REEL: Reducing Effective Execution Latency of Floating Point Operations

Vignyan Reddy†, Syed Zohaib Gilani†, Erika Gunadi†, Nam Sung Kim†, Michael J Schulte** and Mikko H Lipasti†
†University of Wisconsin-Madison **AMD Research
{kothintinare, gilani, egunadi}@wisc.edu, michael.schulte@amd.com, {nskim,mikko}@engr.wisc.edu

***Abstract*—The height of the dynamic dependence graph of a program, as executed by a processor, determines the minimum bound on the execution time. This height can be decreased by reducing the effective execution latency of operations that form dependence chains in the graph. In this paper, we propose a technique called REEL to reduce overall latency of chains of dependent floating point (FP) operations by increasing the throughput of computation. REEL comprises of a high-throughput floating point unit (HFP) that allows early issue of an FP Add that is dependent on another FP Add or FP Multiply. This is complemented by instruction scheduler modifications that allow early issue of dependent FP Adds, and a novel checker logic that corrects any precision errors. Unlike conventional static operation fusion, like fused Multiply-Add (FMA), there are no changes to the instruction set to enable utilization of the new hardware, and no recompilation is necessary. Furthermore, unlike ISA-level FMA, our technique produces results that are bit compatible while boosting performance of Add-Add dependence pairs in addition to Multiply-Add pairs. Our evaluation of REEL using CFP2006 benchmarks shows an average performance gain of 7.6% and maximum performance gain of 17% while consuming 1.2% lower energy.**

I. Introduction

The minimum execution time of a program is equal to the height of its data flow graph which is determined by the instructions in its critical execution path. Extending Little's law to processors, the average instructions per clock (IPC) is determined by average instruction arrival rate and average time an instruction spends in the instruction window (τ). The effective latency of instructions contributes significantly to τ and decreasing this latency can improve processor performance.

Many techniques exist to reduce effective latency of operations. Clever circuit design, transistor-level optimization, and novel designs can reduce the delay of functional units. Pipelining allows instructions to start execution on a functional unit, while a part of this unit is used by other independent instructions. This increases the throughput of the functional unit, thus reducing the effective latency of a group of instructions having some independence amongst them. However, the height of the data flow graph, which is determined by chain of dependent instructions, is not reduced by pipelining.

Operation fusion is a common technique used to reduce the delay of dependent instruction pairs. Static operation fusion is done at compile time by combining common occurrences of dependent instructions into a single instruction. Fused Multiply-Add (FMA) instruction is an example of static fusion that combines Multiply and Add operations. Alternately, instructions can be fused during runtime, like the macro-op fusion in the decode stage of Intel Pentium M processor [1].

In this paper, we propose a novel technique, called REEL, to reduce the effective latency of floating point instructions. This decreases the average time spent by an instruction in the execution window and increases throughput.

This paper makes the following contributions:

1) We propose a novel technique to execute dependent chains of FP instructions (FP Adds dependent on FP Multiplies **and** FP Adds) faster by using a combination of special hardware and instruction scheduling changes. Our special hardware, the high-throughput floating point (HFP) unit, builds on the well-known advantages of FMA, but significantly enhances them. Along with performance boost, our technique has these unique advantages:
   a) Program binary compatibility : Our technique works with program binaries that may or may not have FMA instructions. No compiler support is required for our technique.
   b) Result bit compatibility : FMA instructions generated by a compiler compromise bit-exact results, due to removal of the intermediate rounding step. Our technique maintains bit-exact results and still boosts performance. Additional performance can be gained for some workloads if this result bit compatibility is not required.
   c) FP Add-Add dependence : Dot products, matrix multiplications and other common computational kernels contain chains of dependent FP Adds. HFP benefits such dependence chains in addition to FP-Multiply to FP-Add dependences.
   e) In-ALU registers(IARs) : IARs store intermediate values of multiple instructions present in the HFP pipeline. These registers enable flexible scheduling of multiple concurrent dependence chains and thus improves performance significantly. IARs also ease instruction scheduling by relaxing the precise schedule cycle requirement for dependent instruction that is otherwise required to benefit from the proposed technique.
   f) Register access ports : Unlike other operation fusion techniques, the proposed technique does not require any additional register read or write ports.
2) The performance, area, and energy costs of this technique are evaluated using detailed physical design and cycle accurate simulations.

The rest of the paper is organized as follows. We explain REEL in Section 2 and discuss its design characteristics in Section 3. Section 4 details the evaluation setup, results and analysis. A subset of related work is presented in Section 5 and conclusions of this research are provided in Section 6.

II. REEL

A. Overview

FP arithmetic operations like addition, multiplication and division have long latencies and typically take multiple cycles to execute. FPUs are usually pipelined to increase the throughput by executing independent FP instructions simultaneously. However, dependent FP instructions still need to wait for multiple cycles. Such dependency chains stall the front end and limit the instruction arrival rate resulting in reduced instruction level parallelism. Our analysis of the CFP2006 benchmarks from SPEC2006 show that there are considerable FP Adds dependent on FP Multiply and other FP Adds. Details of this analysis are provided in Section 4. Note that we use Add to refer to either addition or subtraction. Also, all references to Adds, Multiplies and fused Multiply-Add (FMA) are FP unless explicitly specified.

Assume that a Multiply, Add and a native FMA instruction execute in 5, 5 and 6 cycles respectively. Executing a native FMA instruction in lieu of a dependent Multiply-Add pair (Multiply followed by a dependent Add) reduces the overall latency by 4 cycles. In effect, the multiply part of the native FMA has an effective latency of 1 cycle. However, a dependent Add-Add pair or a dependent FMA-FMA pair will still have overall latencies of 10 and 12 cycles. Table III in Section 4 shows that dependent Add-Add pairs (labeled as % of HFADDs in Table III) are quite frequent in some benchmarks. While the ISA can be augmented with instructions to execute Add-Add pairs, we discuss in Section 4 on why REEL is a better choice.

REEL employs a high-throughput floating point (HFP) unit and incorporates appropriate changes to the instruction scheduler. REEL aims to reduce effective latencies of Multiply and Add instructions that have dependent Add instructions. The HFP unit is a specially designed FPU that can execute a Multiply, Add or a native FMA instruction. In addition, the HFP unit is novelly augmented with "In-ALU Registers" (IARs). These IARs hold the intermediate value (the result before normalization and rounding) of the Multiply or Add operation and can supply it to a dependent Add instruction.

The updated scheduling logic is aware of the IAR used by the parent instruction and tags the dependent instructions to use the same IAR. In the baseline, the scheduling logic issues a dependent Add after the execution latency of issuing the parent Add or Multiply. The new scheduling logic issues a dependent Add as early as the next cycle. This reduces the effective latency of the parent operation. We refer to the early scheduling of Adds (there can be more than one) dependent on a Multiply as HFMA. Similarly, we refer to early scheduling of Adds dependent on a Add as HFADD. Note that HFMA and HFADD both have dependent FP Adds. Occurrences of other FP operation dependences are much less frequent and thus not a focus in our design or experiments. In the following subsections, we explain various aspects of REEL.

B. HFP Unit

The HFP unit is the basis for our technique. It is derived from the FMA units proposed in [2] and [3], but significantly modified to support our technique. Figure 1 shows a block diagram for the HFP unit. The diagram assumes the input operands are double precision, which are 64-bits, but the design can easily be modified for single-precision operands or be extended to handle both single and double precision operands. The design contains 5 pipeline stages and are marked as S0 through S4 in the figure. A Multiply, Add and native FMA take 5 cycles to complete execution. Multiply operation receives its inputs from ports A and B while Add operation receives them on ports A and C. A native FMA operation will receive its inputs on all three ports to compute $A * B + C$.

The parent operation's intermediate result before rounding and normalization is stored in the IARs. A dependent operation can obtain the required operand value from the IARs. This value can be immediately forwarded to the Addition block in Figure 1 to be used by depedend Adds. However, if the absolute difference of exponents of the operands being added is larger than 53, the value from IAR may not be used. This limit ensures that the 160-bit adder employed in the HFP unit does not lose any accuracy in the result. If the exponent difference between the value on port C and value forwarded from IAR is 53, the significand of the operand C is aligned to bits [158:106] of the adders. The MSB of the adders is reserved for the sign bit. The 106-bit product result always occupies the bits [105:0] of the adders. For an exponent difference of 0, the significand of value on port C is aligned with the MSBs of the value forwarded from IAR. Finally, for an exponent difference of -53, the significand of value on port C is aligned with bits [52:0] of the adders. Consequently, within this exponent range, we can utilize the intermediate result without losing any bits. If the exponent difference is larger, the significand of value on port C will lose some bits from the MSB side or the LSB side.

The exponent difference is calculated during S1 by the ESH block. If the exponent difference exceeds the given range, the dependent Add instruction is cancelled and reissued after the parent instruction completely finishes execution to generate the result in the standard FP format. Such cases of exponent differences violating the range of -53 to +53, while present, are rare in the SPEC 2006 benchmarks.

C. Scheduling Logic and In-ALU Registers

While the changes to instruction scheduling logic enables use of HFP unit, IARs optimize its benefits. In the baseline processor, the instruction scheduler issues the oldest ready instructions. When an FP operation (Multiply or Add) is issued, the modified instruction scheduler will speculatively issue a dependent Add, that is otherwise ready. Since every operation scheduled on the HFP unit writes an IAR, the scheduler can select any Add instruction that is dependent on any operation still executing on HFP unit. This enables simpler select decisions and minimal changes to scheduling logic.

As observed from Figure 1, HFP unit can internally bypass only one value to the dependent instructions. For this reason, the modified instruction scheduler successfully issues a dependent Add only if atleast one of its operands is readily available from either the register file or the bypass network. This dependence conflict can also cause cancellation of the speculatively issued instruction. If there is more than one unit, the port binding logic in dispatch stage is also modified to bind the instruction to the same port as the parent operation.

IARs aid in simplifying the scheduling logic, increasing number of dependent Adds that can be scheduled early, and decreasing register read energy. Some of such scenarios are:

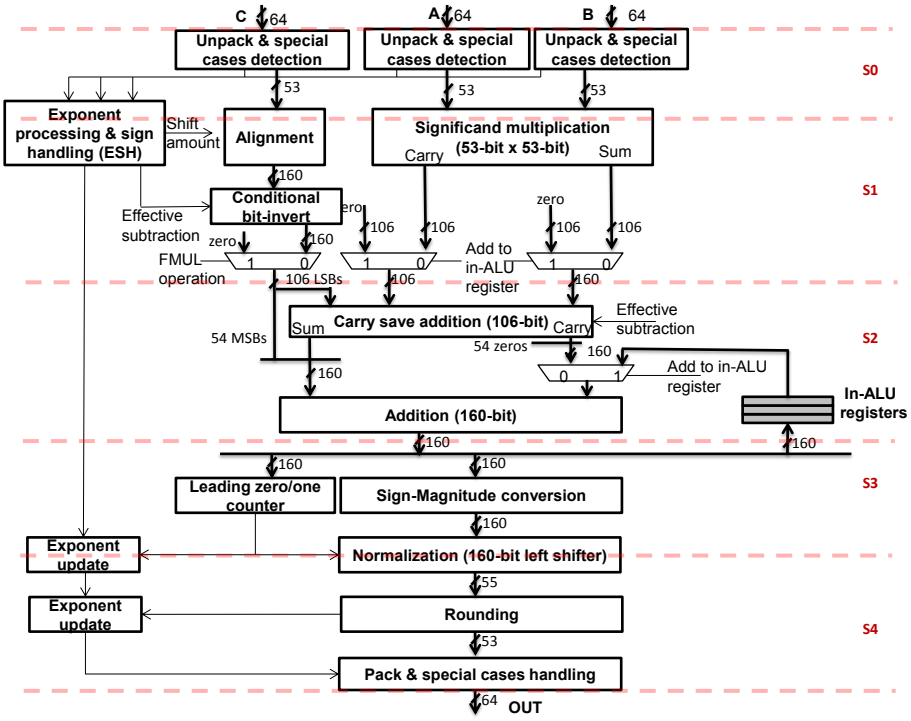

Fig. 1: Conceptual block diagram of the HFP unit

1) The precise schedule cycle requirement for issuing a dependent Add is relaxed because of IARs. This allows more dependent Adds to be issued while simplifying the instruction scheduling logic.
2) If there is more than one dependent Add, all of them can be issued in successive cycles to use the IAR written by the parent FP operation. Though the latency benefits decrease with increased cycle gap between the parent and dependent instructions, the effective latency of multiple dependent operations is now reduced. Note that there is no latency benefit once the parent instruction completes execution.
3) When used in a processor with simultaneous multi-threading (SMT), Multiplies or Adds from different threads can map to different IARs enabling latency benefits for both threads. Here we are only identifying the potential of using IARs. Our analysis does not include SMT scenarios.
4) The number of reads from the FP register file decreases as the value is provided by the IARs. This decreases the FP register file read energy.

I1	FR2 ← FR1*FR0
I2	FR4 ← FR2+FR3
I3	FR6 ← FR4+FR5
I4	FR6 ← FR6+FR5

(a) Example instructions

Stage →	S0	S1	S2	S3	S4
Cycle 0	I1				
Cycle 1	I2	I1			
Cycle 2	I3	I2	I1		
Cycle 3	I4	I3	I2	I1	
Cycle 4		I4	I3	I2	I1
Cycle 5			I4	I3	I2
Cycle 6				I4	I3
Cycle 7					I4

(b) HFP Execution

Fig. 2: Executing Multiply and dependent Add instructions.

Figure 2b shows the schedule of example dependence chain from Figure 2a when using the HFP unit. Data forwarding between different instructions via IARs is highlighted in the figure. The intermediate result of Instruction I1 is stored in an IAR, say IAR0. Instruction I2 is issued in cycle 1 and reads the result of the product from IAR0. The result of I2 is also stored in a different IAR, say IAR1. This value in IAR1 is used to compute the instruction I3 that is issued in cycle 3. Similarly, intermediate value of I3 is used by I4 via a different IAR, like IAR2. Overall, for this example, using REEL reduces execution cycle count from a baseline of 16 cycles to 8 cycles.

D. Result Binary Compatibility:

An attractive quality of REEL is the performance gain while preserving the result binary compatibility (RBC). This would not be possible with an ISA extension like the native FMA instruction. RBC is achieved by using a rounding predictor. When storing the intermediate value into an IAR, the value is floored by clearing the lower order bits. This is a static prediction that rounding will result in a *floor* operation. The unaltered intermediate value is sent down to normalize

and rounding stages. If rounding stage performs a *ceil* instead of *floor*, the rounding prediction was incorrect. All dependent operations that are in flight are cancelled and marked for re-execution. An event counter can be used to disable this technique if too many rounding mispredictions occur.

An implementation that does not have a requirement of result binary compatibility can gain additional performance by disabling the rounding prediction and recovery. Some benchmarks show the additional performance gains when RBC is not required. We present more details on this in Section 4.

E. Benefits

Scheduling dependent Adds dynamically and using IARs to store intermediate values maintains program binary compatibility, RBC and precise exception semantics, while extracting the maximum possible benefit. In contrast, recompilation or binary translation would miss some opportunities due to intervening branches, precise exception semantics, or live fanout from the parent operation.

If ISA were augmented with a new instruction that can add three operands, it will collapse only one dependent Add-Add pair per instruction. The dependence chain between the two new instructions still exists. With REEL, the entire chain of dependent Add-Add pairs is collapsed (as shown in example in Figure 2b).

A small limitation of this technique is that all FP instructions still must reside in the issue queue till they complete. Benefits from lowering the machine occupancy are not gained with this technique.

III. Design Analysis

The optimal implementation of REEL will depend on the target machine, but this section presents some design details of our implementation. Our implementation has a single register file read stage between issue and execute. This is consistent with the recent trend toward relatively shallow pipelines. If future designs return to deeper register file pipelines, cancellation of dependent operations will be more challenging, and may reduce the performance advantage of the proposed technique. In this case, a watchdog counter can be employed to turn this feature off if too many uop cancellations are seen.

In our evaluation, IARs are implemented as a FIFO of depth five. FIFO implementation allows easy management of the IARs. Once the FP operation writing to an IAR finishes, the corresponding IAR becomes invalid and ready for reuse. The IAR tagging for dependent instructions also becomes simple in the scheduler. Scheduler keeps track of the distance between the parent and dependent operation to tag the IAR for the dependent operations' use. A FIFO with five entries, equal to the pipeline stages in the HFP, enables capturing values of different operations executing in the HFP pipeline. This enables early scheduling of any instruction dependent on any instruction execution on HFP and thus optimizes performance. IARs are a part of the speculative state and not the architectural state. On an interrupt or an exception, these registers are just cleared like any other speculative state of the machine.

We used Synopsys Design Compiler with TSMC 65nm standard cell library for obtaining all the synthesis results in this paper. The power estimates were obtained using Synopsys Power Compiler. The synthesis tools can move the location of pipeline registers to optimize for delay and area. Table I shows a comparison of area and power of an FP multiplier, an FP adder and a HFP unit. The pipeline stages assumed for each of these units are also shown in the Table I. The timing goal of 2GHz is met in all three units. There is a small increase in the area and power of the HFP unit due to the wide adder and wide shifter. However, if a native FMA unit is also included in the baseline, HFP unit can actually save on area and power.

	Pipelines	**Power**(mW)	**Area**(μm^2)
FP MUL	4	33.28	33377
FP ADD	4	13.42	18331
Total		46.7	51708
HFP	5	47.87	52517

TABLE I: Area and Power estimates of FP multiplier, FP adder and HFP units.

	Machine Config
OoO structures	4-wide fetch/commit, 6-wide issue, 128 ROB, 36 IQ, 48 LQ, 32 SQ, 96 Int-PRF, 96 FP-PRF, 11-stage pipeline, speculative scheduling with squashing recovery, aggressive memory reordering with store set predictor (4k ssit, 128 lfst) and flush recovery
Branch Predictor	Combined bimodal(16k entry)/gshare(16k entry) w/ selector(16k), 32 entry RAS, 4 way 2k-entry BTB
Integer ALUS	3 (1-cycle)
FP ALUs	2 FP add (5-cycles) and 2 FP multipliers (5-cycles)
Common units	1 integer multiplier (2-cycles), 1 divider (4-cycles), 1 Load (1+2cycles), 1 Store address (1-cycle), 1 Store data (1-cycle), 1 FP divider/square-root (12-cycles)
Memory System (Latency)	L1 I-cache: 64KB, 2-way, 64B line size (2-cycles); L1 D-cache: 32KB, 4-way, 64B line size (2-cycles); L2 Unified: 2MB, 8-way, 128B line size (12-cycle); Off-chip memory: (168-cycles); 32-entry prefetch buffer, stream prefectching on DL1 miss

TABLE II: Baseline machine configuration used in our evaluations.

IV. Results

Table II provides the configuration of our baseline machine. The baseline processor is configured to model a realistic modern desktop processor core. The SPEC2006 benchmark suite and its inputs are intended to represent real world programs and applications [4]. We evaluate REEL with CFP2006 benchmarks that are compiled with gcc-4.6 and "-O3 -m64 -static -fpmath=sse" flags. The Pinpoint tool is used to get *simpoints* for each of these benchmarks [5], [6]. Instruction traces of size 14 Million instructions are generated at non-trivial simpoints (determined by simpoint weights) of each benchmark by using a custom Pin tool [7]. These instruction traces hold all necessary information including the register and memory read values for each instruction. Timing analysis is

CFP2006 benchmarks from SPEC2006			
Benchmark	**IPC**	**% of HFMAs**	**% of HFADDs**
$bwaves$	1.47	2.9	4.7
$cactusADM$	1.08	5.7	8.7
$calculix$	2.09	18.5	4.6
$dealII$	1.71	4.4	1.9
$gamess$	2.31	4.4	7.1
$GemsFDTD$	0.53	3.6	11.9
$gromacs$	1.92	6 .0	4.4
lbm	0.67	16.9	15.4
$leslie3d$	0.56	4	1.4
$milc$	0.57	5.3	12.3
$namd$	2.17	6.3	6.1
$povray$	2.12	1.3	3.9
$soplex$	0.39	1.4	0.5
$tonto$	2.03	2.7	4.9
$zeusmp$	0.90	3.5	1.0

TABLE III: Runtime statistics of the baseline: IPC, percent of HFMAs and HFADDs executed for CFP2006 benchmarks.

performed on the 10 million instructions after warming up the branch predictors, caches for 4 million cycles. Instructions from the trace are cracked into micro operations (uOp) that run on a cycle accurate simulator. Various activity counts recorded by this simulator are used in this analysis. Table III summarizes the weighted IPC of the baseline and the available opportunity of HFMAs and HFADDs.

As observed from Table III, most CFP2006 benchmarks have a generous mix of HFMA and HFADD operations. These operation counts represent number of Add instructions that get one of their operands from the IARs instead of register file or bypass network. These instructions are the ones that benefit from REEL. Note that this opportunity is determined by the scheduler logic and available instruction level parallelism of FP instructions. The absolute opportunity may be larger than what is reported.

Figure 3 shows the relative increase in performance of the baseline processor when using REEL. This figure also shows the performance of REEL when result binary compatibility (RBC) is not required. On an average, using REEL boosts performance by 7.5% with *GemsFDTD* having the maximum gains of 17%. Almost all the benchmarks have their speedups corelating to their respective opportunities. Some benchmarks like *bwaves*, *dealII* and *leslie3d* show low improvement compared to number of Adds scheduled earlier. We identified that memory performance limits gain of additional performance. REEL also boosts performance due to earlier resource reclaimation. This effect is pronounced with *GemdFDTD* which is sensitive to physical register size.

When using REEL, benchmarks with significant opportunity showed an increase in front-end stalls due to the limited issue queue size. In our evaluations, we kept the issue queue size constant for fair evaluation with the baseline. However, we envision that practical implementations would re-evaluate the issue queue size to ensure optimal design.

As shown in Figure 3, the average (geometric mean) performance gain, when RBC is not required, increases to 8.6%. Notably, *calculix* and *GemsFDTD* have a lot of rounding mispredictions, which can reduce REEL's performance when RBC is required. The reported performance is dependent on data but we note that there is enough confidence in basic rounding predictor. A more sophisticated static rounding predictor can be used in lieu of the proposed *always floor* predictor. This would enable REEL with RBC to achieve performance closer to REEL without RBC.

The energy consumption of the different components is collected from synthesized designs. Combined with the event counts from the simulator, the net energy of the backend pipe is obtained for each of the benchmarks. The energy of each Add or Multiply is now increased by 1.87pJ. Across all benchmarks, REEL uses 1.5% more ALU energy while reducing register read energy by 6.4%. Overall, REEL consumes about 1.2% lower energy compared to the baseline processor. Furthermore, the improvements in performance can easily be converted to energy savings by applying dynamic voltage and frequency scaling.

V. Related Work

In this section, we will discuss some related prior work that inspired us with our proposal. Multiply and Accumulate is a common operation done in many signal processing applications. Prior research has shown that processors have performance and power benefits using a fused multiply add (FMA) block for myriad of applications [2], [8], [9], [3]. FMA blocks have been implemented in many commercial processors by different companies in different instruction sets [10], [11], [12].

Combining a Multiply and its dependent Add into a single fused multiply-add (FMA) reduces the effective latency of the Multiply. This has been explored previously [2], [8], [9], [3]. The latency benefit comes from eliminating the post-multiply rounding and normalization. The product is thus immediately used for addition and the final result (after addition) is rounded and normalized. FMA is typically introduced as an instruction set extension and requires programs to be recompiled. This may result in compatibility issues when new code has to run on older machines. Recompilation can be avoided by dynamically fusing a multiply with a dependent add, similar to the macro-op fusion used in the Intel Pentium M processor [1].

Macro-op fusion has the potential to dynamically extract more opportunities of generating an FMA operation than the static analysis performed by the compiler. Additionally, dynamic fusion benefits over static if the multiply result is used by more than one dependent instruction where the compiler will typically not utilize the FMA instruction. On the downside, fusing the micro-ops that are not consecutive in program order requires complex recovery logic to handle precise exceptions. If the fusion is restricted to consecutive pairs, the potential benefit could reduce significantly. Advanced compilers schedule independent instructions in between dependent long-latency operations to hide the delay, complicating the implementation of dynamic fusion.

Both the dynamic and static approaches mentioned above may utilize an additional read port to read the three inputs required by an FMA operation. Additionally, if the multiplier output is also desired, an additional write port may also be utilized. While these concerns are far from insurmountable, they reduce the attractiveness of dynamic fusion as an approach for exploiting FMA hardware.

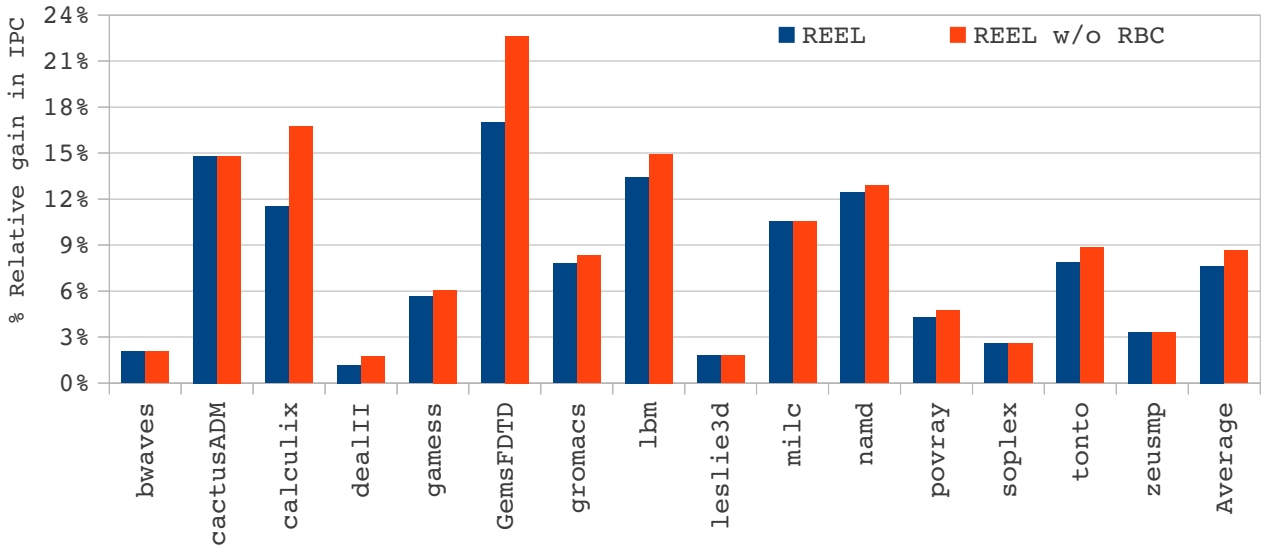

Fig. 3: Relative performance of REEL with and without bit compatibility.

Dally proposes to use the intermediate values in FP arithmetic computations to reduce the impact of FP operation latency [13]. He proposes implementing FP operations as integer micro operations. Intermediate results of these micro operations can be used to execute the dependent FP micro-operations on a superscalar processor. The compiler is aware of these micro-operations and optimizes the code by removing redundant normalizations. Gilani et al., have proposed hardware to forward the intermediate result of one FMA operation to a dependent FMA operation [2]. The machine ISA is augmented to enable the compiler to encode the information that enables early scheduling the dependent FMA instructions.

VI. Conclusion

In this paper, we demonstrate that reducing the height of the dynamic dataflow graph by reducing effective execution latency can provide significant performance benefits. We propose a novel technique called REEL to reduce execution latency. REEL comprises of a HFP unit that allows scheduling dependent FP adds before parent operation finishes. One HFP unit replaces an FP multiplier and an FP adder, and contains internal registers called *In-ALU registers* which store intermediate values of FP Multiply and FP Add. These IARs are visible to the issue logic, allowing earlier and flexible issue of dependent FP Adds, thus reducing the effective latency of the parent FP operation. The net effect of using HFP is significant improvement in performance with some energy savings, resulting in excellent energy-delay product.

VII. Acknowlegements

This work was supported in part by generous grants from Advanced Micro Devices, IBM, and the National Science Foundation (CCF-095360). Nam Sung Kim has a financial interest in AMD.

References

[1] 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*, 07(2):21 –36, february 2003.

[2] S.Z. Gilani, N.S. Kim, and M. Schulte. Energy-efficient floating-point arithmetic for software-defined radio architectures. In *ASAP- 2011*.

[3] E. Quinnell, E.E. Swartzlander, and C. Lemonds. Bridge floating-point fused multiply-add design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, Dec. 2008.

[4] Standard Performance Evaluation Corporation. Spec cpu 2006. *www.spec.org/cpu2006*, 2011.

[5] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.

[6] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT*, 2001.

[7] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[8] H. Sun and M. Gao. A novel architecture for floating-point multiply-add-fused operation. In *Proceedings of the 2003 Joint Conference of the Fourth International Conference on Information, Communications and Signal Processing and Fourth Pacific Rim Conference on Multimedia.*

[9] J.D. Bruguera and T. Lang. Floating-point fused multiply-add: reduced latency for floating-point addition. In *ARITH-17*, 2005.

[10] E. Hokenek, R.K. Montoye, and P.W. Cook. Second-generation RISC floating point with multiply-add fused. *Solid-State Circuits, IEEE Journal of*, oct 1990.

[11] Intel Corporation. Intel(R) advanced vector extensions programming reference. *www.intel.com*, 2011.

[12] M. Butler, L. Barnes, D.D. Sarma, and B. Gelinas. Bulldozer: An approach to multithreaded compute performance. *Micro, IEEE*, 31(2):6 –15, march-april 2011.

[13] W.J. Dally. Micro-optimization of floating-point operations. In *ASPLOS-III*, 1989.