The proposed HAIR scheme reduces the IRF area. Reducing the area shortens the wordlines and bitlines, thereby reducing access time and dynamic energy per access. Another way to reduce the IRF area is to reduce the number P of integer physical registers. However, this hurts performance on certain workloads, as can be seen in Figure 2, which shows normalized performance when P varies. Instead, we propose to reduce the IRF area by halving the number of read ports.

3.1 The HAIR register file

The HAIR IRF is depicted in Figure 3. The IRF is split into two identical banks: bank 0 contains even-numbered physical registers, bank 1 contains odd-numbered physical registers. Each ALU or AGU reads one input from bank 0, the other input from bank 1. ALUs and AGUs contain multiplexers to permute the left and right inputs for non-commutative operators (division, shift,...). One STD X-pipe is

[3] Some microarchitectures, such as the AMD Zen family, partition the scheduler buffer statically so that each partition is tied to a specific subset of the X-pipes. Our proposition is compatible with all scheduler types.

[4] To the best of our knowledge, the steering policy implemented in commercial CPUs has not been disclosed so far.

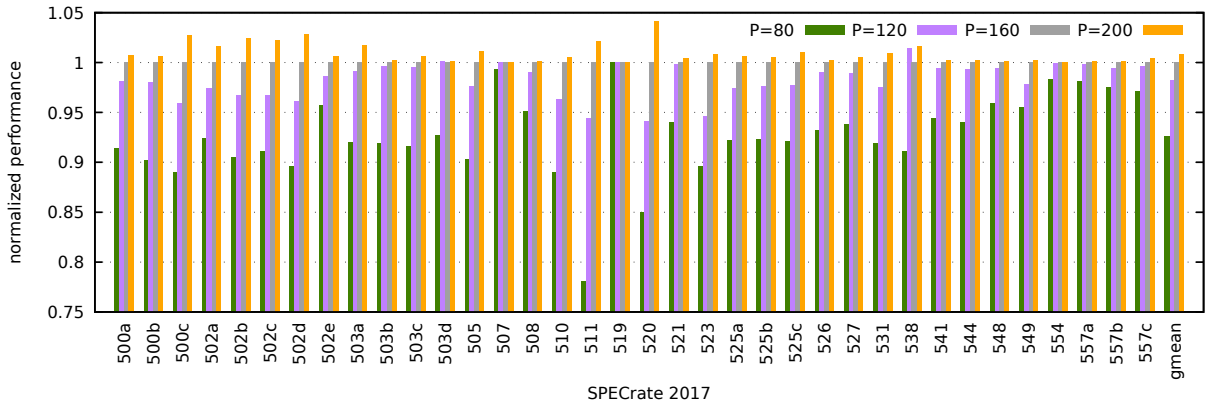

Figure 2: Normalized performance of the baseline microarchitecture when the number P of integer physical registers is varied from 80 to 200. The rightmost group of bars is a weighted geometric mean, where the weight for a trace is inversely proportional to the number of traces for that benchmark (for instance, trace 500a has weight $1/3$ because there are 3 traces for benchmark 500.perlbench).

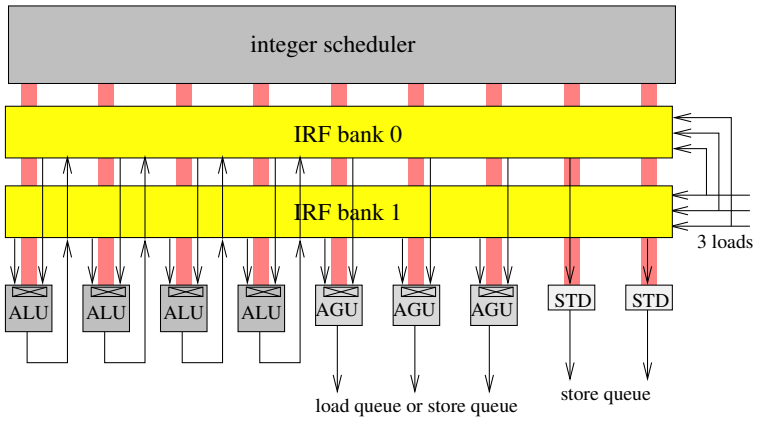

Figure 3: The HAIR IRF is split into two identical banks: banks 0 and 1 contain even-numbered and odd-numbered physical registers respectively. Each ALU or AGU X-pipe reads one input from bank 0 and the other input from bank 1. One STD X-pipe reads bank 0, the other STD reads bank 1. ALU X-pipes can write in either bank. Each bank has 8 read ports and 7 write ports.

	SRAM	IRF area	access time	read energy	write energy
baseline	160-entry, 16R7W	0.098 mm^2	0.274 ns	3.61 pJ	4.93 pJ
HAIR	two 80-entry 8R7W	0.054 mm^2	0.165 ns	1.44 pJ	2.61 pJ
ratio		0.55	0.60	0.40	0.53

Table 3: Area, delay and energy of the IRF as given by CACTI-P (22nm, ITRS HP, 0.8V, no ECC, minimize energy-delay product)

specialized to read bank 0, the other STD reads bank 1. ALU X-pipes can write in either bank. Each bank has 8 read ports and 7 write ports.

For a register file with R reads ports and W write ports (not double-pumped), the bit cell area is roughly proportional to $(3 + R + W)^2$, assuming single-ended bitlines [67]. The SRAM array area is therefore proportional to $P \times (3 + R + W)^2$ where P is the number of physical integer registers. The area of one bank, normalized to the baseline IRF area, is $\frac{(P/2)(3+8+7)^2}{P(3+16+7)^2} \simeq 0.24$. So the total IRF area is roughly half the baseline IRF area.

Multi-ported register files often use double-pumping and/or replication [51, 26, 13, 39, 47]. For instance, if the baseline IRF uses two P-entry 8R7W replicas, we can implement each bank of the HAIR IRF as a P/2-entry 8R7W array, and the total area reduction is $\frac{2(P/2)(3+8+7)^2}{2P(3+8+7)^2} = 0.5$. If, instead, the baseline uses a 8R4W double-pumped IRF, we can use double pumping on each bank of the HAIR IRF and the total area reduction is $\frac{2(P/2)(3+4+4)^2}{P(3+8+4)^2} \simeq 0.54$.

The area reduction should coincide with a reduction of access time and dynamic energy. We modeled multi-ported SRAMs with CACTI-P[5] [35], which was released with the McPAT framework [34]. Table 3 provides the area, delay and energy of the IRF as given by CACTI-P: HAIR makes the IRF access time 40% shorter,[6] the read energy and write energy 60% and 47% smaller respectively. Note that the CACTI energy numbers ignore the potential energy reduction that could be obtained if the reduced delay allowed to remove some pipeline stages.[7]

3.2 Sequential register access and the Half-Price architecture

While μops reading a single register are not affected by odd/even banking, there is a problem for μops reading two registers when these registers are mapped in the same bank. This problem can be solved with *sequential register access* (SRA), a method introduced by Balasubramonian et al. [4] and used by Kim and Lipasti in their Half-Price architecture (HPA) [30]. HAIR is inspired from HPA. However, HPA does not use odd/even banking. Instead, it provides a single read port per X-pipe. That is, instead of having 16 read ports as in Figure 1, the IRF in HPA would have 9 read ports.

SRA works as follows in HPA. When an X-pipe selects a 2-input[8] μop for execution, it uses two consecutive clock cycles to issue the μop. In the first cycle, a pseudo-move μop is issued by the X-pipe to read the first input operand, which is stored in the output latch of the ALU/AGU. In the second cycle, the 2-input μop is issued, accesses the register file for the second input operand, and catches the first operand, read a cycle earlier, from the local bypass. Scheduling-wise, this is almost identical to issuing a non-pipelined operation and adds very little complexity to the scheduler.

[5] We corrected a bug in CACTI which made the number of single-ended read ports stuck at zero, ignoring the value from the configuration file. We also modified the cost function for energy-delay product optimization so that it takes into account write energy besides read energy.

[6] The extra multiplexer shown in Figure 3 as part of the ALU/AGU may impact the clock cycle. If so, we can take advantage of the reduced IRF access time and put the multiplexer in the register read stage instead.

[7] The simulation results presented in this study assume that HAIR does not change the pipeline length.

[8] In HAIR, like in HPA [30], a 2-input μop that is not a zero idiom and reads the same register twice is considered a single-input μop. The multiplexers in the ALUs and AGUs for permuting the left and right inputs (Figure 3) can also duplicate an input.

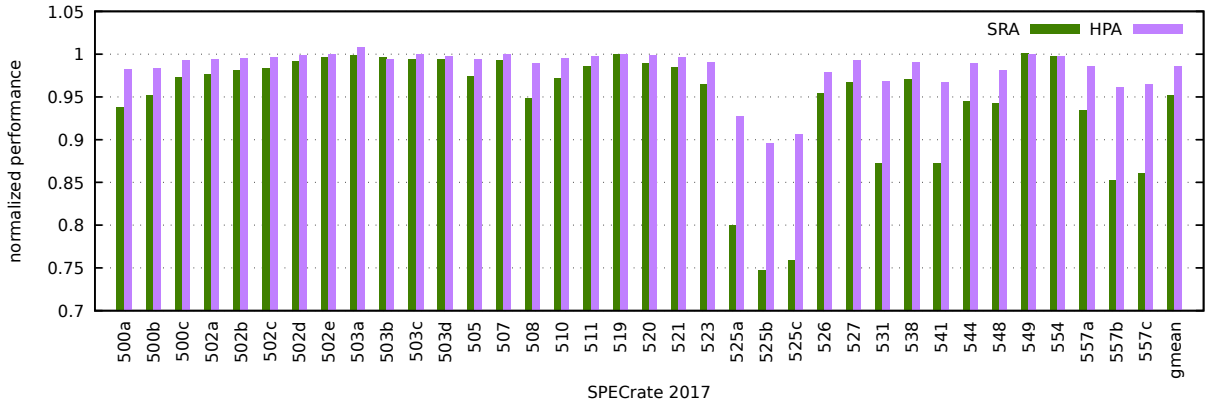

Figure 4: Performance of HPA [30], with ("HPA") and without ("SRA") the one-cycle bypass window.

```
0  LOOP:
1  add rdx, r9
2  add rbp, r8
3  add r12, rsi
4  add r13, rdi
5  add rbx, rcx
6  add r14, r10
7  add r15, rax
8  cmp rdx, 99999999
9  jle LOOP
```

Listing 1: Loop devised to hurt HPA's performance (Intel x86-64)

There is a performance penalty from issuing one extra μop (the pseudo-move) and adding one cycle to the original μop latency. While only 2-input μops are concerned by this penalty, these μops are not rare and SRA may hurt performance. To mitigate the problem, HPA exploits the fact that a μop is often selected for issue immediately after its source operands are ready. HPA modifies the scheduler so that a 2-input μop can issue without SRA if it is selected for issue during the one-cycle *bypass window* which is the first cycle when both source operands are ready. In this case, one of the two operands can be caught on the fly from the bypass network, and the register read port provides the other operand [30]. However, if a 2-input μop is not selected for issue during the one-cycle bypass window (because it is not the oldest ready μop on that X-pipe at that time, or because it has been rescheduled), it is issued with SRA.

Figure 4 shows the performance of HPA[9] with and without the one-cycle bypass window. Disabling the bypass window hurts HPA's performance, with about 5% average IPC loss compared to the baseline, and up to 25% IPC loss on benchmark 525.x264. The one-cycle bypass window reduces the IPC loss, which is only 1.4% on average, but up to 10.5% on 525.x264.

HPA was published 18 years ago and, to the best of our knowledge, has not been implemented in a real processor yet. Neither AMD nor Intel mention anything resembling HPA in their optimization guides [2, 28]. We believe that, had HPA been implemented as described in [30], it would have been documented, as its effect on performance is noticeable on certain workloads.

Listing 1 shows a loop devised to hurt HPA's performance. This loop executes with an IPC of 4.5 on our simulated baseline[10] and with an IPC of 2.25 on the simulated HPA. That is, HPA halves the IPC. We also ran this loop on an Intel Haswell processor and on an Intel Cascade Lake: on both

[9]The HPA paper also proposes a scheduler scheme. We simulated only the HPA register scheme.
[10]The conditional branch is macro-fused with the CMP.

microarchitectures, we measured an IPC of 4.5 from the performance counters, which is identical to our simulated baseline. All the Intel microarchitectures from Haswell (2013) to Rocket Lake/Cypress Cove (2021) have 4 ALUs and an IPC that should exceed 4 on this loop [1].

It is not always clear why a published idea is not implemented. There may be several possible reasons. For HPA, we can only guess what these reasons might be:

- The performance loss from HPA is non negligible on certain workloads (e.g., 525.x264).
- Implementing the bypass window may require modifying the scheduler in a non straightforward way. Kim and Lipasti described an implementation for CAM-based wakeup [30]. They did not consider matrix-based wakeup [75, 40, 56], which was implemented in some processors [21, 73, 61].

For HAIR, our goal is to minimize the performance loss from banking the register file, but with minimal modifications to the scheduler, which is one of the timing-critical parts of the microarchitecture [39]. We use SRA because it is simple to implement. After register renaming, we know the source physical registers of any 2-input μop. If the two registers have the same parity, this is a bank conflict, which requires SRA. Therefore, whether a μop needs SRA is known when inserting the μop into the scheduler buffer.

3.3 Pseudo-random parity and register renaming

As shown in Figure 4, the performance loss is quite substantial if SRA is used on every 2-input integer μop. However, with HAIR banking, if integer registers are randomly mapped to the odd or even bank, there is one chance in two that a 2-input μop experiences no bank conflict. In a conventional register renaming scheme, free physical registers are provided by a *free list*, which can be implemented explicitly as a queue [72] or implicitly with a free-register generator if CAM-based renaming is used [33, 9]. With a queue-based free list, the physical register numbers output by the free list should be random enough to halve bank conflicts.

If we want a non-random physical register allocation, such as alternating between the odd and even banks, we must modify register renaming and have separate free lists for even-numbered and odd-numbered physical registers. Seznec et al. described such renaming scheme [59], which we call *dual free list*. With a dual free list, a pseudo-random *parity bit* is determined for each μop writing an integer register. The parity bit tells whether to take the destination physical register from the odd or even free list. The parity bit is obtained before the renaming stage, according to a simple algorithm such as random or alternation. Therefore, it is possible to allocate free physical registers to the μops before the renaming stage [59].

To keep parity assignment simple, we stall the renaming stage when one of the two free lists is empty. If the utilization of the odd and even free lists is approximately balanced, when a free list becomes empty, the other free list should be close to empty as well.[11]

Figure 5 shows the performance of the HAIR banking scheme with a single free list (i.e., conventional renaming) and with a dual free list when parity is chosen either randomly or by alternating between odd and even. For random parity, the single free list is slightly better than the dual free list in general. First, this shows that the conventional free list is a good enough random number generator for halving

[11] Occasional unbalance is unavoidable, as we control register allocation but not register release. Statistical unbalance with random parity can be modeled by flipping a coin multiple times, counting heads and tails, stopping when either the number of heads or the number of tails equals $P/2$, with P the number of integer physical registers. The absolute value of the difference between the numbers of heads and tails equals the number of wasted registers. On average, this is approximately twice the mean absolute deviation (MAD) of a binomial distribution $B(P, 1/2)$, assuming a large P. For a large P, the MAD is approximately $\sqrt{2/\pi}$ times the standard deviation [6]. The variance of $B(P, 1/2)$ is $P/4$, hence the average number of wasted registers is approximately $\sqrt{2P/\pi}$. For instance, with $P = 160$, about 10 physical registers are wasted on average.

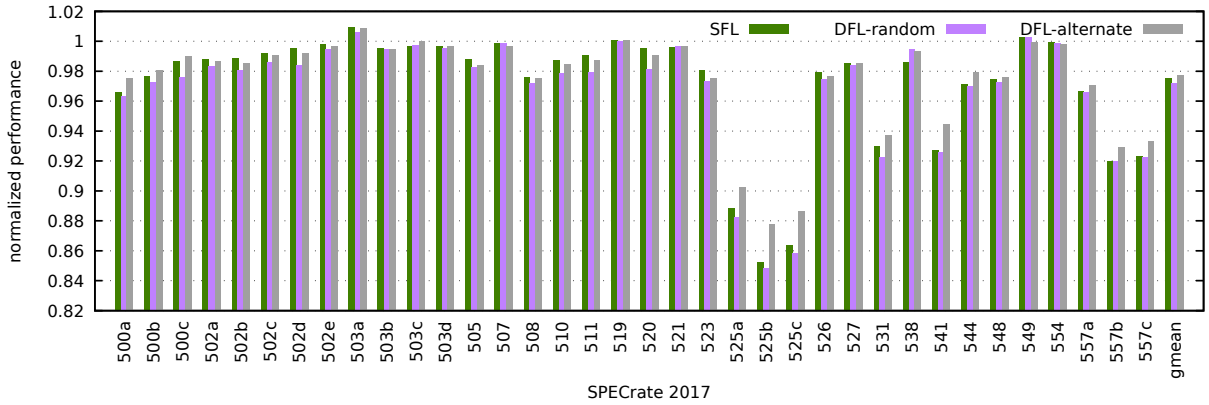

Figure 5: Performance of HAIR banking with a single free list (SFL) and with a dual free list (DFL) and random or alternating parity.

bank conflicts. Second, this shows that, with a dual free list, stalling the rename stage when one of the two free lists is empty causes little performance loss. Figure 5 also shows that the IPC loss from bank conflicts can be quite significant on certain workloads (up to 15% on 525.x264).

With alternating parity, the fraction of 2-input integer μops experiencing a bank conflict is less than 50% in general (the median is 43.5% on the 36 traces), because parity alternation takes advantage of frequent patterns, in particular when a μop reads the registers written by the two previous μops. As a result, alternation reduces the average IPC loss. Still, the IPC loss is quite significant on certain workloads (up to 12% on 525.x264). Pseudo-random parity alone is not sufficient for odd/even banking to be viable: we must find a way to reduce the number of bank conflicts.

4 Removing bank conflicts with a parity table

To approach the baseline performance, we must find a way to remove most bank conflicts without introducing complex hardware.

It should be noted that not all conflicts can be removed. To understand why, consider the *coupling graph* $\mathcal{G}$ defined as follows: $\mathcal{G}$ is the undirected graph whose vertices are static instructions and such that an edge connects two instructions X and Y if there exists a dynamic 2-input μop using the results of X and Y. If this is the case, the two vertices are adjacent in the graph and instructions X and Y are said to be coupled. Instructions that do not feed a 2-input μop have no vertex in $\mathcal{G}$.

Removing all the bank conflicts is equivalent to coloring the vertices of $\mathcal{G}$ with two colors in such a way that two adjacent vertices never have the same color. This is possible only for *bipartite* graphs [68]. Obviously, not all graphs are bipartite (e.g., consider 3 vertices forming a triangle). The solution we propose is based on the following assumptions:

- the coupling graph $\mathcal{G}$ is bipartite or nearly bipartite;
- when a bank conflict is detected at rename for a 2-input μop, the instructions producing the inputs are often still in the ROB.

4.1 Parity table

The idea is to introduce a *parity table* providing a parity bit to each instruction so that, when bank conflicts happen, we can correct the parity bits and remove conflicts. We assume the dual free list register renaming scheme described in Section 3.3.

Each entry of the parity table holds a single parity bit. The table is accessed with a simple PC-based hash function such as (in C language)

```
index = (PC ^ (PC>>L)) % (1<<L)
```

where PC (program counter) is the address of the instruction and 2^L is the number of table entries. Parity bits are read in parallel with the instruction cache access. The hash function above, geared to the x86 instruction-set architecture (ISA), makes it possible to read a *bundle* of 64 parity bits for all consecutive PCs in a 64-byte line at once. Then, once the instruction boundaries are known (at decode), the needed parity bits are extracted from the bundle. For a fixed-length ISA such as ARMv8, we would need to read a bundle of 16 parity bits per 64-byte cache line.

As a workload executes, bank conflicts happen and the parity table is progressively modified to try to remove conflicts, as we explain below in section 4.2. After some time, the parity table likely contains ones and zeros in similar proportions, and the utilization of the odd and even free lists should be approximately balanced. However, if we start from a parity table containing only zeros (e.g., just after powering the processor up), and if the workload contains only a few 2-input μops, it may take some time for the parity table to reach a balanced state. To avoid this (rare) situation, the parity table is initialized with an equal number of zeros and ones.

The parity table scheme should behave like the pseudo-random parity assignment policy of Section 3.3 as far as free list balancing is concerned, with one difference though: as the parity table is accessed with the PC, all the μops in the same instruction get the same parity bit. This may lead to unbalanced utilization of the odd and even free lists when executing a tight loop. For better balance, we alternate the parity of IRF-writing μops from the same instruction. That is, the first μop that writes the IRF gets the parity p read from the parity table. If the instruction contains a second μop writing the IRF, this μop gets the opposite parity $\overline{p}$. If the instruction contains a third μop writing the IRF, this μop gets parity p, and so on.

4.2 Updating the parity table

The parity table is updated as follows. We introduce a *wait* bit, an 8-bit Conflict Physical Register Number (CPRN) and a 9-bit ROB Pointer (RP). At rename, if a bank conflict is detected for a 2-input μop (called *conflicting* μop) and if the wait bit is not set, one of the two inputs is randomly chosen.[12] Then, CPRN is set to the physical register of the chosen input, the wait bit is set, and RP points to the ROB entry of the conflicting μop. At retirement, if the wait bit is set, CPRN is compared with the destination physical registers of the μops being retired. If CPRN matches the destination register of a retiring μop, the parity bit corresponding to the retiring instruction is flipped[13] and the wait bit is reset.

When a physical register number is recorded in CPRN, there is no guarantee that the μop writing the register is still in the ROB. We need a timeout mechanism to prevent the wait bit from being blocked forever: the wait bit is reset when the conflicting μop retires, that is, when the ROB head pointer equals RP.

It should be stressed that an instruction whose result is not read by any 2-input μop never updates the parity table. For such instruction, the parity table serves as a random bit generator.

[12] We generate a pseudo-random bit by XORing a bit from the left input physical register number with a bit from the right input physical register number (as this is a bank conflict, the least-significant bit of the register number must not be used here, as this bit is the same for both inputs).

[13] The old parity bit does not need to be read again from the parity table. It can be recovered from the least-significant bit of CPRN and from the μop's position in the retiring instruction.

4.3 Branch mispredictions

The wait bit and the CPRN and RP registers are set when detecting a bank conflict, at rename. It is possible that an earlier, not yet executed branch was mispredicted and, consequently, that instruction fetch is following a wrong control-flow path. If the instruction writing CPRN is anterior to the mispredicted branch and still in the ROB, this instruction will generate a *wrong-path* parity table update when retiring.

Our trace-driven simulator does not model wrong-path updates. Nevertheless, we believe that wrong-path updates are not a problem, for the following reason. Wrong-path updates might be a performance problem if they add spurious edges to the coupling graph $\mathcal{G}$ in such a way that the graph becomes non-bipartite because of these extra edges. However, wrong-path updates do not necessarily lead to spurious edges. The branch predictor is trained to follow control-flow paths that have already been taken in the past and that are likely to be taken again in the future. The most likely effect of a wrong-path update is that it prevents a future bank conflict.

If wrong-path updates were to be a problem, the problem could be solved by postponing parity table update until the conflicting μop retires from the ROB. This can be implemented as follows. First, when a branch misprediction is detected, if the wait bit is set and if the mispredicted branch is at a ROB position older than RP, the wait bit is reset. Second, when the instruction writing CPRN retires, the wait bit is not reset. Instead, the parity table index corresponding to this instruction is computed from its PC and saved in an *index* register, and a *valid-index* bit is set. Third, if the wait bit is still set when the μop at ROB entry RP retires, the wait bit is reset and, if the valid-index bit is set, the parity table entry pointed to by *index* is flipped.

We simulated both postponed update and immediate update and did not observe any significant IPC change. All the simulation results presented hereinafter are for immediate update.

4.4 Hardware cost of the parity table

The baseline IRF represents $160 \times 64 = 10240$ bits of storage. A 4K-bit parity table has less than half the number of bits of the baseline IRF. The parity table is actually much smaller, area-wise, than half the baseline IRF, as it only needs a single read port to read a bundle and a single one-bit write port to flip a parity bit.

The energy dissipated in the parity table is mostly from reads, as writes are much less frequent than reads and can be neglected. In theory, we must read one parity bit per instruction. In practice though, we read more than one bit because of x86 variable length instructions and because a whole instruction cache line is fetched. Assume a whole instruction cache line of L bytes is fetched. Let p_j be the probability that an instruction is a jump and assume that an instruction is I bytes. The average number of bytes between successive jumps is I/p_j. The average number of fetched lines per jump is $F = 1 + \frac{I/p_j - 1}{L}$ assuming the target of a jump can be at any position in a line (x86 ISA). The average number ϕ of useful instructions fetched per line is $\phi = \frac{1/p_j}{F} = \frac{L}{I + p_j(L-1)}$, and the number of parity bits read per x86 instruction is $L/\phi = I + p_j(L-1)$. For example, with $I = 4$, $L = 64$, $p_j = 1/8$, we read $L/\phi \approx 12$ bits per x86 instruction. With the ARMv8 ISA, we would read only $L/(I\phi) \approx 3$ bits per instruction.

This should be compared with the IRF bits read and written per instruction. On the integer SPECrate benchmarks, the median is about 1.6 IRF reads and 0.75 IRF writes per x86 instruction, that is, $1.6 \times 64 = 102$ bits read and $0.75 \times 64 = 48$ bits written per instruction. On the floating-point benchmarks, the median is about 1 IRF reads and 0.3 IRF writes per x86 instruction, that is, about 64 bits read and 19 bits written per instruction.

The much smaller area and much lower bit access rate both make energy dissipation in the parity table negligible compared to that in the IRF. Modeling the parity table with CACTI-P as a 512-byte SRAM with one 64-bit read port and one 64-bit write port,[14] we obtain 0.41 pJ of dynamic read energy

[14] The actual write width is one bit, but CACTI models the same data width for all ports.

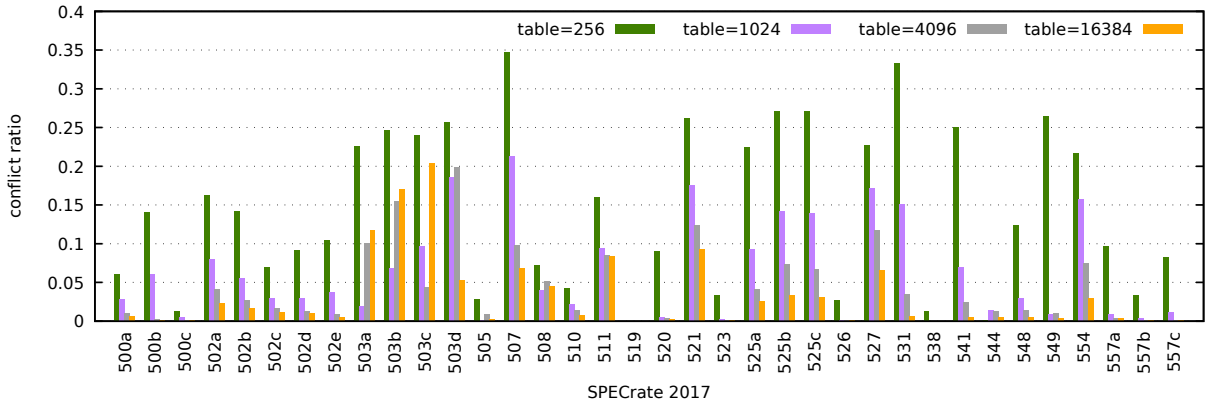

Figure 6: Fraction of dynamic integer 2-input μops having a bank conflict, for different parity table sizes (256 bits, 1 Kbit, 4 Kbits, 16 Kbits).

```
0   OUTERLOOP:
1   movsd    xmm1, [rbp+0]
2   xor      eax, eax
3   INNERLOOP:
4   movsd    xmm0, [rsp+rax]
5   mulsd    xmm0, xmm1
6   movsd    [rdx+rax], xmm0
7   add      rax, 8
8   cmp      rax, 8000
9   jne      INNERLOOP
10  add      rdx, 8000
11  add      rbp, 8
12  cmp      rdx, rcx
13  jne      OUTERLOOP
```

Listing 2: Example of loop nest with residual bank conflicts (Intel x86-64)

per 64-bit bundle and, neglecting parity table writes, we find that the total dynamic energy of the parity table is less than 1.5% that of the baseline IRF for all benchmarks except 519.lbm, for which it is 2.8%.

4.5 Residual bank conflicts

Figure 6 shows the bank conflict ratio, that is, the fraction of dynamic integer 2-input μops generating a bank conflict, for different parity table sizes. The parity table is generally effective at reducing the number of bank conflicts. Even a table as small as 256 bits keeps the conflict ratio under 35%, often bringing it under 25%. The number of conflicts decreases further with a larger parity table. With a 1 Kbit parity table, the conflict ratio is under 20% except for 507.cactuBSSN.

With a sufficiently large parity table, bank conflicts practically vanish for certain benchmarks. However, for other benchmarks, especially the floating-point ones, there is a residual conflict ratio that does not disappear even with a large parity table. Counterintuitively, the number of conflicts sometimes increases with the parity table size. This is most visible on 503.bwaves. This comes from architectural registers that are read many times without being modified, which we call *long-lived* registers. Long-lived registers are often used by the compiler for address computations in accessing arrays. Most of the residual bank conflicts that we observe in our simulations come from address computations.

Listing 2 shows a loop nest doing the outer product of a vector with itself (this is an ad hoc C program that we compiled with `gcc -O2`). The coupling graph corresponding to this loop nest is shown in Figure 7. The three edges in the graph correspond to the address computations in instructions

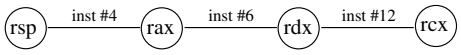

Figure 7: Coupling graph for the loop nest shown in Listing 2.

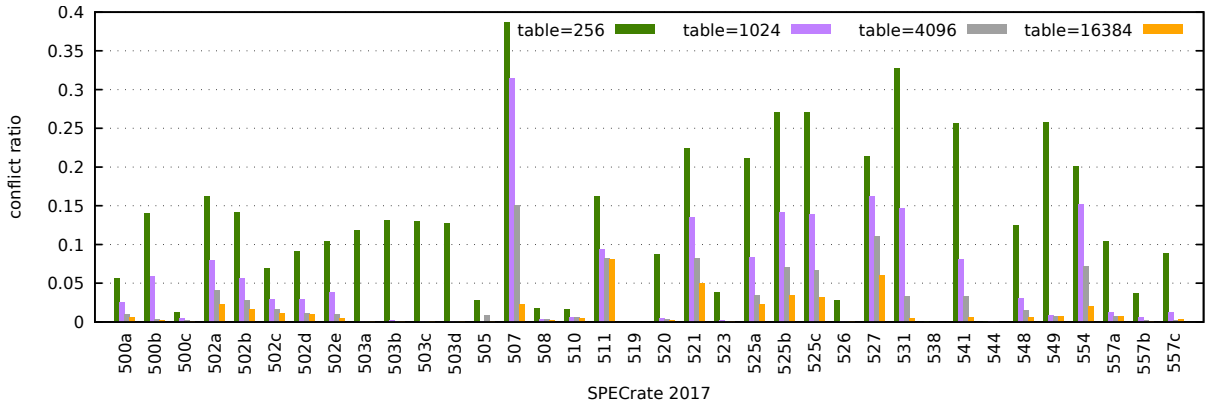

Figure 8: Effect of remap μops on the fraction of dynamic integer 2-input μops having a bank conflict.

#4 and #6 and the comparison #12. The vertex labeled `rax` corresponds to instruction #7 and the vertex labeled `rdx` corresponds to instruction #10. Registers `rsp` and `rcx` are initialized before the loop and the instructions writing these registers are not shown in Listing 2.

The coupling graph in Figure 7 is a tree, hence it is bipartite: it is possible to color the vertices so as to avoid any bank conflict. Such coloring implies that vertices `rsp` and `rcx` have different colors. In other words, registers `rsp` and `rcx` should be mapped to physical registers of different parity. However, the instructions writing `rsp` and `rcx` are executed only once, before the loop. The IRF banks where these instructions write is determined by whatever bit values are in the parity table entries these instruction map to. If we are unlucky, these two instructions write in the same bank, and bank conflicts must occur during the execution of the loop nest, as the color of vertices `rsp` and `rcx` cannot be changed. It should be noted that, even though most bank conflicts happen on instructions #4 and #6 (as they belong to the inner loop), bank conflicts on instruction #12 tend to switch vertex `rdx` back to the color different from that of vertex `rsp`, making instruction #12 indirectly responsible for bank conflicts on instructions #4 and #6.

4.6 Remap μops

The IPC loss from residual conflicts is negligible on our benchmarks, even on 503.bwaves. We do not know if residual conflicts can be a problem on some other workloads. Therefore, we propose a solution, described below.

Besides the wait bit, CPRN and RP, we introduce a 5-bit Conflict Architectural Register Number (CARN). When CPRN is set upon a conflicting 2-input μop, we also record in CARN the corresponding architectural register. If timeout occurs (i.e., the conflicting μop retires and the wait bit is still set), we do not reset the wait bit. Instead, we inject a *remap* μop at rename. The remap μop is equivalent to a `mov` CARN,CARN μop. It is renamed and executed like a normal μop. The remap μop is assigned the parity opposite to that of CPRN. One clock cycle is stolen at the rename stage for injecting the remap μop. To keep the hardware simple, this is the only μop that is renamed during this cycle. The remap μop is given a destination physical register corresponding to its assigned parity. The remap μop, when it executes, copies the value from the old physical register into the newly allocated one. When the remap μop retires, the old physical register is freed and the wait bit is reset. Note that there can be at most a single remap μop in the ROB at a time as the wait bit remains set until the remap μop is retired.

Figure 8 shows the effect of remap μops on the bank conflict ratio. Remap μops are effective at reducing the residual conflict ratio. With a 16 Kbits parity table, the fraction of integer 2-input μops with a bank conflict is under 5% except for 511.povray[15] and 527.cam4. Notice that, for 503.bwaves, which had the most residual conflicts, a 1-Kbit parity table practically removes all bank conflicts. With a 16-Kbit parity table, less than 0.05 remap μops are injected per 100 instructions on average except for one trace, 557a, where it is 0.07%. As the parity table gets smaller, the number of bank conflicts increases and so does the number of remap μops. With a 256-bit parity table, the average number of remap μops per instruction is less than 0.2% except for 541.leela where it is 0.4%. Remap μops tend to be more numerous in workloads experiencing frequent branch mispredictions, as the ROB occupancy is often low after wrong-path instructions have been deleted.

Nevertheless, while remap μops reduce residual conflicts, they do not affect the IPC in any significant way. We do not know if this is true only for our benchmarks and microarchitecture configuration or if remap μops might be useful in certain conditions.

4.7 Understanding the effect of a limited parity table

As can be seen in Figure 8, enlarging the parity table generally reduces the number of bank conflicts. This raises the question of the efficiency of the parity table. The parity table is just one array indexed through a hash function. Hash functions are commonly used in various microarchitectural tables such as caches and branch predictors. However, hash functions are subject to the birthday paradox [69]: the probability of collision in a hash table is not negligible even when the quantity of data to store is much smaller than the table. Caches, and other tables such as TLBs, generally solve the collision problem with set associativity, while branch predictors solve it with multiple tables and different hash functions [41, 43]. A direct-mapped parity table may seem simplistic in comparison. However, we do not believe that a more complicated scheme can do much better. A quick but superficial explanation is that there is a degree of freedom for removing a bank conflict as we can choose to flip either the left or right input of a conflicting 2-input μop.

The rest of this section proposes an analysis to help understand how the size of the parity table affects its ability to remove bank conflicts. The proposed analysis is based on some simplifying assumptions allowing to leverage basic results from random graph theory. See, for instance, chapter 11 in reference [46] for an accessible introduction to random graphs.

The coupling graph $\mathcal{G}$ introduced at the beginning of Section 4 associates a vertex with each instruction feeding a 2-input integer μop. If the graph is bipartite, the vertices can be colored with two colors in such a way that adjacent vertices always have different colors, which means no bank conflicts. However, this ignores the fact that the parity table has a finite size and that a table entry may be shared by several instructions. To understand the effect of a limited parity table, we must consider a graph G whose vertices are not instructions but parity table entries. Graph G is obtained by merging the vertices of $\mathcal{G}$ mapped to the same entry of the parity table. The merging of vertices may turn a bipartite $\mathcal{G}$ into a non-bipartite G. Our goal is to find a coloring of G such that the number of edges connecting vertices of the same color is minimum. This is the *maximum cut* (AKA *max-cut*) problem, which seeks to partition vertices into two complementary subsets so as to maximize the number of edges, AKA *cut*, going from one subset to the other [70]. Note that the maximum cut of a bipartite graph consists of all the edges.

To simplify the model, we assume that a register value is read by at most one 2-input μop. Under this assumption, all the vertices of $\mathcal{G}$ have degree one, that is, $\mathcal{G}$ consists of disconnected pairs of vertices connected by an edge. Thus, $\mathcal{G}$ has m edges and $2m$ vertices. With this simplification, graph G can be approximated as a $G(n, m)$ random graph[16] where the number n of vertices equals the number of parity table entries. Indeed, mapping an edge of $\mathcal{G}$ (i.e., a pair of vertices) onto the parity table through

[15] 511.povray has very few integer 2-input μops, see table 1.

[16] In the $G(n, m)$ random graph model, m distinct edges are randomly selected among the $\binom{n}{2}$ pairs of vertices [18, 46, 22].

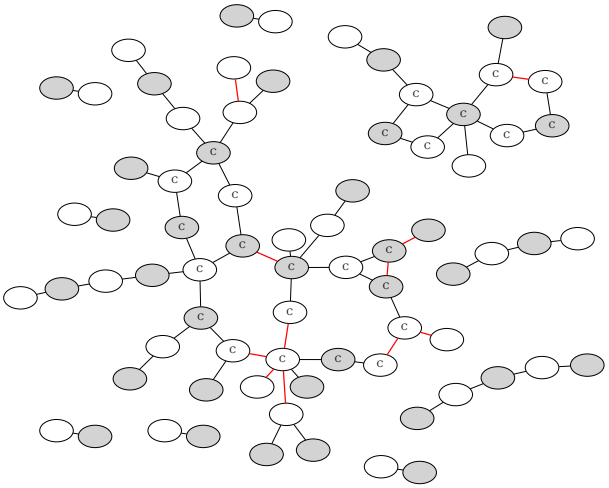

Figure 9: An example random graph G ($n = 100$, $m = 70$) and the coloring of vertices obtained after 10 million iterations of the SRCE algorithm. Isolated vertices (unused table entries) are not shown. Vertices labeled "C" belong to the 2-core of the graph. Conflicting edges are in red. The set of conflicting edges keeps changing as SRCE continues to iterate.

a randomizing hash function is akin to selecting two random vertices in G. As the merging of vertices preserves edges, m is also the number of edges of G.[17] The *average degree* c of G equals the total number of half-edges, that is, twice the number of edges, divided by the number of vertices of G:

$$c = \frac{2m}{n}$$

Note that the parity table entries onto which no vertex of $\mathcal{G}$ is mapped represent isolated vertices of G. Isolated vertices participate in the definition of the average degree c.

As long as $c \leq 1$, almost all the vertices of a random graph belong to isolated trees (theorems 4b and 4c in [18]). Therefore, when $m \leq n/2$, G is bipartite or almost bipartite. In other words, when the number of static 2-input integer μops does not exceed half the parity table size, there exists with high probability a 2-coloring of G making the number of bank conflicts negligible.

This is not an intuitive fact. Consider the critical threshold $c = 1$, that is, $m = n/2$. In this case, $\mathcal{G}$ has $2m = n$ vertices, i.e., as many vertices as the number of parity table entries. The average number of isolated vertices of G equals $n \times (1 - 1/n)^n \approx ne^{-1} \approx 0.37n$. The n vertices of $\mathcal{G}$ are mapped onto just 63% of the n table entries due to many collisions. Yet, from the theory of random graphs, we know that G is almost surely bipartite or nearly bipartite.

Above the critical threshold, that is, for $c > 1$, G is unlikely to be bipartite and the max-cut problem becomes difficult (max-cut is NP-hard in the general case [70]). The simple hardware algorithm described in Section 4.2 is not competitive with known software heuristics for finding approximate solutions to the max-cut problem. Our hardware algorithm is an approximation of a simple algorithm that we call SRCE for *Select a Random Conflicting Edge*. SRCE iterates the following steps *indefinitely*:

1. randomly select an edge;
2. if the selected edge is a conflicting one:
   (a) randomly select one of the two vertices bearing the selected edge;
   (b) flip the color of the selected vertex.

[17] While $\mathcal{G}$ is a *simple graph*, G is a multigraph [71], as merging vertices may create loops and parallel edges. Nevertheless, if n is sufficiently large, loops and parallel edges can be neglected.

Figure 9 shows an example random graph G and the coloring of vertices obtained after 10 million SRCE iterations. This graph is typical of random graphs above the critical threshold, with a significant fraction of vertices in a "giant" connected component and the other vertices in small components which are mostly trees [18, 46].

The vertices labeled with a "C" belong to the 2-core of G, which is the induced subgraph of G that remains after we have removed, recursively, all the vertices of degree less than two [46]. A random graph of degree $c > 1$ is very likely to feature both a giant connected component and a non-empty 2-core whose sizes increase simultaneously with c [22].

A non-empty 2-core generally prevents SRCE from removing all the conflicting edges because of the graph cycles it contains. If vertices are initialized with random colors, SRCE reduces the number of conflicting edges in the beginning. However, after a certain number of iterations, there remains residual conflicts that SRCE cannot remove. Note that residual conflicting edges are not fixed and keep changing in number and location as SRCE continues to iterate.

To model the behavior of SRCE (approximately), we make the following simplifying assumption, which we call the *independence assumption*: the probability that an edge is conflicting is assumed independent of the vertex degree and of whether some other edges are conflicting.

The first step is to determine the probability s_d that the vertex selected by SRCE has degree d. Under the independence assumption, s_d also equals the probability that a vertex selected by picking a random edge (whether conflicting or not) has degree d. It should be noted that s_d is distinct from the probability p_d that a random vertex has degree d. For instance, SRCE cannot select an isolated vertex ($s_0 = 0$). Nevertheless, s_d can be derived from p_d as follows. There are np_d vertices of degree d in G. The total number of half-edges attached to these vertices is $np_d d$. Hence, under the independence assumption, the probability that SRCE selects a vertex of degree d is

$$s_d = \frac{np_d d}{2m} = \frac{d}{c} p_d \tag{1}$$

The second step is to model the dynamic behavior of SRCE. Let x_t denote the fraction of edges that are conflicting at iteration t. The total number of conflicting edges is mx_t. With probability x_t, SRCE selects a vertex at iteration t and, with probability s_d, this vertex has degree d. Note that the selected vertex has at least one of its incident edges that is a conflict, which is the edge through which the vertex was selected. By flipping the color of a selected vertex of degree $d \geq 1$, SRCE removes $1 + (d-1)x_t$ conflicts and adds $(d-1)(1-x_t)$ new conflicts, under the independence assumption. The following recurrence relation summarizes the above paragraph:

$$mx_{t+1} = mx_t + x_t \sum_{d=1}^{\infty} \Big[-1 - (d-1)x_t + (d-1)(1-x_t) \Big] s_d$$

which can be written

$$x_{t+1} = g(x_t) \tag{2}$$

with

$$g(x) = x - \frac{x}{m} \Big(1 - \gamma + 2\gamma x \Big) \tag{3}$$

where γ is defined as

$$\gamma = \sum_{d=1}^{\infty} (d-1) s_d = \sum_{k=0}^{\infty} k q_k \tag{4}$$

with q_k defined as

$$q_k = s_{k+1} = \frac{k+1}{c} p_{k+1} \tag{5}$$

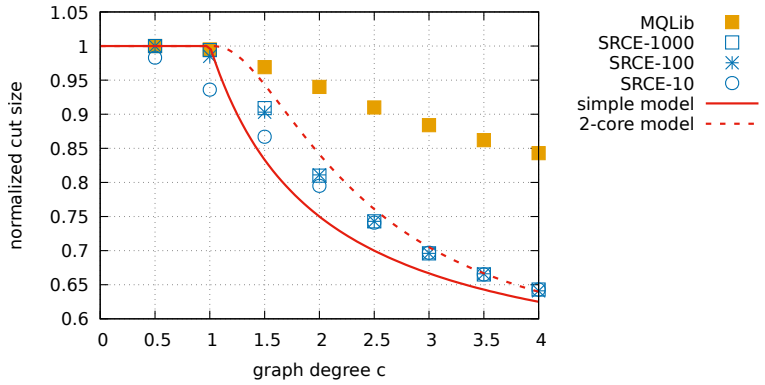

Figure 10: Average normalized cut size obtained by Monte Carlo simulations of SRCE on random graphs with $n = 1000$ vertices, as a function of the graph degree $c = 2m/n$, with m the number of edges. Three points are shown for SRCE, corresponding to the cut obtained after $10 \times m$, $100 \times m$ and $1000 \times m$ iterations. MQLib is an approximation of the maximum cut [17]. The cut size is averaged over 100 random graphs. For $c > 1$, the simple model is equation (7) and the 2-core model is equation (9).

where we have used equation (1). Probability q_k is known as the *excess degree distribution* [46], the excess degree being the number of incident edges on a vertex attached to a randomly selected edge, not counting the selected edge itself. Hence γ is the average excess degree of G. We show in appendix (A) that the average excess degree of a large random graph is equal to the average vertex degree, that is, $\gamma = c$. When $\gamma \leq 1$, the bracketed quantity on the right-hand side of (3) is positive, hence $g(x) < x$ for $x \in (0, 1)$ and iteration (2) converges to the fixed point $x_\infty = g(x_\infty) = 0$. This is consistent with the fact that SRCE eventually removes all conflicts when $c \leq 1$. Equation (2) also tells that the number of SRCE iterations it takes to remove (almost) all conflicts is roughly proportional to $m/(1-c)$.

When $\gamma > 1$, iteration (2) converges to the only fixed point of $g(x)$ in $(0, 1)$ which is

$$x_\infty = g(x_\infty) = \frac{1}{2}\left(1 - \frac{1}{\gamma}\right) \tag{6}$$

The normalized cut size for $c > 1$ is

$$\frac{|\text{cut}|}{m} = 1 - x_\infty = \frac{1}{2}\left(1 + \frac{1}{c}\right) \tag{7}$$

Figure 10 shows the normalized cut size obtained by Monte Carlo simulations of SRCE, averaged over 100 random graphs. The figure also shows the maximum cut estimated with the MQLib software [17] and the cut obtained after $10m$, $100m$ and $1000m$ SRCE iterations. It also shows the curve corresponding to equation (7) (solid red curve).

For $c \leq 1$, SRCE removes all the conflicts after iterating sufficiently many times. That is, the cut size equals the number m of edges. The closer c is to one, the more iterations are needed.

For $c > 1$, the model underestimates the number of conflicts that SRCE can remove. As SRCE progresses, conflicts tend to concentrate in the 2-core of G. The independence assumption, which is not mathematically exact, does not permit reproducing this empirical observation. We can obtain a better approximation of the asymptotic behavior of SRCE by considering the 2-core as an isolated graph, neglecting conflicts on edges not belonging to the 2-core. When $c > 1$, the number of 2-core edges is approximately equal to my^2 (see [22]) where y, the fraction of vertices in the giant connected component, is the root in $(0, 1)$ of the equation (equation 11.16 in [46])

$$y = 1 - e^{-cy} \tag{8}$$

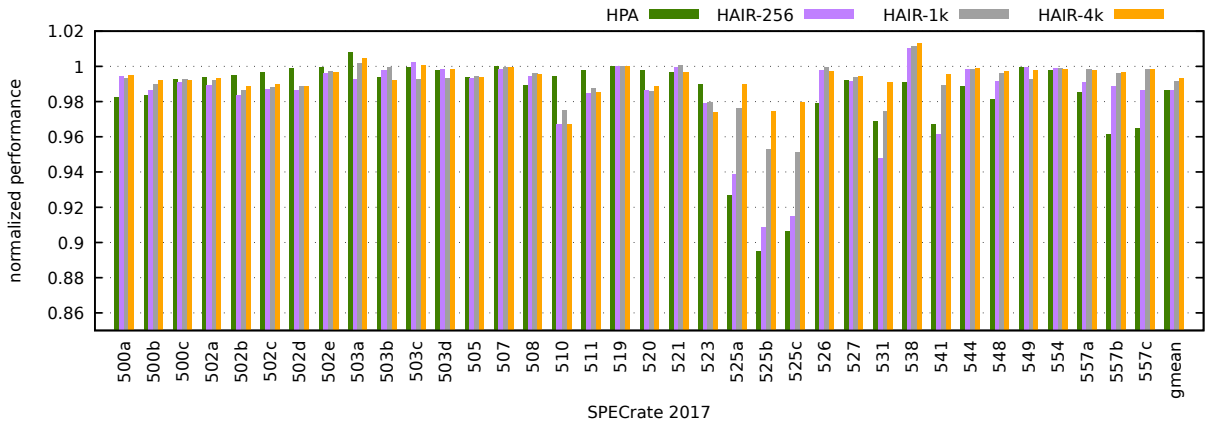

Figure 11: Performance of HAIR for different parity table sizes (256, 1K, and 4K bits). HPA is also shown (Section (3.2)).

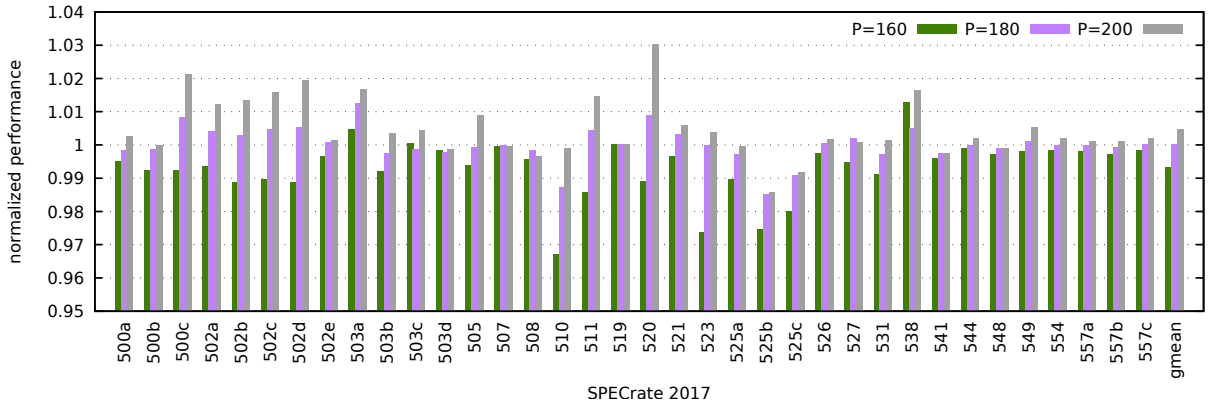

Figure 12: Performance of HAIR-4k with $P = 160$, $P = 180$ and $P = 200$ integer physical registers.

The number of conflicts for the isolated 2-core is my^2x_∞ where x_∞ is the asymptotic conflict probability (6). We show in appendix B that the average excess degree γ' of the 2-core is $\gamma' = c$. Hence the (overestimated) cut size is, for $c > 1$,

$$\frac{|\text{cut}|}{m} = 1 - y^2 x_\infty = 1 - \frac{y^2}{2}\left(1 - \frac{1}{c}\right) \tag{9}$$

This second model is shown as the dashed red curve in Figure 10.

In summary, the ability of the parity table to remove bank conflicts depends on c, the average number of distinct instructions sharing (i.e., writing) a table entry. When $c > 1$, the max-cut problem is difficult, and SRCE cannot approach the maximum cut, although it helps remove some conflicts. However, when $c < 1$, the max-cut problem is easy and SRCE removes practically all the conflicts after a sufficient number of iterations.

4.8 Speedup

Figure 11 shows the performance of HAIR for three different parity table sizes: 256, 1024 and 4096 bits. HAIR with a 4K parity table loses only 0.7% of IPC, on average, compared to the baseline. It also outperforms HPA by 0.7% on average. Looking at workloads individually, HAIR-4k loses up to 3.3% of IPC (on 510.parest) compared with the baseline while HPA loses up to 10.5% (on 525.x264).

Except for benchmark 525.x264, the small performance gap between HAIR-4k and the baseline does not come from bank conflicts. We checked that the occasionally unbalanced utilization of the STD

X-pipes has little effect on performance. The main cause of the gap is the dual free list. The parity table is generally a good enough random bit generator for balancing the utilization of the two free lists. Nevertheless, balancing is imperfect and register renaming sometimes stalls despite one of the two free lists containing some free registers (see Section 3.3).

Nevertheless, HAIR reduces the IRF delay (see Table 3), which can be exploited to increase the number of integer physical registers and recover some of the performance lost due to register renaming. Figure 12 shows the performance of HAIR-4k when the number of integer physical registers is increased from 160 to 180 and 200. With 20 extra registers, the average performance of HAIR-4k equals that of the baseline despite a slight performance loss on 525.x264 due to bank conflicts and on 510.parest due to register renaming.

HAIR's utility depends on many factors. Whether the energy vs. performance tradeoff offered by HAIR is better than that provided by voltage/frequency scaling depends on how clock frequency varies with voltage, how performance varies with frequency, how CPU energy varies with frequency and voltage and what fraction of the CPU energy is spent in the IRF. As for the temperature reduction, it depends on many factors, such as the CPU core floorplan, the number of cores, the thermal interface material, the heat sink, etc. A precise assessment of HAIR's utility is beyond the scope of this paper.

5 Related work

Several different microarchitecture ideas have been proposed to reduce the complexity of CPU register files. The related work closest to HAIR is the Half-Price Architecture [30], which we already discussed in Section 3.2.

Reducing the number of ports is one of the directions followed by researchers. Some μops read fewer than two registers, some do not write any registers. Moreover, source operands are often provided by the bypass network, but conventional CPU microarchitectures still read the register file for these operands, eventually discarding the read value [74, 66, 4, 48]. The bypass network can be viewed as a cache for computed but not yet written values. A register cache, smaller than the register file, might be introduced to further reduce the number of operands that must be read from the register file [74, 12, 4, 7, 60]. Once global arbitration is included in the scheduler logic to deal with register file ports contention, the register file can be further simplified with banking, where each bank holds a subset of the physical registers and of the read and write ports, bank conflicts being prevented by the scheduler [66, 4].

However, past propositions have often neglected the scheduling problem. In practice, these ideas are difficult to implement without increasing the hardware complexity of modern schedulers that attach μops to X-pipes at dispatch time. By providing dedicated register file ports to each X-pipe, conventional microarchitectures greatly simplify the scheduler logic, permitting the picker of each X-pipe to work independently from other X-pipes [24].

Instead of preventing register file port contention, another possibility is to tolerate it by issuing μops speculatively, ignoring port contention. If a μop cannot complete, its execution is aborted and the μop is reissued [65]. However, this approach hurts performance and wastes energy.

Another possibility is to allocate register file ports to a μop before the μop is inserted into the scheduler. Seznec et al. described a clustered microarchitecture, called WSRS, in which the register file is banked so that each cluster is specialized to read and write in only a subset of the banks [59]. For WSRS to be viable, every μop must be executable somewhere, and therefore each cluster must be able to execute any type of μop [59]. It is not clear how this idea could be adapted for conventional, non-clustered microarchitectures, whose X-pipes are generally not identical.

The related work mentioned above, like HAIR, considers the register file complexity problem at the microarchitecture level and from one specific angle: the number of ports. However, register file complexity also depends on the physical register file capacity, which has two dimensions: the number of

physical registers and the register width. Various ideas have been proposed to better utilize physical registers along either or both of these dimensions, so that the same performance can be obtained with fewer physical registers or, more generally, with less storage. Examples of such ideas include early register release [45, 20, 63], physical register sharing [29, 3, 64, 49], and exploiting narrow values [36, 19, 31]. Most of these ideas are in principle compatible with HAIR, except those requiring to change register renaming in a radical way, such as late register allocation [44]. For example, physical register sharing is admittedly implemented in certain commercial processors for eliminating register-to-register move μops [29, 49, 28, 2]. Move elimination is done at register renaming by mapping the architectural destination register onto the physical source register. Combining HAIR with move elimination is straightforward: a move μop is removed exactly in the same manner with or without HAIR (the parity bit assigned to the move is just ignored).

6 Conclusion

The proposed HAIR scheme halves the area of the integer register file by splitting it into two banks holding even-numbered and odd-numbered integer physical registers, respectively, each bank providing a read port to each ALU and AGU. When both inputs of a 2-input μop lie in the same bank, this is a bank conflict, and the μop takes one extra cycle to read its inputs, which may hurt performance. We have shown that the bank conflict probability can be reduced with a parity table providing a bit to each instruction for deciding the parity of the destination physical register. The parity table is much smaller, area-wise, than the register file and dissipates much less energy. On our simulated microarchitecture running the SPECrate 2017 benchmarks, a 4 Kbit parity table recovers most of the performance loss from bank conflicts. We have proposed an analytical model to explain why a direct-mapped parity table is sufficient. The HAIR scheme can be used to reduce energy and/or increase the number of integer physical registers.

In theory, the HAIR scheme could be applied to the floating-point register file too. In practice, however, HAIR may not be as useful for floating-point registers as for integer registers. Floating-point registers generally have fewer read ports than integer registers, for two reasons. First, floating-point registers are not involved in address computations. Second, higher floating-point throughput can be obtained with wider floating-point vectors, not necessarily with a greater number of register file ports. Moreover, HAIR assumes 2-input μops, but some floating-point operations, such as fused multiply-add, read three floating-point registers. It is not clear how odd/even banking could be used in this case.

Acknowledgments

This research was partially supported by an Intel research grant. We thank Arthur Perais and the anonymous reviewers for their feedback.

A Average excess degree of large random graphs

In a large random graph, the vertex degree follows a Poisson distribution [46]

$$p_d = e^{-c}\frac{c^d}{d!} \tag{10}$$

Combining (4), (5) and (10), we find that the average excess degree of G is

$$\gamma = \sum_{k=0}^{\infty} k\frac{k+1}{c}e^{-c}\frac{c^{k+1}}{(k+1)!} = \frac{e^{-c}}{c}\sum_{k=1}^{\infty}\frac{c^{k+1}}{(k-1)!} = ce^{-c}\sum_{k=1}^{\infty}\frac{c^{k-1}}{(k-1)!} = c \tag{11}$$

The average excess degree of a large random graph is equal to the graph degree.

B Average excess degree of the 2-core of large random graphs

To obtain the average excess degree γ' of the 2-core of a random graph G, we need the degree distribution p'_d of the 2-core of G, which can be derived from previous work [16, 15] assuming a Poisson distribution for p_d: for $c > 1$ and $d \geq 2$,

$$p'_d = \frac{e^{-cy}}{y(1-c+cy)} \frac{(cy)^d}{d!} \tag{12}$$

where c is the average degree of G and y is the solution in $(0, 1)$ of equation (8). The average degree of the 2-core is

$$c' = \sum_{d=2}^{\infty} dp'_d = \frac{cy}{1-c+cy} \tag{13}$$

where we have used (8). The excess degree distribution q'_k of the 2-core is obtained from (5), (12) and (13): for $k \geq 1$,

$$q'_k = \frac{k+1}{c'} p'_{k+1} = \frac{e^{-cy}}{y} \frac{(cy)^k}{k!}$$

The average excess degree distribution of the 2-core is

$$\gamma' = \sum_{k=1}^{\infty} kq'_k = \frac{e^{-cy}}{y} \sum_{k=1}^{\infty} \frac{(cy)^k}{(k-1)!} = \frac{e^{-cy}}{y} cy \sum_{k=1}^{\infty} \frac{(cy)^{k-1}}{(k-1)!} = c$$

The average excess degree of the 2-core of a large random graph is equal to the graph degree.

References

[1] A. Abel and J. Reineke. uiCA: accurate throughput prediction of basic blocks on recent Intel microarchitectures. In *International Conference on Supercomputing (ICS)*, 2022. URL: https://uica.uops.info/.

[2] AMD. Software optimization guide for AMD EPYC 7003 processors. Publication 56665, revision 3.00, November 2020. URL: https://developer.amd.com/resources/developer-guides-manuals/.

[3] S. Balakrishnan and G. S. Sohi. Exploiting value locality in physical register files. In *International Symposium on Microarchitecture (MICRO)*, 2003.

[4] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *International Symposium on Microarchitecture (MICRO)*, 2001.

[5] A. W. Barr, A. L. Cox, and S. Rixner. Translation caching: skip, don't walk (the page table). In *International Symposium on Computer Architecture (ISCA)*, 2010.

[6] D. Berend and A. Kontorovich. A sharp estimate of the binomial mean absolute deviation with applications. *Statistics and Probability Letters*, 83(4):1254–1259, April 2013.

[7] E. Borch, E. Tune, S. Manne, and J. Emer. Loose loops sink chips. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2002.

[8] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6), November 2000.

[9] T. N. Buti, R. G. McDonald, Z. Khwaja, A. Ambekar, H. Q. Le, W. E. Burky, and B. Williams. Organization and implementation of the register-renaming mapper for out-of-order IBM POWER4 processors. *IBM Journal of Research and Development*, 49(1), January 2005.

[10] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *International Symposium on Computer Architecture (ISCA)*, 1998.

[11] Standard Performance Evaluation Corporation. SPEC CPU 2017, 2017. URL: https://www.spec.org/cpu2017/.

[12] J.-L. Cruz, A. González, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *International Symposium on Computer Architecture (ISCA)*, 2000.

[13] G. S. Ditlow, R. K. Montoye, S. N. Storino, S. M. Dance, S. Ehrenreich, B. M. Fleischer, T. W. Fox, K. M. Holmes, J. Mihara, Y. Nakamura, S. Onishi, R. Shearer, D. Wendel, and L. Chang. A 4R2W register file for a 2.3GHz wire-speed POWER processor with double-pumped write operation. In *International Solid-State Circuits Conference (ISSCC)*, 2011.

[14] J. Donald and M. Martonosi. Leveraging simultaneous multithreading for adaptive thermal control. In *Workshop on Temperature-Aware Computer Systems (TACS)*, 2005.

[15] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes. k-core architecture and k-core percolation of complex networks. *Physica D: Nonlinear Phenomena*, 224(1-2), December 2006.

[16] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes. k-core organisation of complex networks. *Physical Review Letters*, 96(4):040601, February 2006.

[17] I. Dunning, S. Gupta, and J. Silberholz. What works best when? a systematic evaluation of heuristics for max-cut and QUBO. *INFORMS Journal on Computing*, 30(3), 2018.

[18] P. Erdős and A. Rényi. On the evolution of random graphs. *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, 5, 1960. URL: https://users.renyi.hu/~p_erdos.

[19] O. Ergin, D. Balkan, K. Ghose, and D. Ponomarev. Register packing: exploiting narrow-width operands for reducing register file pressure. In *International Symposium on Microarchitecture (MICRO)*, 2004.

[20] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. Increasing processor performance through early register release. In *International Conference on Computer Design (ICCD)*, 2004.

[21] J. A. Farrell and T. C. Fischer. Issue logic for a 600-MHz out-of-order execution processor. *IEEE Journal of Solid-State Circuits*, 33(5), May 1998.

[22] A. Frieze and M. Karoński. *Introduction to Random graphs*. Cambridge University Press, 2016.

[23] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. The Intel Pentium M processor: microarchitecture and performance. *Intel Technology Journal*, 7(2), May 2003.

[24] A. González, F. Latorre, and G. Magklis. *Processor microarchitecture: an implementation perspective*, chapter 6, page 54. Morgan & Claypool, 2011.

[25] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7), July 2008.

[26] S. K. Hsu. *Advanced microarchitecture and circuit design techniques for on-chip memories in CMOS technology*. PhD thesis, Oregon State University, 2006. URL: https://ir.library.oregonstate.edu/concern/graduate_thesis_or_dissertations/r207tt70n.

[27] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron. Scaling with design constraints: predicting the future of big chips. *IEEE Micro*, 31(4), July 2011.

[28] Intel. Intel 64 and IA-32 architectures optimization reference manual. June 2021. URL: https://www.intel.com.

[29] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *International Symposium on Microarchitecture (MICRO)*, 1998.

[30] I. Kim and M. H. Lipasti. The half-price architecture. In *International Symposium on Computer Architecture (ISCA)*, 2003.

[31] M. Kondo and H. Nakamura. A small, fast and low-power register file by bit-partitioning. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[32] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In *International Symposium on Microarchitecture (MICRO)*, 2003.

[33] D. Leibholz and R. Razdan. The Alpha 21264: a 500 MHz out-of-order execution processor. In *IEEE Computer Society International Conference (COMPCON)*, 1997.

[34] S. Li, J. Ho. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. The McPAT framework for multicore and manycore architectures: simultaneously modeling power, area and timing. *ACM Transactions on Architecture and Code Optimization*, 10(1), April 2013.

[35] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. CACTI-P: architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *International Conference on Computer-Aided Design (ICCAD)*, 2011.

[36] M. H. Lipasti, B. R. Mestan, and E. Gunadi. Physical register inlining. In *International Symposium on Computer Architecture (ISCA)*, 2004.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. Pin : building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[38] A. J. Martin, M. Nyström, and P. I. Pénzes. ET^2: A metric for time and energy efficiency of computation. In R. Graybill and R. Melhem, editors, *Power Aware Computing*. Springer, 2002. URL: https://resolver.caltech.edu/CaltechCSTR:2001.007.

[39] H. McIntyre, S. Arekapudi, E. Busta, T. Fischer, M. Golden, A. Horiuchi, T. Meneghini, S. Naffziger, and J. Vinh. Design of the two-core x86-64 AMD "Bulldozer" module in 32 nm SOI CMOS. *IEEE Journal of Solid-State Circuits*, 47(1), January 2012.

[40] A. A. Merchant and D. S. Sager. Scheduling operations using a dependency matrix. US patent 6334182, August 1998.

[41] P. Michaud. A statistical model of skewed associativity. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2003.

[42] P. Michaud. Best-offset hardware prefetching. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[43] S. Mittal. A survey of techniques for dynamic branch prediction. *Concurrency and Computation: Practice and Experience*, 31(1), January 2019.

[44] T. Monreal, A. González, M. Valero, J. González, and V. Viñals. Delaying physical register allocation through virtual-physical registers. In *International Symposium on Microarchitecture (MICRO)*, 1999.

[45] T. Monreal, V. Viñals, A. González, and M. Valero. Hardware schemes for early register release. In *International Conference on Parallel Processing (ICPP)*, 2002.

[46] M. Newman. *Networks*. Oxford University Press, second edition, 2018.

[47] H. Nguyen, J. Jeong, F. Atallah, D. Yingling, and K. Bowman. A 7-nm 6R6W register file with double-pumped read and write operations for high-bandwidth memory in machine learning and CPU processors. *IEEE Solid-State Circuits Letters*, 1(12), December 2018.

[48] I. Park, M. D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *International Symposium on Microarchitecture (MICRO)*, 2002.

[49] A. Perais and A. Seznec. Cost effective physical register sharing. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[50] A. Perais, A. Seznec, P. Michaud, A. Sembrant, and E. Hagersten. Cost-effective speculative scheduling in high performance processors. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[51] R. P. Preston, R. W. Badeau, D. W. Bailey, S. L. Bell, L. L. Biro, W. J. Bowhill, D. E. Dever, S. Felix, R. Gammack, V. Germini, M. K. Gowan, P. Gronowski, D. B. Jackson, S. Mehta, S. V. Morton, J. D. Pickholtz, M. H. Reilly, and M. J. Smith. Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading. In *International Solid-State Circuits Conference (ISSCC)*, 2002.

[52] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[53] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *International Symposium on Microarchitecture (MICRO)*, 1999.

[54] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *International Symposium on Computer Architecture (ISCA)*, 2000.

[55] E. Rotem, A. Yoaz, L. Rappoport, S. Robinson, Y. Mandelblat, A. Gihon, E. Weissman, R. Chabukswar, V. Basin, R. Fenger, M. Gupta, and A. Yasin. Intel Alder Lake CPU architectures. *IEEE Micro*, 42(3), May 2022.

[56] P. G. Sassone, J. Rupley, E. Brekelbaum, G. H. Loh, and B. Black. Matrix scheduler reloaded. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[57] A. Seznec and P. Michaud. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction Level Parallelism*, 8, February 2006.

[58] A. Seznec, J. San Miguel, and J. Albericio. The inner most loop iteration counter: a new dimension in branch history. In *International Symposium on Microarchitecture (MICRO)*, 2015.

[59] A. Seznec, E. Toullec, and O. Rochecouste. Register write specialization register read specialization: a path to complexity-effective wide-issue superscalar processors. In *International Symposium on Microarchitecture (MICRO)*, 2002.

[60] R. Shioya, K. Horio, M. Goshima, and S. Sakai. Register cache system not for latency reduction purpose. In *International Symposium on Microarchitecture (MICRO)*, 2010.

[61] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1), January 2015.

[62] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *International Symposium on Computer Architecture (ISCA)*, 2003.

[63] H. Tabani, J.-M. Arnau, J. Tubella, and A. González. A novel register renaming technique for out-of-order processors. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[64] L. Tram, N. Nelson, F. Ngai, S. Dropsho, and M. Huang. Dynamically reducing pressure on the physical register file through simple register sharing. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2004.

[65] J. H. Tseng and K. Asanović. Banked multiported register files for high-frequency superscalar microprocessors. In *International Symposium on Computer Architecture (ISCA)*, 2003.

[66] S. Wallace and N. Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1996.

[67] N. H. E. Weste and D. M. Harris. *CMOS VLSI design: a circuits and systems perspective*. Addison-Wesley, fourth edition, 2010. Chapter 12.

[68] Wikipedia. Bipartite graph, 2022. URL: https://en.wikipedia.org/wiki/Bipartite_graph.

[69] Wikipedia. Birthday problem, 2022. URL: https://en.wikipedia.org/wiki/Birthday_problem.

[70] Wikipedia. Maximum cut, 2022. URL: https://en.wikipedia.org/wiki/Maximum_cut.

[71] Wikipedia. Multigraph, 2022. URL: https://en.wikipedia.org/wiki/Multigraph.

[72] K. C. Yeager. The Mips R10000 superscalar microprocessor. *IEEE Micro*, 16(2), April 1996.

[73] T.-Y. Yeh. Low-power, high-performance architecture of the PWRficient processor family. *IEEE Micro*, 27(2), March 2007.

[74] R. Yung and N. C. Wilhelm. Caching processor general registers. In *International Conference on Computer Design (ICCD)*, 1995.

[75] N. Zaidi, G. Hammond, and K. Shoemaker. Method and apparatus for scheduling instructions in waves. US patent 6016540, January 1997.

[76] V. Zyuban. Unified architecture level energy-efficiency metric. In *Great Lakes Symposium on VLSI (GLVLSI)*, 2002.