Simulating Multiple Functional Unit Pools and Partial Bypass Networks in gem5

David Zhao Akeley

April 12, 2020

Abstract

Bypassing between functional units is crucial for reducing the latency beteen instruction executions in an out-of-order processor. The mainline gem5 hardware simulator's out-of-order processor supports simulating complete bypassing between all functional units. However, with the high number of functional units provided in superscalar processors, it may not be practical to realize such complete bypassing in hardware.

This paper describes a modification to the gem5 simulator that allows it to simulate some partial bypass networks. The modification splits functional units into one or more functional unit pools, and simulates bypassing only between units in the same pool. Bypassing failures are modeled as a one cycle penalties, which are assessed whenever newly-produced values need to be communicated between pools. This paper characterizes bypassing failures as either congestion, (in)capability, or confluence failures, and describes statistics added to gem5 to count each failure type.

This modification is then used to measure the performance loss on four workloads incurred by replacing a complete bypass network with one that is split between two or four functional unit pools, and to evaluate different simple strategies for assigning instructions to functional unit pools. With the best-performing strategy, the performance impact of splitting the representative processor's functional units into four functional unit pools is relatively low – on the order of 2%.

Code:

`https://github.com/akeley98/FU-pools/commit/734a89e13548824a5675d1d0e18c6de5b0e0c8f1`

1 Introduction

In an out-of-order processor, bypassing allows instructions waiting on a data hazard to start execution as soon as their last-arriving dependency is produced by a functional unit (FU), without incurring the latency of waiting for the write-back to occur. Bypassing can significantly decrease execution time; however, given that the cost of bypassing grows quadratically with the number of FUs connected, it may not be feasible to support complete bypassing between FUs in modern processors with wide superscalar pipelines.

As a result, processors may support only partial bypassing, providing only a subset of all possible bypassing connections between FUs. This risks increasing the execution latency of chains of dependent instructions – if an issuing instruction is assigned to an FU that does not support bypassing with the FU producing the last-arriving input value, then the issuing instruction must wait for the input value to be visible by other means. This motivates the search for strategies for assigning instructions to FUs in a way that minimizes the impact of such "bypassing failures".

To this end, this paper describes a modification to the gem5 simulator's `FullO3CPU` out-of-order processor that allows it to model partial bypassing, and to collect statistics on the number of bypass failures, broken down into different categories. This modification is then used to evaluate the performance impact of various simple FU assignment strategies on a small number of representative workloads.

2 Bypassing Model

At time of writing, the mainline gem5 `FullO3CPU` supports a single functional unit pool, which models a collection of FUs of various capabilities, with a complete bypassing network connecting all FUs. Any instruction is ready for execution as soon as its last input value is ready on any FU. I modified the `FullO3CPU` to support multiple functional pools, with complete bypassing between FUs within a single pool, and no bypassing between different pools. If a value is ready on a certain FU pool on clock cycle C, then, for that clock cycle, ready instructions dependent on that value may only begin execution on said FU pool. This restriction is lifted on cycle $C + 1$, at which point the produced value is considered visible to all FUs. This matches physical implementations that spend one cycle writing a result back to a renamed physical register file visibled to all functional units. I don't attempt to model physical implementations that impose a higher, or variable, penalty for bypass failures (such as implementations that only make values globally visible when committed to the architectural state).

These multiple FUs are specified using the new optional `fuPools` Python list parameter. For backwards-compatibility, if `fuPools` is not specified, the older `fuPool` parameter is used, which models one FU pool with complete bypassing as before.

3 Bypassing Examples & Failure Modes

This section describes categories of bypassing failures, and the statistics my modification to `gem5` collects on them. I first describe my debug log format, and an example of successful bypassing.

To help with understanding and debugging the bypassing model presented, I provide two new gem5 debug flags, `IQFU`, and `SIMDFU`. `IQFU` enables debug traces that

1. Print the renamed destination register, clock cycle, and producing FU pool number of each value produced.
2. Print the renamed source and destination registers (separated by '`->`') of each instruction that the CPU attempts to execute. If successful, the number of the FU pool assigned to execute the instruction is also printed. If unsuccessful, the reason for failure is printed.

The `SIMDFU` flag is similar, but only prints traces for instructions categorized as SIMD instructions (which includes most scalar float instructions for x86, due to SSE).

This is an example of successful instruction executions' debug trace output.

```
13567000: system.cpu.iq: cycleCounter 17562: r139 produced on FU pool 1.
...
13567000: system.cpu.iq: cycleCounter 17562: r139 r220 r139 -> r213
         executing on FU pool 1 (bypassed).
...
13567500: system.cpu.iq: cycleCounter 17563: r213 produced on FU pool 1.
...
13585000: system.cpu.iq: cycleCounter 17598: r213 r213 -> r11
         executing on FU pool 0 (no bypassing).
```

The first instruction executed is dependent on register 139. Because the instruction is executed on the same clock cycle that register 139's value was produced – cycle 17562 – the CPU has no choice but to attempt to assign the instruction to FU pool 1, which is capable of bypassing register 139's value. In this case, the assignment was successful.

The second instruction executed is dependent on the value produced by the first instruction. However, because enough clock cycles have elapsed since register 213's value was produced, bypassing is not needed to supply the second instruction's input values, and the CPU is free to assign the instruction to any FU pool. In this case, the CPU chose to assign the instruction to a different FU pool, pool number 0.

The total number of instructions executed using at least one bypassed input value, and the number executed without bypassing, is reported by the new `insts_with_bypassing` and `insts_without_bypassing` statistics, respectively.

3.1 Congestion Bypass Failure

A congestion bypass failure occurs when an instruction's input values have all been produced, but one or more of the inputs are young enough that bypassing is needed for them to be visible, and all FUs that see the bypassed inputs and are capable of executing the ready instruction are busy.[1] (Note that by this definition, a congestion bypass failure can still occur even with complete bypassing).

Suppose that we have a CPU with several FU pools, each of which has only one multiplier, and it executes the pseudocode

[1] Excluding the special case of there being no FUs capable of executing the instruction at all.

```
r1 = mul r1 r3 ; executed on cycle 0, latency 4 cycles
r0 = add r0 r2 ; executed on cycle 0, latency 1 cycle
r4 = mul r0 r4 ; executed on ???
```

The `r1 = mul r1 r3` and `r0 = add r0 r2` instructions are unrelated. Suppose that the add takes 1 clock cycle, while the multiply takes 4. If the unrelated instructions are both executed on the same FU pool, say pool 0, then the CPU will experience a congestion bypass failure on cycle 1. At that point, `r0`'s value is available for bypassing within pool 0, but pool 0 cannot execute the dependent `mul r0 r4` instruction as the pool's multiplier is still busy. The CPU will either have to wait for pool 0's multiplier to be ready, or wait one clock cycle for `r0`'s value to be visible globally, and attempt to schedule the multiply on another FU pool, which might have a free multiplier.

The possibility of congestion then suggests that unrelated instructions should be executed on different FU pools, in order to as much as possible keep FUs ready to process bypassed values.

Here is an example of debug trace output for a congestion bypass failure:

```
 651500: system.cpu.iq: cycleCounter 758: r168 produced on FU pool 1.
 651500: system.cpu.iq: cycleCounter 758: r137 produced on FU pool 1.
 651500: system.cpu.iq: cycleCounter 758: r413 produced on FU pool 1.
 651500: system.cpu.iq: cycleCounter 758: r414 produced on FU pool 1.
 651500: system.cpu.iq: cycleCounter 758: r147 produced on FU pool 1.
 651500: system.cpu.iq: cycleCounter 758: r145 produced on FU pool 1.
 651500: system.cpu.iq: cycleCounter 758: r143 produced on FU pool 1.
 651500: system.cpu.iq: cycleCounter 758: r16 r247 r137 r155 ->
         executing on FU pool 1 (bypassed).
 651500: system.cpu.iq: cycleCounter 758: r195 r195 r413 -> r16 r415 r416
         executing on FU pool 1 (bypassed).
 651500: system.cpu.iq: cycleCounter 758: r16 r193 r155 -> r165
         executing on FU pool 1 (bypassed).
 651500: system.cpu.iq: cycleCounter 758: r16 r168 r155 -> r171
         congestion bypass fail on FU pool 1.
 651500: system.cpu.iq: cycleCounter 758: r153 -> r144
         executing on FU pool 1 (no bypassing).
...
 652000: system.cpu.iq: cycleCounter 759: r16 r168 r155 -> r171
         executing on FU pool 1 (no bypassing).
```

In this example, on cycle 758, the `r16 r168 r155 -> r171` instruction can only be executed on FU pool 1, since the `r168` input was produced in the same cycle on FU pool 1. However, the capable FUs in pool 1 have already been reserved at that point. (Apparently, the subsequent `r153 -> r144` instruction was of a different class and did not experience congestion). In this example, the CPU chose to wait one clock cycle for the needed FU to be free, and again attempt to schedule the delayed instruction on pool 1. Because of the delay, bypassing was no longer needed.

The new `congestion_bypass_fails` statistic counts the number of congestion bypass failures.

3.2 Capability Bypass Failure

A capability bypass failure occurs when an instruction's inputs are ready with bypassing, but none of the FUs that see the bypassed value are capable of executing the instruction. This is counted by the `capability_bypass_fails` statistic.[2] However, this paper does not further investigate capability bypass failures – all simulations run used identical FU pools.

3.3 Confluence Bypass Failure

A confluence bypass failure occurs when an instruction has multiple last-arriving input values from different producers, but no one FU supports bypassing from all producers. It is so named because they can only occur for instructions that join two chains in a dependency graph, and because "confluence" starts with the letter C.

Suppose that this pseudocode is executed:

```
r0 = add r0 r2 ; executed on cycle 0, latency 1 cycle
r1 = add r1 r3 ; executed on cycle 0, latency 1 cycle
r4 = add r0 r1
```

The `r0 = add r0 r2` and `r1 = add r1 r3` instructions are, at first, unrelated. Suppose that, acting on the findings of the congestion bypass failure section, the two instructions are assigned to different FU pools. Then, `r0` and `r1` will be ready on cycle 1, but on different FU pools. There will then be no FU pool capable of executing `r4 = add r0 r1`; none have all inputs values visible simultaneously. The CPU will then be forced to wait for an additional cycle for the values of `r0` and `r1` to be visible globally.

This suggests that the goals of reducing congestion failures and reducing confluence failures are, at least on a surface level, exactly opposed, making it difficult to devise a simple "perfect strategy". Had the instructions producing `r0` and `r1` been assigned to the same FU pool, the confluence bypass failure would not have occured.

Here is an example of debug trace output for a confluence bypass failure:

```
6179500: system.cpu.iq: cycleCounter 8879: r14 produced on FU pool 3.
6179500: system.cpu.iq: cycleCounter 8879: r223 produced on FU pool 2.
6179500: system.cpu.iq: cycleCounter 8879: r186 produced on FU pool 2.
6179500: system.cpu.iq: cycleCounter 8879: r67 produced on FU pool 1.
6179500: system.cpu.iq: cycleCounter 8879: r14 r223 r153 ->
         confluence bypass fail.
6180000: system.cpu.iq: cycleCounter 8880: r14 r223 r153 ->
         executing on FU pool 3 (no bypassing).
```

The instruction that attempted to execute had multiple last-arriving inputs – `r14` and `r223` – ready on cycle 8879. However, they were produced on different FU pools, so on cycle 8879, no single FU pool had the information needed to execute the instruction. The CPU was then forced to wait one cycle, to cycle 8880, in order to execute the instruction.

The `confluence_bypass_fails` statistic counts all such failures.

[2] I didn't test this though.

4 Bypassing Simulation Implementation

This section summarizes the strategy used to implement the within-FU-pools bypassing model in gem5. Some of the quoted code has since been moved to `src/cpu/o3/fu_pools_strategy.hh`.

The gem5 simulator dynamically allocates a structure for each instruction in-flight, which is accessed through the `DynInstPtr` type and is used to store the status of an instruction through each pipeline stage. The structure does not seem to correspond to any physical hardware component.

Among the state stored within this structure is the number of input source registers whose values are ready (`readyRegs`). Every time a needed input value is produced, the `markSrcReady` function is called, which increments the number of ready registers and marks the instruction as ready to issue when the last input is received:

```
template <class Impl>
void
BaseDynInst<Impl>::markSrcRegReady()
{
    DPRINTF(IQ, "[sn:%lli] has %d ready out of %d sources. RTI %d)\n",
            seqNum, readyRegs+1, numSrcRegs(), readyToIssue());
    if (++readyRegs == numSrcRegs()) {
        setCanIssue();
    }
}
```

It's not so clear to me what happens if there are duplicate arguments, and since the actual ready register's number is not given, care has to be taken not to call `markSrcRegReady` twice for the same operand. Somehow it works anyway, and understanding why was not crucial. To provide the additional information needed to model FU pool bypassing, I simply had to write a new function, `markSrcRegReadyFuPool`, as a drop-in replacement wherever `markSrcRegReady` was used.

The `markSrcRegReadyFuPool` function provides the FU pool number (index) and cycle number that the source register's value was produced on. This allows the instruction record to know

1. When and where bypassing can be done; recorded by the final values of `latestSrcRegCycle` and `srcRegFuPoolIndex` when the last input register is marked ready.
2. If a confluence bypass failure will occur while scheduling this instruction. This occurs if the last few calls to `markSrcRegReadyFuPool` are on the same clock cycle, but have different `fuPoolIndex` arguments. A confluence failure is indicated by the magic -1 `srcRegFuPoolIndex` value.

   ```
   void
   markSrcRegReadyFuPool(int fuPoolIndex, Cycles currentCycle)
   {
       assert(fuPoolIndex >= 0);
       this->markSrcRegReady();

       // New cycle? Older registers are now visible on all FUs, so
       // just record the FU producing the new result for this cycle.
   ```

```
        if (latestSrcRegCycle != currentCycle) {
            latestSrcRegCycle = currentCycle;
            srcRegFuPoolIndex = fuPoolIndex;
        }
        // Otherwise, if multiple results on different FU Pools are
        // available this cycle, record via srcRegFuPoolIndex that
        // bypassing is not possible this cycle.
        else if (srcRegFuPoolIndex != fuPoolIndex) {
            srcRegFuPoolIndex = -1;
        }
    }
```

When the instruction is ready to issue, the `srcRegFuPoolIndex` indicates which FU pool (if any) can see the bypassed values and may execute the instruction. If the current clock cycle is greater than that recorded by `latestSrcRegCycle`, all inputs are old enough to not need bypassing to be visible, and the value of `srcRegFuPoolIndex` is ignored.

```
    /** Given that the instruction is ready to issue, return whether
     * bypassing is needed to execute the instruction this cycle (due
     * to src reg results being too new). If so, write through
     * (*outFuPoolIndex) the number of the FU Pool that this instruction
     * may execute on (or -1, if no FU Pool can execute the instruction).
     * If not, the instruction's needed inputs are available at all FU Pools.
    */
    bool
    getBypassFuPoolIndex(int* outFuPoolIndex, Cycles currentCycle) const
    {
        // assert(this->readyToIssue());
        if (currentCycle != latestSrcRegCycle) return false;
        *outFuPoolIndex = srcRegFuPoolIndex;
        return true;
    }
```

Finally, to provide the needed information for `markSrcRegReadyFuPool`, I had to add a cycle counter to the CPU class, and a variable to the instruction record type, `fuPoolIndex`, that indicates which FU pool it was executed on (and hence which FU pool is able to bypass the instruction's output value).

5 FU Pool Selection Strategies

This section describes the different strategies tested for assigning instuctions to FU Pools. In all cases, they apply only for instructions that do not require bypassing to execute – instructions requiring bypassing to execute are always assigned to the FU pool producing the bypassed inputs, if possible.

1. **Greedy:** The greedy strategy tries every FU pool in order, picking the first pool with a free FU capable of executing the instruction. This generally has the effect of saturating the pools first in order, and minimally utilizing those last to be tried.

2. **Random:** This is similar to the greedy strategy, except that the starting index for the search is randomized. FU pools are queried for a free capable FU in a circular order starting from the random index.
3. **Load Balancing:** All FU pools are queried for the number of free FUs capable of executing the instruction. The one reporting the highest number is assigned the instruction (in case of a tie, the earliest one queried is chosen).

The load balancing strategy is motivated (by the risk of congestion bypass failures) to prevent saturating an FU pool's resources, while the greedy strategy is motivated more by the risk of confluence bypass failures.

6 Results

The following simulation configuration was used to evaluate the effects on performance that partial bypassing and various FU pool selection strategies have.

- `DerivO3CPU` base out-of-order superscalar CPU, with instruction issue width 8.
- x86_64 ISA.
- 12 integer ALUs.
- 4 integer multiply/divide units.
- 4 floating point units (mostly unused due to SSE).
- 4 floating point multiply/divide units.
- 12 SIMD units (SSE).
- 4 memory read-write ports.
- 4.0 GHz CPU clock.
- `DDR3_1600_8x8` memory.
- 16 kB, 2-way associative L1 instruction cache.
- 64 kB, 8-way associative L1 data cache.
- 2048 kB, 8-way associative L2 cache.

Four workloads were used for simulation: two chosen from PolyBench: 3-matrix-multiply (`3mm`) and Cholesky decomposition; and two custom workloads: one-layer convolutional neural network (`cnn`), and game particle simulation.

Each workload was simulated on a baseline processor with complete bypassing between functional units, and on processors that evenly split the functional units into two or four FU pools. The latter configurations were tested with all three FU pool selection strategies described earlier – greedy, random, and load balancing (balance).

Each workload and CPU configuration combination reports the percentage slowdown incurred compared to the baseline CPU with complete bypassing, the percentage of instructions executed with

bypassed operands, the decrease (in percentage points) in the percentage of such bypassed instructions compared to baseline, and the number of congestion and confluence bypassing failures detected.

Note: The workloads are run by the `main()` function of `fupools.py`, and the tables generated by `write_tex_tables()`.

6.1 Convolutional Neural Network Workload

The CNN workload is characterized by aggressive optimizations for vectorization and instruction-level parallelism. Loop unrolling and fusion is performed to do work on four output feature maps in parallel in each inner loop iteration. As such there are four mostly-independent chains of instructions in-flight at any given time.

strategy	FU pools	% slowdown from complete	% insts bypassed	$-\Delta$% points bypassed	congestion (million insts)	confluence (million insts)
complete	1	0.00%	63.90%	0.00%	0.68	0.00
greedy	2	0.16%	62.51%	1.38%	3.69	2.81
random	2	0.16%	62.03%	1.87%	2.82	8.44
balance	2	0.50%	61.62%	2.28%	2.15	11.96
greedy	4	0.02%	58.81%	5.09%	15.10	9.44
random	4	0.30%	59.13%	4.77%	11.27	13.00
balance	4	0.34%	59.71%	4.18%	8.70	12.72

6.2 Particles Workload

The particles workload is representative of "typical" code written under real-world conditions and time constraints, with no particular regard given to performance. The code was sourced from a Unity3D C++ plugin that performs voxel-based collision detection and Bezier interpolation as part of a system for visualizing tumors as a particle cloud. This workload is characterized by a large number of dynamic memory allocations and data-dependent branches.

strategy	FU pools	% slowdown from complete	% insts bypassed	$-\Delta$% points bypassed	congestion (million insts)	confluence (million insts)
complete	1	0.00%	49.00%	0.00%	4.40	0.00
greedy	2	0.17%	46.82%	2.18%	15.58	1.98
random	2	1.56%	45.36%	3.64%	11.69	14.77
balance	2	2.19%	44.69%	4.30%	10.69	20.41
greedy	4	1.88%	41.75%	7.25%	40.40	10.72
random	4	2.58%	41.87%	7.13%	25.91	21.23
balance	4	2.87%	41.60%	7.39%	26.22	23.64

6.3 3-Matrix-Multiply Workload

strategy	FU pools	% slowdown from complete	% insts bypassed	−Δ% points bypassed	congestion (million insts)	confluence (million insts)
complete	1	0.00%	68.27%	0.00%	0.00	0.00
greedy	2	0.98%	64.65%	3.62%	8.11	0.00
random	2	0.62%	60.46%	7.81%	3.87	9.16
balance	2	1.90%	60.99%	7.28%	0.93	10.36
greedy	4	1.14%	54.65%	13.62%	11.82	8.93
random	4	1.36%	53.11%	15.16%	10.92	12.38
balance	4	0.59%	51.89%	16.38%	10.77	15.14

6.4 Cholesky Decomposition Workload

strategy	FU pools	% slowdown from complete	% insts bypassed	−Δ% points bypassed	congestion (million insts)	confluence (million insts)
complete	1	0.00%	60.37%	0.00%	8.32	0.00
greedy	2	0.92%	52.83%	7.54%	43.12	0.01
random	2	1.36%	54.24%	6.13%	24.40	20.60
balance	2	1.66%	54.58%	5.79%	15.97	30.20
greedy	4	1.05%	48.08%	12.29%	56.97	7.21
random	4	1.86%	47.78%	12.60%	51.21	27.24
balance	4	1.58%	47.05%	13.32%	52.56	31.36

7 Discussion

As predicted, partial bypassing has consistently lower performance than complete bypassing. Interestingly, the magnitude of the effect is highly workload-dependent, with the highly-optimized CNN workload being minimally affected by partial bypassing, and the poorly-optimized particles workload being the most sensitive.

At least with the simple strategies tested here, there is also a clear tradeoff between congestion and confluence bypass failures, with the greedy strategy having the highest number of congestion failures and the lowest number of confluence failures, the load balancing strategy typically being the reverse, and the random strategy somewhere in between (but generally closer to load balancing than greedy).

As predicted, this tradeoff means that there is no simple perfect FU pool selection strategy. Fortunately, with the best-performing strategy – greedy – the worst-case performance penalty of partial bypassing is not all that devastating: 1.88%, on the 4 FU pool, particles workload benchmark.

The only significant counterexample to the greedy strategy being best is the 3-matrix-multiply, 4 FU pool benchmark, which incurred a 1.14% performance loss with the greedy strategy, compared to a 0.59% loss with load balancing. I suggest future work running a larger sampling of workloads to determine how consistent the preference for the greedy strategy is. It would be fortunate if this pattern were consistent, as the hardware costs of the greedy strategy are likely very modest.

The percentage of instructions executed with bypassing does not perfectly correlate with performance. For example, in the 2 FU pool Cholesky decomposition benchmarks, greedy outperforms random and load balancing, despite having a lower proportion of bypassed instructions – 52.83% with greedy, compared to over 54% for the other two strategies. This weakly suggests that confluence bypass failures are in some way worse than congestion failures (but again also suggests future work running a larger sampling of workloads).

One possibility is that confluence bypass failures are more likely to occur on the critical path. This motivates, as future work, a criticality-aware FU pool selection strategy. One FU pool could be designated as a "critical" pool reserved for instructions identified as being on the critical path. The forward token passing approach described by Fields et al. could be modified to identify not only critical instructions, but also chains of instructions that feed into the critical path (with minimal "slack") that should also be assigned to the critical FU pool. This would have the effect of preventing any confluence failures from stalling the critical instructions, as all its time-sensitive dependencies will be centralized on one FU pool, while also preventing congestion failures from interfering with the critical path, by keeping unrelated instructions from congesting the critical FU pool.

As a final suggestion, gem5 may be further modified to support simulating bypass networks more general than the within-FU-pools-only bypassing networks presented in this paper. The simulations run here use four read-write ports. This makes congestion bypassing failures very likely for memory instructions when simulated with four FU pools, as each pool will have only one memory unit.[3] One could imagine alleviating this problem by simulating a network that divides arithmetic units into several FU pools, with no bypassing between pools, but with complete bypassing from all arithmetic pools to a special memory-only pool.

8 References

N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R.Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator

B. Fields, S. Rubin, and R. Bodk. Focusing processor policies via critical-path prediction

9 Appendix: More Memory Port Benchmarks

I ran additional "more memory" benchmarks, which were identical to the main benchmarks, except that the four read-write ports are replaced with an unrealistically high number of memory units: eight read ports and eight write ports, still split evenly among FU pools. These tables report the original performance comparisons alongside comparisons done on the more memory benchmarks (prefixed with mm-). Each multiple-FU-pool configuration is compared with the equivalent complete bypassing configuration (i.e. non-mm configurations are compared with complete, mm configurations with mm-complete).

Generated with `fupools.py`, `mm_write_tex_tables()`

[3]The appendix partially addresses this issue.

9.1 Convolutional Neural Network Workload

strategy	FU pools	% slowdown from complete	% insts bypassed	−Δ% points bypassed	congestion (million insts)	confluence (million insts)
complete	1	0.00%	63.90%	0.00%	0.68	0.00
mm-complete	1	0.00%	64.89%	0.00%	0.09	0.00
greedy	2	0.16%	62.51%	1.38%	3.69	2.81
random	2	0.16%	62.03%	1.87%	2.82	8.44
balance	2	0.50%	61.62%	2.28%	2.15	11.96
mm-greedy	2	0.26%	63.35%	1.54%	5.00	1.30
mm-random	2	0.15%	62.84%	2.05%	2.29	9.04
mm-balance	2	0.29%	62.72%	2.17%	1.17	10.27
greedy	4	0.02%	58.81%	5.09%	15.10	9.44
random	4	0.30%	59.13%	4.77%	11.27	13.00
balance	4	0.34%	59.71%	4.18%	8.70	12.72
mm-greedy	4	0.14%	57.37%	7.52%	24.53	6.51
mm-random	4	0.36%	59.91%	4.98%	11.68	12.84
mm-balance	4	0.47%	60.69%	4.20%	8.56	11.48

9.2 Particles Workload

strategy	FU pools	% slowdown from complete	% insts bypassed	−Δ% points bypassed	congestion (million insts)	confluence (million insts)
complete	1	0.00%	49.00%	0.00%	4.40	0.00
mm-complete	1	0.00%	49.71%	0.00%	2.13	0.00
greedy	2	0.17%	46.82%	2.18%	15.58	1.98
random	2	1.56%	45.36%	3.64%	11.69	14.77
balance	2	2.19%	44.69%	4.30%	10.69	20.41
mm-greedy	2	0.09%	48.74%	0.97%	6.12	0.81
mm-random	2	1.44%	47.04%	2.67%	4.82	14.63
mm-balance	2	2.26%	46.25%	3.46%	4.71	20.36
greedy	4	1.88%	41.75%	7.25%	40.40	10.72
random	4	2.58%	41.87%	7.13%	25.91	21.23
balance	4	2.87%	41.60%	7.39%	26.22	23.64
mm-greedy	4	1.52%	44.63%	5.09%	23.25	8.55
mm-random	4	2.36%	44.18%	5.53%	14.80	20.44
mm-balance	4	2.73%	43.99%	5.72%	14.07	23.50

9.3 3-Matrix-Multiply Workload

strategy	FU pools	% slowdown from complete	% insts bypassed	−Δ% points bypassed	congestion (million insts)	confluence (million insts)
complete	1	0.00%	68.27%	0.00%	0.00	0.00
mm-complete	1	0.00%	68.27%	0.00%	0.00	0.00
greedy	2	0.98%	64.65%	3.62%	8.11	0.00
random	2	0.62%	60.46%	7.81%	3.87	9.16
balance	2	1.90%	60.99%	7.28%	0.93	10.36
mm-greedy	2	-0.02%	68.24%	0.03%	0.00	0.05
mm-random	2	0.73%	63.28%	5.00%	0.00	7.95
mm-balance	2	1.86%	61.54%	6.73%	0.00	10.27
greedy	4	1.14%	54.65%	13.62%	11.82	8.93
random	4	1.36%	53.11%	15.16%	10.92	12.38
balance	4	0.59%	51.89%	16.38%	10.77	15.14
mm-greedy	4	0.35%	61.99%	6.28%	6.45	5.95
mm-random	4	0.79%	59.54%	8.73%	1.96	12.17
mm-balance	4	0.54%	59.93%	8.34%	0.24	13.46

9.4 Cholesky Decomposition Workload

strategy	FU pools	% slowdown from complete	% insts bypassed	−Δ% points bypassed	congestion (million insts)	confluence (million insts)
complete	1	0.00%	60.37%	0.00%	8.32	0.00
mm-complete	1	0.00%	61.31%	0.00%	3.85	0.00
greedy	2	0.92%	52.83%	7.54%	43.12	0.01
random	2	1.36%	54.24%	6.13%	24.40	20.60
balance	2	1.66%	54.58%	5.79%	15.97	30.20
mm-greedy	2	0.79%	57.50%	3.80%	17.52	0.00
mm-random	2	1.58%	57.27%	4.04%	8.78	20.51
mm-balance	2	2.21%	55.82%	5.49%	9.60	29.85
greedy	4	1.05%	48.08%	12.29%	56.97	7.21
random	4	1.86%	47.78%	12.60%	51.21	27.24
balance	4	1.58%	47.05%	13.32%	52.56	31.36
mm-greedy	4	0.80%	51.78%	9.53%	43.95	4.64
mm-random	4	1.94%	52.59%	8.72%	24.94	26.46
mm-balance	4	1.46%	50.74%	10.57%	24.78	28.34