Exploring Early and Late ALUs for Single-Issue In-Order Pipelines

Alen Bardizbanyan and Per Larsson-Edefors
Chalmers University of Technology, Gothenburg, Sweden
alenb@chalmers.se, perla@chalmers.se

***Abstract*—In-order processors are key components in energy-efficient embedded systems. One important design aspect of in-order pipelines is the sequence of pipeline stages: First, the position of the execute stage, in which arithmetic logic unit (ALU) operations and branch prediction are handled, impacts the number of stall cycles that are caused by data dependencies between data memory instructions and their consuming instructions and by address generation instructions that depend on an ALU result. Second, the position of the ALU inside the pipeline impacts the branch penalty. This paper considers the question on how to best make use of ALU resources inside a single-issue in-order pipeline. We begin by analyzing which is the most efficient way of placing a single ALU in an in-order pipeline. We then go on to evaluate what is the most efficient way to make use of two ALUs, one early and one late ALU, which is a technique that has revitalized commercial in-order processors in recent years. Our architectural simulations, which are based on 20 MiBench and 7 SPEC2000 integer benchmarks and a 65-nm postlayout netlist of a complete pipeline, show that utilizing two ALUs in different stages of the pipeline gives better performance and energy efficiency than any other pipeline configuration with a single ALU.**

I. Introduction

The need for energy-efficient computing has further amplified the differentiation between processor pipeline organizations: While complex out-of-order pipelines are certainly needed to sustain extreme single-thread performance, energy-efficient in-order pipelines are essential components for a wide array of situations when performance requirements are more relaxed [1]. Since the simplicity of the in-order pipeline makes it attractive for many-core processors, in-order pipelines become enablers when parallelism is to be translated to performance [2]. Interestingly, there are efforts in industry to match the performance of out-of-order processors using in-order pipelines with techniques like dynamic-code optimization [3].

While their simplicity in terms of instruction handling reduces energy per cycle over their out-of-order counterparts, in-order pipelines suffer from the fundamental disadvantage of stall cycles due to data dependencies. Here, the access stage of the arithmetic logic unit (ALU) in the pipeline has a very big impact on the number of stall cycles. Previous work has defined two different configurations of in-order pipelines and explored them in terms of cycle count [4]: The two different configurations are load-use interlock (LUI) and address-generation interlock (AGI). In LUI, the ALU is accessed in the conventional execute stage of the pipeline in which address generation is also handled. This pipeline stalls if there is an immediate dependency between a load instruction and an ALU instruction. In AGI, the ALU is accessed in the end of data memory access stage. This way all dependencies between load operations and ALU operations are eliminated. But since ALU operations happen late, stall cycles will be caused if the address generation unit (AG) needs the result of the ALU immediately. In addition, branch resolution is delayed which increases the misprediction latency.

Designing a CPU pipeline is a complex process that involves many different tradeoffs, from cycle performance at the architectural level to timing performance at the circuit level. With the ultimate goal to improve performance and energy metrics, this work is concerned with design tradeoffs for the ALU resources in a single-issue in-order pipeline. The choice of single-issue functionality is motivated by the fact that we want to explore how to best use ALU resources in the simplest possible pipeline configuration, for which each design tradeoff will have a very noticeable impact. Promising tradeoffs are likely to be applicable to wider issue pipelines. Our contributions include:

- Beside the LUI and AGI pipeline configurations [4], we also evaluate an intermediate configuration in which the ALU is accessed after the execute stage and before the end of the data memory access stage.
- We evaluate an approach to in-order pipeline design that has very recently been utilized in industry: The approach involves two ALUs in the pipeline, namely an early and a late ALU [5]. This approach tries to simultaneously leverage the benefits of LUI and AGI pipelines, while reducing the disadvantages.
- Methodology wise, we use a placed and routed 7-stage in-order pipeline, which is inspired by a recent commercial implementation. The access to a physical implementation allows us to significantly advance the previous evaluation [4] by not only considering tradeoffs that concern execution time but also energy dissipation.

We will first review the methodology we used for evaluating the different pipeline configurations. Next, we will present an overview of energy per pipeline event for the default LUI pipeline configuration. Then we will present and comment on the other pipeline configurations, which all will be normalized to the LUI pipeline, after which results are shown.

II. Evaluation Methodology

In order to explore different pipeline configurations with respect to energy, we developed an RTL model of a 7-stage in-order MIPS-like LUI-type pipeline without caches. The processor configuration of the RTL model and the cache/memory

978-1-4673-7166-7/15/$31.00 ©2015 IEEE

matches exactly the pipeline configuration in Table I which was used during architectural simulations in SimpleScalar [6].

TABLE I
PROCESSOR CONFIGURATION FOR ARCHITECTURAL SIMULATIONS.

BPR, BTB	bimod, 128 entries
Branch penalty	6 cycles
ALUs & MUL	1
Fetch & issue width	1
L1 DC & L1 IC	16kB, 4-way, 32B line, 2 cycle hit
L2 unified cache	128kB, 8-way 32B line, 12 cycle hit
Memory latency	120 cycles

Fig. 1 presents a simplified diagram of the 7-stage pipeline used for the evaluations. The stages are as follow: Instruction fetch 1 (IF-1), instruction fetch 2 (IF-2), instruction decode (ID), execute (EXE), data cache access 1 (DC-1), data cache access 2 (DC-2), and write-back (WB). In addition, the multiplier is divided in three stages, namely multiplier 1 (mult-1), multiplier 2 (mult-2), and multiplier 3 (mult-3). The multiple stages used for multiplication, instruction fetch and data cache accesses are important for performance reasons; if fewer stages are used these operations limit the pipeline's maximal clock rate significantly. The address generation unit (AG) and ALU reside in the EXE stage.

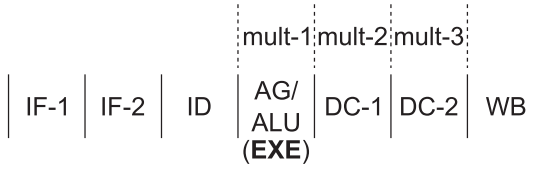


Fig. 1. Stage diagram of the 7-stage pipeline.

Based on a commercial 65-nm 1.2-V low-power process technology, the pipeline was synthesized using Synopsys Design Compiler [7], while the place and route steps were carried out using Cadence Encounter [8]. The postlayout pipeline was operational up to a clock rate of 675 MHz under worst-case conditions, i.e., worst-case process corner, 1.1-V supply voltage, and 125 °C operating temperature.

In order to verify the complete pipeline and to extract energy values for different pipeline events, we used five different benchmarks from the EEMBC benchmark suite [9]; since these lack system calls, they are practical for RTL logic simulation. All benchmarks were successfully run until completion on the postlayout netlist, in the process generating node-accurate switching activity information. Synopsys PrimeTime [10] was used to obtain the final energy values per pipeline event; for this power analysis, typical conditions were assumed, i.e., typical process corner, 1.2-V supply voltage, and 25 °C operating temperature. Since the EEMBC benchmarks are too small to generate sufficient data for our pipeline design trade-off analysis, the energy values were backannotated to SimpleScalar in order to evaluate larger and more representative benchmarks from the MiBench [11] and SPEC CPU2000 [12] (SPEC2000int) integer benchmark suites.

The MiBench and SPEC2000int benchmarks were compiled using the GCC-MIPS compiler with -O2 optimization flag. The compiler used a scheduling algorithm optimized for a MIPS R2000 type of pipeline [13]. For this type of pipeline, the scheduler tries to move ALU operations away in time from the load operations to which they have dependencies. This is, however, not the optimal case for some of the pipeline configurations that are evaluated in this paper. Hence, for the pipeline configurations that can have address generation dependencies with ALU operations, the instructions are dynamically analyzed to assess the possibility of moving load/store operations away from the dependent ALU operation. This way, scheduling is optimized for every different pipeline configuration (Sec. IV).

III. ENERGY ANALYSIS OF THE 7-STAGE PIPELINE

To establish a reference case to which we will relate our subsequent results, we here present an energy analysis of the default 7-stage LUI-type pipeline. Table II shows the energy dissipation of different pipeline stages, i.e., energy dissipated on each active cycle of a pipeline stage. Apart from the multiplier (MULT), the other pipeline stages are always active as soon as there is no stall cycle. The multiplier is only active during multiplication operations. The last value in Table II shows the energy dissipated in top-level clock buffers. The local clock buffers inside the pipeline stages are included in the total energy of the corresponding stage. The overall clock energy is 13.5 pJ per cycle. As expected, the multiplier and the branch predictor dissipates the highest energy in the pipeline.

TABLE II
PIPELINE ENERGY BREAKDOWN.

Stage or unit	Energy (pJ)
IF-1	1.9
IF-2	1.75
ID (Bpr / Ctrl / RF)	40.9 (24.5 / 11.0 / 5.4)
ALU / MULT	10.3 / 56.5
DC-1	1.25
DC-2	1.31
Clock buffer	6.5

The energy dissipation for level-1 instruction (L1 IC) and level-1 data (L1 DC) caches are estimated using the SRAM library files provided by the manufacturer. These libraries are prepared with the postlayout results and, hence, these models are very accurate. Each access to a 16kB 4-way associative L1 IC or L1 DC costs 167 pJ of energy. We are omitting the controller energy in this work, but it is usually a very small part of the cache energy. The clock network energy dissipated in caches is 5 pJ in total when they are not activated. This value, added together with the clock network energy of the pipeline per cycle, is assumed to be dissipated on each stall cycle in the architectural simulations. Thus, it is assumed that the processor dissipates 18.5 pJ of energy for each stall cycle.

We assume sequential access optimization for the L1 IC in which the program counter (PC) value is tracked to see if the next access is going to be to the same cache line. Thus, the way information for most of the IC accesses can be determined and only a single data way of the L1 IC is activated [14]. During a single way activate only 26.5 pJ energy is dissipated instead of 167 pJ. By accessing the L1 IC conventionally whenever the PC value comes from the branch predictor, the IC access

scheme becomes very straightforward; furthermore, it does not affect the critical path. This optimization is essential, otherwise the L1 IC energy will be unrepresentative and dominate the overall energy. Since our exploration does not have any impact on data accesses and since data accesses are less frequent compared to instruction accesses, this work assumes no L1 DC optimizations.

IV. Pipeline Configurations and Dependencies

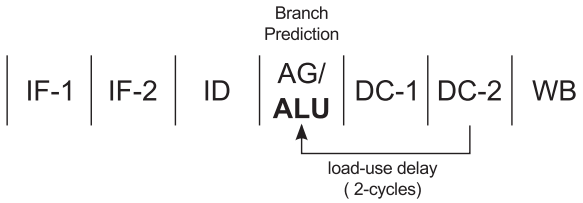
Fig. 2 shows the four different pipeline configurations that are evaluated in this work. It should be noted that this work is not concerned with the multiplier: Our evaluations show that stall cycles caused by dependencies between load operations and multiplication operations in a conventional 7-stage pipeline (see Fig. 1) are only 7% of the stall cycles caused by the data dependencies between load operations and ALU operations. This indicates that the data dependency stalls of the multiplier operations are almost negligible. As a result, this work focuses on the integer ALU which also handles branch resolution.

Fig. 2a represents the reference LUI-type pipeline which is similar to the MIPS R2000 processor [13] insofar as the ALU is accessed directly in the execute stage in which address generation is also handled. This pipeline configuration has a 2-cycle load-use delay, which causes stall cycle(s) if an ALU operation is dependent on a load operation and the distance between them is less than three cycles.

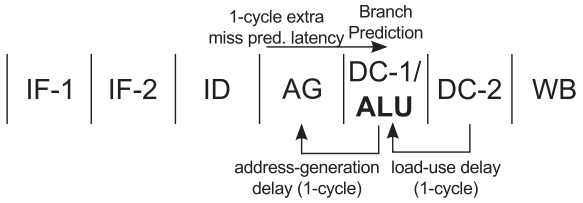
Fig. 2b represents a pipeline in which the ALU is located in the DC-1 stage (like in MIPS1004K [15]). We will call this pipeline configuration *intermediate LUI/AGI*. In this configuration, the load-use delay is reduced to one cycle, but a 1-cycle address generation dependency is introduced. Hence a stall cycle will be caused if a dependent ALU operation comes directly after a load operation, or an AG operation depends on the result of the previous ALU operation. In addition, the branch penalty is increased by one cycle compared to the reference LUI pipeline configuration.

Fig. 2c represents an AGI-type pipeline in which the ALU is located in the DC-2 stage. The ARM Cortex-A5 processor uses this type of configuration [16]. While the load-use delay is completely eliminated, this configuration leads to an increase of the address generation dependency to two cycles. Hence, if an AG operation is dependent on the result of one of the previous two ALU operations, stall cycle(s) will be caused. The branch penalty is increased by two cycles compared to the reference LUI pipeline configuration.

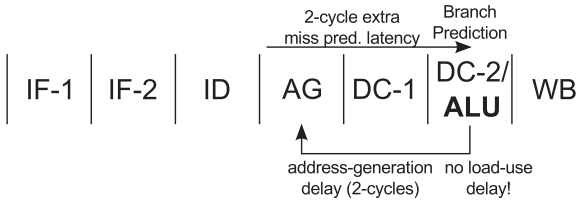
Finally, Fig. 2d represents a pipeline which utilizes two identical ALUs in different stages, namely early and late ALUs. The early ALU is located in the EXE stage and the late ALU is located in the DC-2 stage. This configuration was introduced in the ARC HS processors [5] and we will refer to this as *dual-ALU*. In this configuration, the load-use delay is completely eliminated, just as in the previously described configuration. In addition, most of the address generation dependencies are also eliminated by using the early ALU as soon as the dependencies allow so. As soon as there is no load-use dependency, the early ALU is used. Whenever an ALU



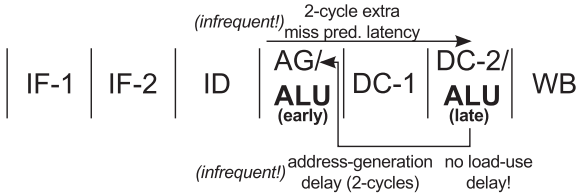
(a) Reference configuration with ALU in EXE stage (LUI).



(b) Configuration with ALU in DC-1 stage (intermediate LUI/AGI).



(c) Configuration with ALU in DC-2 stage (AGI).



(d) Configuration with ALUs in both EXE and DC-2 stages (dual-ALU).

Fig. 2. Evaluated pipeline configurations.

operation depends on one of the previous two load operations, it is diverted to the late ALU. If there are other ALU operations immediately depending on the result of diverted ALU operations, these are also diverted to the late ALU. If, during this process, an AG operation depends on one of the previously diverted two ALU operations, stall cycle(s) are caused. Similarly, the 2-cycle extra branch prediction penalty is only caused if the branch resolution depends on one of the last two diverted ALU operations. But these stall cycles or extra branch penalties happen very infrequently in this configuration (see Sec. V). It should be noted that the two ALUs never perform any redundant operations: Dependencies can be detected dynamically so that an ALU operation executes on either the early or the late ALU. The ALUs can execute different ALU operations in the same cycle.

Fig. 3 explains the working mechanism of the dual-ALU pipeline configuration. This code snippet also includes examples to show the source of stall cycles for different types of pipelines. For example, the address generation operation for *op2* depends on *op1* and, hence, *op2* will cause a 1-cycle stall for the intermediate LUI/AGI pipeline, and it will cause a 2-cycle stall for the AGI pipeline. Another example can be given for load-use dependency: *op3* depends on *op2* and, hence, *op3*

will cause a 2-cycle stall for the LUI pipeline, and it will cause a 1-cycle stall for the intermediate LUI/AGI pipeline. The comments in the code snippet of Fig. 3 identify which ALU operation will be handled in which stage for the dual-ALU pipeline configuration. Since *op3* depends on *op2*, it is diverted to the late ALU and the upcoming ALU operations are also diverted to the late ALU due to the chain dependencies. Hence, the branch operation of *op5* will have an extra 2-cycle penalty if the branch is mispredicted. Even though *op7* is diverted to the late ALU, *op9* can be handled in the early ALU since no dependency exists between *op9* and the instructions immediately preceding. It should be noted that since there is exactly one instruction between *op9* and *op7*, both the early and late ALUs will be used in the same cycle. The branch of *op10* can be resolved without any extra misprediction penalty since it is being resolved in the early ALU.

```
op1:  add $3,$2,$1  (use early ALU)
op2:  ld $5,$3(100)
op3:  sub $7,$5,$10 (use late ALU)
op4:  sra $12,$7,$3 (use late ALU)
op5:  bne L3, $12,$8 (resolve branch in late ALU)
    L2:
op6:  ld $3,$29(100)
op7:  add $8,$3,$5 (use late ALU)
op8:  sw $8,$29(200)
op9:  sra $10,$10,1(use early ALU)
op10: beq L2,$10,$0 (resolve branch in early ALU)
```

Fig. 3. MIPS assembly code snippet for dual-ALU approach.

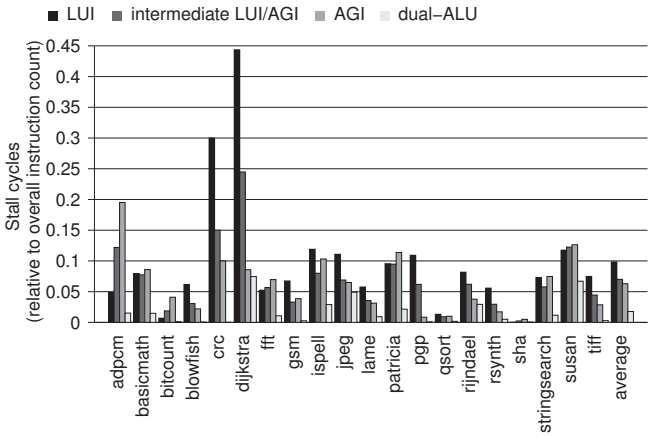
It should be noted that moving the ALU to later stages in the pipeline does not increase the required forwarding for the ALU, since it requires less and less forwarding from the stages ahead. But it requires an extra 68 flip-flops for each stage it is moved. These flip-flops dissipate 1 pJ of energy for each active cycle, which is 1.5% of the overall pipeline energy per cycle, not considering multiplication and cache access. The overhead is 2 pJ when the ALU is moved two stages. In the dual-ALU approach, the 2 pJ energy overhead only manifests when an ALU operation is diverted to the late ALU.

V. Exploration Results

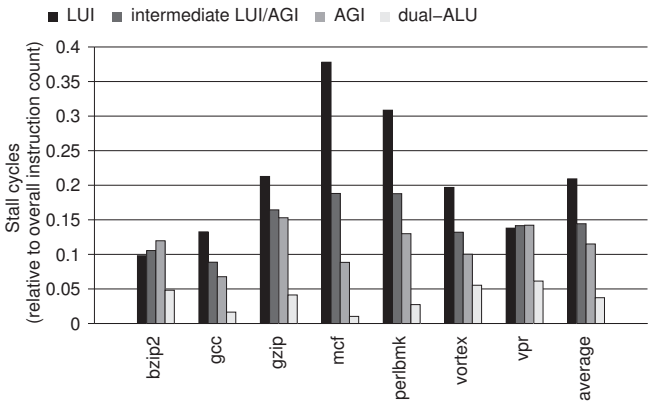
In this section, we present our exploration results in terms of stall cycles, execution time and energy dissipation for the four pipeline configurations previously introduced.

Fig. 4 shows stall cycles statistics. Since the stall cycles are normalized to the overall instruction count, the figure shows the execution time overhead if all the instructions were to execute in one cycle. Since this is not the case in processors due to memory operations etc., the execution time statistics are presented separately. The stall cycle trend is the same for the MiBench benchmarks presented in Fig. 4a and the SPEC2000int benchmarks presented in Fig. 4b. As the ALU is moved to later stages in the pipeline, the stall cycles are decreasing. The dual-ALU configuration reduces the stall cycles significantly compared to the single ALU approaches.

The LUI-type pipeline with its ALU in the execute stage has 10% (MiBench) and 21% (SPEC2000int) stall cycles due to



(a) Stall cycles using (MiBench).

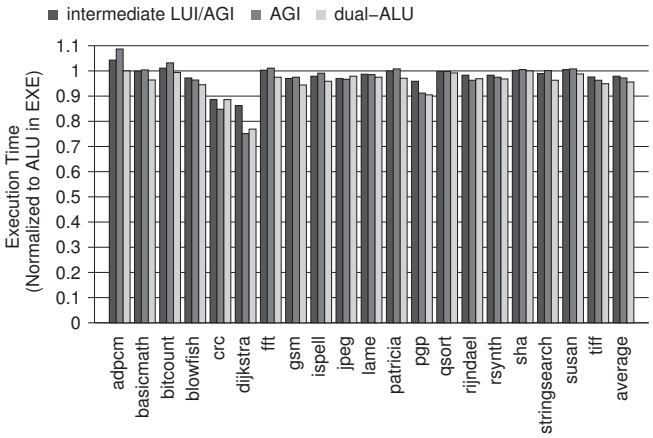


(b) Stall cycles (SPEC2000int).

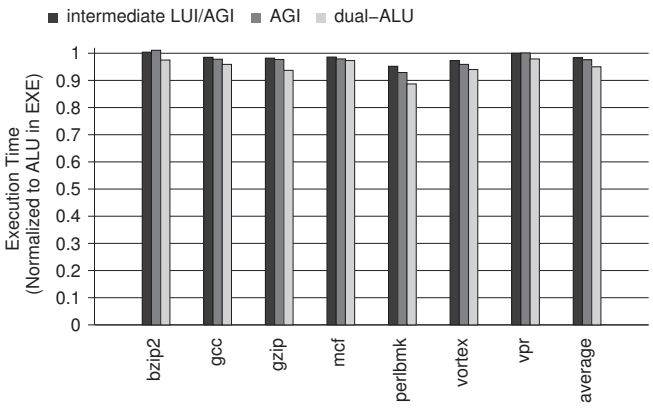
Fig. 4. Stall cycles for different pipeline configurations.

data dependencies. The intermediate LUI/AGI pipeline reduces the stall cycles to 7.0% and 14.4%, while they are reduced to 6.2% and 11.5% for the AGI pipeline. Stall cycles are only 1.7% and 3.7% for the dual-ALU pipeline. Since the SPEC2000int benchmarks have more load-use dependencies than MiBench, the proportion of stall cycles is much higher here for the LUI pipeline. Also, it is clear that the location of the ALU has a more pronounced impact on the SPEC200int results. The dual-ALU configuration reduces the stall cycles by approximately 83% for both of the benchmark suites. In some cases, moving the ALU to later stages can cause more stall cycles due to an increasing branch penalty. This is especially visible for the *adpcm* benchmark in which load/store operations are very few compared to the overall instruction count. For this benchmark, it is clear that an increasing branch penalty nullifies the benefits of moving the ALU to later stages, but this is not the common case. Moreover, the dual-ALU configuration, which never increases the stall cycles, performs well for this benchmark.

Fig. 5 presents execution times normalized to the LUI pipeline (Fig. 2a). On average, the execution time is improved by 2.1% (MiBench) and 1.7% (SPEC2000int) for the intermediate LUI/AGI pipeline. The improvement is 2.8% and 2.4% for the AGI pipeline, while for the dual-ALU pipeline, the improvement is 4.4% for MiBench and 5.0% for SPEC2000int.

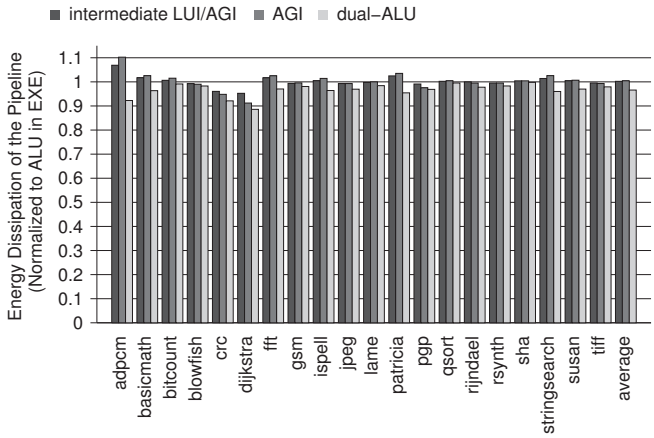


(a) Execution time (MiBench).

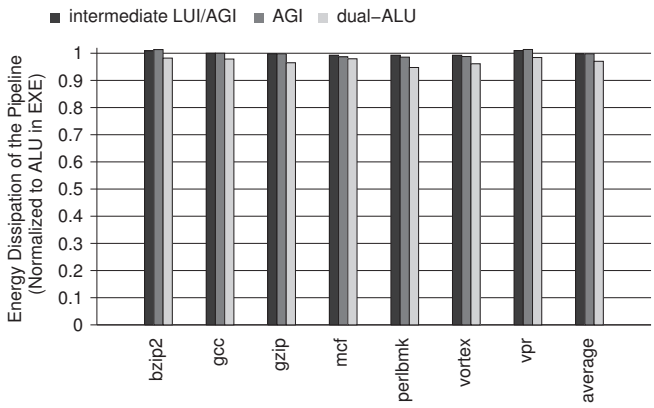


(b) Execution time (SPEC2000int).

Fig. 5. Execution time for different configurations (normalized to LUI).



(a) Pipeline energy dissipation (MiBench).



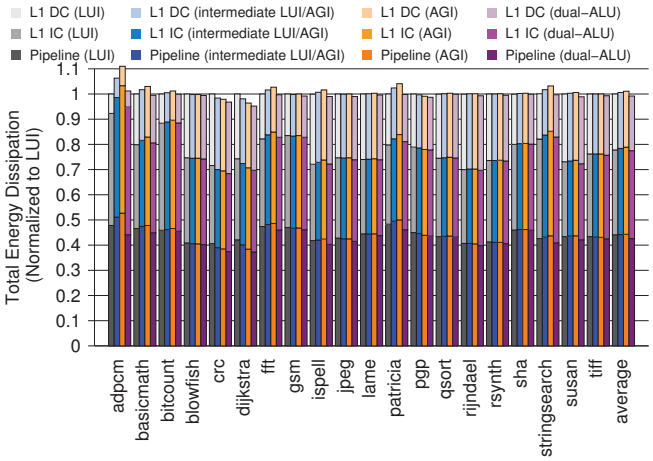
(b) Pipeline energy dissipation (SPEC2000int).

Fig. 6. Pipeline energy for different configurations (normalized to LUI).

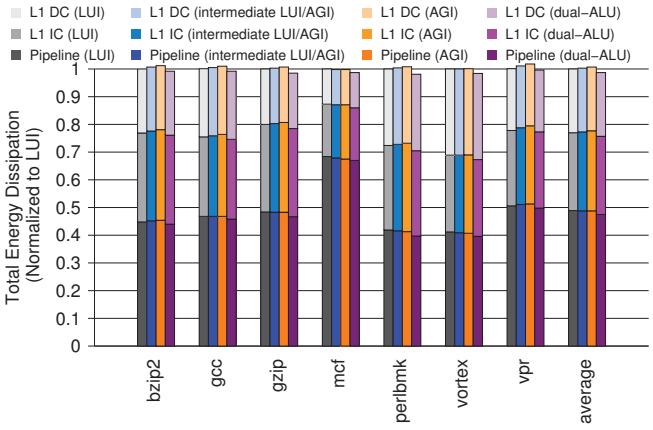
Even though relatively more stall cycles are eliminated in the SPEC2000int benchmarks, the execution time improvement is not that different from MiBench. This is due to the fact that SPEC2000int benchmarks have much higher cache miss rates and, hence, the stall cycles that do occur due to pipeline hazards have less impact on execution time. The execution time improvement can be as high as 25% for the *dijkstra* benchmark when the dual-ALU configuration is used. The single-ALU pipelines occasionally increase execution time due to increasing stall cycles, but this is not that common. In contrast, the dual-ALU pipeline always improves the execution time as expected from the stall cycle results.

Fig. 6 shows the pipeline energy for different configurations; the energy is normalized to the LUI pipeline. This energy metric includes all the pipeline components and the clock network apart from the level-1 caches. As shown, on average, the pipeline energy increase is negligible for the intermediate LUI/AGI and the AGI pipeline for both MiBench and SPEC2000int. Even though the execution time is improved for both configurations, the energy dissipation is slightly increased due to an increasing branch misprediction latency. When the ALU is moved to a later stage in the pipeline there are more instructions that are fetched and processed in the first stages of the pipeline during a branch misprediction. Fetching and processing an extra instruction is much more costly in terms of energy than a stall cycle and, hence, the energy saved on stall cycles is canceled out by the increased energy dissipation due to branch mispredictions. On average, the energy dissipation of the pipeline for dual-ALU configuration is reduced by 3.5% and 3% for MiBench and SPEC2000int, respectively. For the dual-ALU configuration, the energy dissipation is reduced for all benchmarks. The reduction in the pipeline energy dissipation can be as high as 12% as for the *dijkstra*. The energy savings are a result of 1) a shorter execution time which reduces clock network energy and 2) a significantly reduced branch misprediction overhead.

Fig. 7 shows the total energy dissipation normalized to the LUI pipeline in Fig. 2a. Since the energy is given for three CPU components, i.e., L1 DC, L1 IC, and pipeline, we also present the LUI pipeline energy. On average, the total energy dissipation is increased by 0.6% (MiBench) and 0.3% (SPEC2000int) for the intermediate LUI/AGI pipeline. The increase in total energy is 1.1% and 0.7% for the AGI pipeline. The reason for the increasing energy dissipation is that the energy overhead of L1-IC accesses during branch mispredictions are included. Turning our attention to the dual-ALU configuration, the total energy is *reduced* by 0.8% (MiBench) and 1.3% (SPEC2000int). The relative energy improvement will be higher if the L1 DC is optimized for energy, since the absolute energy dissipation will decrease.



(a) Total energy dissipation (MiBench).



(b) Total energy dissipation (SPEC2000int).

Fig. 7. Total energy for different configurations (normalized to LUI).

VI. Conclusions

We have shown, in line with previous work [4], that moving the ALU to later stages in the pipeline can improve the execution time. We have also shown that moving the ALU causes, due to increased branch penalties, only a negligible increase in energy dissipation. In this work, the extra energy dissipated in level-2 caches is not included and, hence, this small energy overhead might disappear when the energy savings in higher-level caches during stall cycles is considered.

So long as they do not cause a dramatic increase in power or energy dissipation, execution time improvements are always welcome. We have shown that the dual-ALU pipeline configuration that was commercialized recently [5] improves the execution time considerably and reduces the energy dissipation at the same time. The area overhead of the extra ALU is 12% of the pipeline area without level-1 caches and 3.7% of the overall core area which includes pipeline and level-1 caches. Thus, the dual-ALU approach is a very viable option for single-issue in-order pipelines. Nugent proposed a dual-ALU pipeline configuration in which ALUs are located in the EXE and DC-1 stages [17]. This configuration can potentially provide slightly higher energy savings compared to the dual-ALU approach evaluated in this paper due to the fact that branch mispredictions, which can not be resolved in the early ALU, are resolved one cycle earlier. However, the configuration will provide less performance improvements since it has a 1-cycle load-use delay for the late ALU.

One other aspect of moving an ALU to later stages is that it requires a different scheduling to be optimally used. For example, if one processor generation initially has the ALU in the execute stage, but that this ALU in a more advanced generation is moved to a later stage then the scheduling should change: Instead of moving ALU operations away from load operations, AG operations have to be moved away from ALU operations that have an immediate dependency. This scheduling will not affect the correctness, but the efficiency of the ALU approach. Since the dual-ALU approach eliminates most of the dependencies, it is the pipeline configuration evaluated in the paper that is the least affected by the scheduling problem. This is another good motivation for using two ALUs.

References

[1] P. Greenhalgh, *big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7*, White Paper, ARM, Sep. 2011.

[2] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "Tile64™ processor: A 64-core SoC with mesh interconnect," in *IEEE Int. Solid-State Circuits Conf.*, Feb. 2008, pp. 88–89.

[3] D. Boggs, G. Brown, N. Tuck, and K. Venkatraman, "Denver: Nvidia's first 64-bit ARM processor," *IEEE Micro*, vol. 35, no. 2, pp. 46–55, Mar. 2015.

[4] M. Golden and T. Mudge, "A comparison of two pipeline organizations," in *27th Annual Int. Symp. on Microarchitecture*, 1994, pp. 153–161.

[5] M. Thompson, *Optimizing high-end embedded designs with the ARC HS processor family*, Technical Bulletin, Synopsys. [Online]. Available: http://www.synopsys.com/Company/Publications/DWTB/Pages/dwtb-arc-hs-2014Q1.aspx

[6] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002.

[7] *Design Compiler®, v. 2010.03*, Synopsys, Inc., Mar. 2010.

[8] *Encounter® Digital Implementation (EDI), v. 10.1.2*, Cadence Design Systems, Inc., Jul. 2011.

[9] Embedded Microprocessor Benchmark Consortium. [Online]. Available: http://www.eembc.org

[10] *PrimeTime® PX, v. 2011.06*, Synopsys, Inc., Jun. 2011.

[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Int. Workshop on Workload Characterization*, Dec. 2001, pp. 3–14.

[12] SPEC CPU2000 V1.3, Standard Performance Evaluation Corporation. [Online]. Available: https://www.spec.org/cpu2000/

[13] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design, The Hardware/Software Interface*, 2nd ed. Morgan Kaufman Publishers Inc., 1998.

[14] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy, "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *34th ACM/IEEE Int. Symp. on Microarchitecture*, Dec. 2001, pp. 54–65.

[15] *MIPS® 1004K™ Coherent Processing System Datasheet*, MIPS Technologies, Jul. 2009.

[16] T. R. Halfhill, "ARM's midsize multiprocessor," *Microprocessor*, Oct. 2009.

[17] P. R. Nugent, "Repeated ALU in pipelined processor design," US Patent US5 333 284 A, 1994.