Power-Performance Trade-off using Pipeline Delays

G. Surendra, Subhasis Banerjee, S. K. Nandy
CAD Lab, SERC, Indian Institute of Science
Bangalore INDIA 560012
e-mail: {surendra@rishi., subhasis@rishi., nandy@}serc.iisc.ernet.in

Abstract— We study the delays faced by instructions in the pipeline of a superscalar processor and its impact on power and performance. Instructions that are ready-on-dispatch (ROD) are normally delayed in the issue stage due to resource constraints even though their data dependencies are satisfied. These delays are reduced by issuing ROD instructions earlier than normal and executing them on slow functional units to obtain power benefits. This scheme achieves around 6% to 8% power reduction with average performance degradation of about 2%. Alternatively, instead of reducing the delays faced by instructions in the pipeline, increasing them by deliberately stalling certain instructions at appropriate times minimizes the duration for which the processor is underutilized leading to 2.5-4% power savings with less than 0.3% performance degradation.

I. Introduction

The units of a superscalar microprocessor often do not operate at their full potential due to dynamic variations in data dependencies and resource constraints. The performance benefits obtained during such phases of underutilization is negligible and (such phases) can be reduced to some extent by stalling/throttling [1][2] or shutting off (clock gating) certain stages of the processor pipeline. In this paper we investigate the underutilization in the fetch stage of the pipeline since it critically affects both power and performance. We show that a complexity effective "early fetch throttling" heuristic applied during fetch underutilization results in 2.5% to 4% power savings with less than 0.3% performance degradation. Most superscalar processors also employ the oldest first instruction selection policy in their issue logic. In such processors, younger ready instructions suffer from delayed selection. In particular, we show that instructions that are ready-on-dispatch (ROD) (i.e. whose dependencies are satisfied at the dispatch stage) are the ones that suffer from delayed issue with about 57% of them spending more than 1 cycle in the instruction queue (IQ). In this paper we modify the selection policy so that ROD instructions are selected earlier than normal but are issued to slow low power functional units (FUs) [3] in an attempt to obtain energy savings.

We use Wattch [4] (with *cc3* style of clock gating), a performance and power analysis simulation tool that is built on top of SimpleScalar [5] which is a well known architectural simulator to carry out our experiments. The base processor configuration used is a 4-way superscalar processor with in-order commit, dynamic branch prediction with 7 cycles misprediction penalty, 64(and 32) entry IQ (and load store queue), 64K 2-way set associative L1 instruction and data caches with 1 cycle hit latency and 1MB L2 cache with 12 cycle latency. We evaluate our ideas on a set of benchmarks from MiBench [6], compile the benchmarks with $-O3$ optimization and use the input data sets provided along with the program distribution. Statistics collection begins only after the initialization phase elapses.

The rest of the paper is organized as follows. In section II we analyze processor underutilization and propose the fetch throttling scheme for reducing underutilization. In section III, we analyze the delays faced by instructions in the IQ based on their readiness at dispatch and evaluate the early issue policy mentioned previously. We summarize the main results and conclude the paper in section IV.

II. Fetch Stage Underutilization and throttling

Wall [7] showed that the amount of ILP (Instruction Level Parallelism) within a single application varies by up to a factor of three. In other words, at any instant in time, it is possible for different pipeline stages to be operating in various degrees of (under/full) utilization. The fetch stage is said to be underutilized if the number of instructions that can be fetched per cycle from the instruction cache is less than the maximum fetch rate. Our simulations indicate that the average fetch rate is 3.5 instructions/cycle with the full fetch rate maintained for only 50% of the time. The fetch stage fluctuates between normal and underutilized states depending on program behavior and available resources. In this paper we assume that the fetch stage is underutilized if 2 or fewer instructions can be fetched per cycle which corresponds to about 30% of all instruction fetches (not including zero instruction fetches).

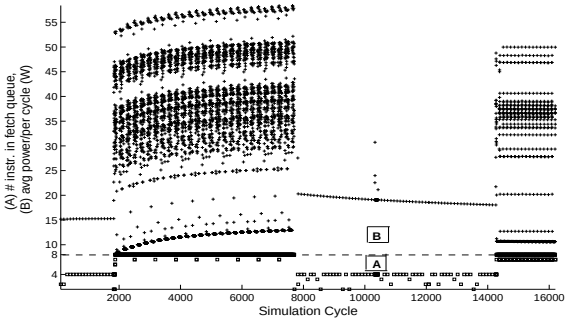

Fig. 1. Variation in (A) number of instructions in IFQ (below dashed line) and (B) avg power as a function of simulation time for *cjpeg*

The instruction fetch rate will be lower than the maximum fetch rate when the number of free entries in the Instruction

Fetch Queue (IFQ) is less than the fetch width and becomes zero when the IFQ is full. In the proactive early fetch throttling scheme, we stall the fetch stage when the number of entries in the IFQ crosses a threshold (a threshold of 70% with 2 cycles stall gave the best Energy-Delay Product (EDP) gains). On a fetch stall, instruction cache, branch predictor and memory bus accesses are disabled (gated) resulting in lower energy consumption. Fig 1 shows the variation of average power as a function of simulation time (portion B - above dashed line). For comparison purposes we also show the variation in the IFQ size (the maximum fetch queue size is 8) as simulation proceeds (portion A - below dashed line) It is seen that power dissipation is significantly lower when the number of instructions in the IFQ is 4 rather than 8 indicating that reducing the density of instructions in the IFQ improves power. Fetch throttling does this by ensuring that the IFQ is cleared to some extent before new instructions are fetched

Fetch throttling when applied during phases of underutilization decreases the EDP by a maximum of 2.5% in most benchmark programs. The single most important component resulting in power savings due to early fetch throttling is the icache power which reduces by 6.5%. Usually, every icache access results in the entire cache line being fetched. However, some of the fetched instructions are wasted in the event that the IFQ has fewer empty slots than the cache line size. Another access to the same cache line has to be initiated in the near future to fetch the instructions that were not utilized. Fetch throttling utilizes the entire fetch bandwidth by ensuring that most instructions in the cache line are used thereby saving power. In other words, though the number of instructions fetched from the icache is fixed (for a program), the number of cache line accesses to fetch those instructions is reduced.

It is observed that only one or two instructions are fetched per cycle (on average) for about 15% and 20% of the time. On applying fetch throttling during reduced fetch bandwidth, the above values reduce to around 3% and 4% respectively. Since fetch stalls help in clearing the IFQ, the time spend by instructions in the IFQ decreases by 14% and average power reduces. Also, in subsequent cycles, the maximum fetch bandwidth is utilized in fetching instructions thereby ensuring that performance degradation is minimized. Overflows in the Register Update Unit (RUU) and Load/Store queue (LSQ) are also minimized by 15% and 5% due to fetch throttling. Since the occupancy of various queues is reduced due to stalls, the wakeup logic power is decreased since it involves a broadcast of tags (or results) to all entries in the RUU.

III. Instruction Delays in the Issue Queue

Instructions that enter the pipeline face varying amounts of delay due to structural and data hazards. These delays increase as more stalls are introduced due to techniques such as throttling. However, performance degradation depends on other factors such as criticality and utility of instructions being delayed. For example, delaying the execution of an instruction fetched from a mispredicted path will not affect performance but will save energy since it is likely to be squashed at a later point in time.

The two basic functions involved in the instruction issue logic are instruction wakeup and selection. The selection

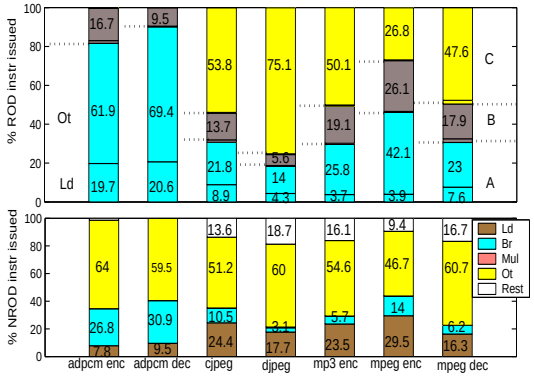

Fig. 2. Top: ROD instructions that spend (A) 1 cycle, (B) > 1, < 3 cycles, (C) > 3 cycles in IQ. Bottom: NROD instructions issued in 1 cycle. For example, in *cjpeg*, 53.8% of ROD instructions spend >3 cycles in the IQ; 13.6% of NROD instructions spend more than 1 cycle in the IQ *after* they become ready to issue; 24.4% of NROD instructions (which happen to be loads) are issued immediately after they become ready.

logic is responsible for selecting ready instructions from the IQ based on either oldest first, random or highest latency first policy [8][9] and issuing them to available FUs for execution. The larger the IQ, the larger is the pool of instructions available for selection and higher is the performance and power. Ready instructions spend more than one cycle in the IQ if they are not immediately selected for issue by the selection logic. There are two possible reasons for this. Firstly, if the FU than can execute the instruction is busy and secondly, if the instruction is not selected because there are other older ready instructions that are selected earlier and the issue ports are exhausted.

Fig 2 shows the time (in cycles) spent by ROD and NROD (Not-Ready-On-Dispatch) instructions in the IQ *after* their dependencies are resolved. Each bar in the top figure (ROD instructions) is further subdivided into three distinct portions (having different shades) denoted by A, B, and C. Portion C represents ROD instructions that spend more than 3 cycles in the IQ. Portion B depicts ROD instructions that spend more than 1 cycle and less than 3 cycles in the IQ while portion A represents ROD instructions that are immediately issued (in the next cycle) after dispatch. We see that nearly 60% of all ROD instructions spend more than 1 cycle in the IQ while 40% spend more than 3 cycles waiting to be issued. Further, our simulations indicate that most ROD instructions are non-critical [10] and can therefore be executed on slow low power functional units. Also, most ROD instructions that spend greater than 3 cycles in the IQ belong to the integer ALU (denoted by *Ot* in Fig. 2) class which normally complete execution in 1 cycle. Fig 2 (bottom) shows the number of NROD instructions issued immediately (in the next cycle) after they become ready. A significantly larger fraction of the older NROD instructions are critical (compared to ROD instructions) and are not delayed in the IQ once their dependencies are satisfied. This is due to the oldest first selection policy which gives priority to these older instructions. A further breakup indicating the types of instructions issued in 1 cycle is also shown in the figure. The portion indicated by the white shade (denoted by ”Rest” in the legend) represent NROD instructions that spend more than 1 cycle (non critical NROD

instructions) in the IQ.

To reduce the waiting time of ROD instructions in the IQ, we use an early selection policy so that these instructions are issued earlier than normal. To obtain energy gains these ROD instructions are issued to slow low power FUs. Early issue with slow execution ensures that performance degradation does not exceed acceptable levels. Every cycle, the selection logic picks a certain number of ready instructions to be issued based on resource availability using the oldest first selection policy. A further check is carried out to determine if a ROD instruction is selected for issue in the current cycle. If an ROD instruction is selected, it is issued to a slow FU if one is available. If no ROD instruction is selected for issue in the current cycle, the IQ is searched for a ROD instruction. If one is found, the search is terminated and the ROD instruction is given a higher priority (over other selected NROD instructions) and issued (provided the slow FUs are available) in the current cycle. Since searching the IQ takes up additional energy (this is about 2% overhead) and is known to be in the critical path [9], we limit our search to a window of a maximum of 5 entries. Further, we begin our search at some arbitrary point (preferably near the tail of the queue) since ROD instructions are more likely to be found near the tail. If no ROD instruction is found within the specified window, the issue logic proceeds in the normal way. We assume that only one ROD instruction can be issued per cycle (we use one slow FU in our experiments since the number of NROD instructions exceeds ROD instructions by roughly a factor of 3). With this scheme, the IQ continues to contain entries in the oldest first order, but the selection mechanism is altered to give priority to a ROD instruction. The effect of fetch throttling and early issue of ROD instructions with delayed execution on performance and EDP is shown in fig 3. We assume that the slow FUs operate at 2.2V, 300MHz while the normal FUs operate at 2.5V, 600MHz.

It is observed that introducing stalls in the fetch stage (bar 1) during phases on underutilization has negligible impact on the execution time, reduces power by 2.5% to 4%, and improves the EDP for all benchmarks (except *mpeg dec*). Early issue of ROD instructions with slow execution (bar 2) yields better power savings (around 6% to 7.5%) but also degrades performance by an average of 2% (max 7.5% for *mpeg dec*). Combining fetch throttling with early issue (bar 3) results in the best power and EDP gains among all schemes for most benchmarks. Issuing ROD instructions earlier and executing them on normal FUs (bar 4) is also an equally competitive scheme and reduces EDP to some extent. This indicates that early issue of ROD instructions (regardless of whether they are executed on slow FUs or normal FUs) is beneficial and confirms the fact that ROD instructions are normally latency tolerant.

IV. Conclusions

In this paper we examined the effect of introducing pipeline stalls during phases of processor underutilization on power and performance. Power profiles of programs indicate that power dissipation is dependent on the instruction density in pipeline stages. Stalls help in clearing up various queues within the pipeline reducing the instruction occupancy time and power with some performance degradation. The impact of introducing stalls on power depends on the branch prediction accuracy and the available parallelism. We have also shown that instructions that are normally ready-on-decode (ROD) suffer from delayed selection in an oldest first issue policy and that close to 40% ROD instructions wait for more than 3 cycles in the IQ. We evaluate an early issue policy to reduce the waiting time of ROD instructions in the IQ and show that this early issue achieves around 6% to 8% power reduction with about 2% degradation in performance (average value) when a single fixed slow low power FU is used. Finally, we combine throttling with early issue to exploit the mutually beneficial tendencies seen them and achieve higher power savings.

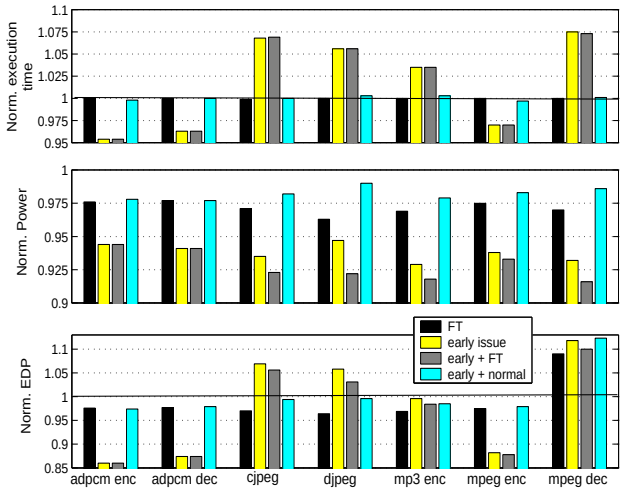

Fig. 3. Statistics with fetch throttling and early issue of ROD instructions

References

[1] J. L. Aragon, J. Gonzalez, and A. Gonzalez, "Power-Aware Control Speculation through Selective Throttling," *In Proc. HPCA*, 2003.

[2] A. Baniasadi, and A. Moshovos, "Instruction Flow-Based Front-end Throttling for Power-Aware High-Performance Processors," *In Proc. ISLPED*, 2001.

[3] J. S. Seng, E. S. Tune, D. M. Tullsen, "Reducing Power with Dynamic Critical Path Information", *In Proc. Micro-34*, Dec 2001.

[4] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for Architectural-Level Power Analysis and Optimizations", *In Proc. ISCA*, June 2000.

[5] D. Burger, T.M Austin, and S. Bennet, "Evaluating Future Microprocessors: The SimpleScalar Tool Set," Technical Report CS-TR-96-1308, University of Wisconsin-Madison, July 1996.

[6] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," *IEEE 4th Workshop on Workload Characterization*, Dec 2001.

[7] D. W. Wall, "Limits of Instruction-Level Parallelism," *In Proc. ASPLOS*, Nov 1991.

[8] A. Buyuktosunoglu, A. El-Moursy, D. Albonesi, "An Oldest-First Selection Logic Implementation for Non-Compacting Issue Queues", *15th Intn'l ASIC/SOC Conf.*, 2002.

[9] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-Effective Superscalar Processors," *In Proc. ISCA* 1997.

[10] B. Fields, S. Rubin, R. Bodik,"Focusing Processor Policies via Critical-Path Prediction," *In Proc. ISCA* 2001.