The 47th IEEE International Midwest
Symposium on Circuits and Systems

Exploiting the Behavior of Ready Instructions for Power Benefits in a Dynamically Scheduled Embedded Processor

G. Surendra, Subhasis Banerjee, S. K. Nandy
CAD Laboratory, Supercomputer Education and Research Center
Indian Institute of Science, Bangalore 560012, India
Email: {surendra@cadl, subhasis@cadl, nandy@serc}.iisc.ernet.in

***Abstract*—Many instructions in a dynamically scheduled superscalar processor spend a significant time in the Instruction Window (IW) waiting to be selected even though their dependencies are satisfied. These "delays" are due to resource constraints and the oldest first selection policy used in many processors that gives a higher priority to older ready instructions than younger ones. In this paper, we study the "delay" and criticality characteristics of instructions based on their readiness during dispatch. We observe that most ready-on-dispatch (ROD) instructions are non critical and show that 57% of these instructions spend more than 1 cycle in the IW. We analyze the impact of (i) steering ROD instructions to slow low power functional units and (ii) early issue of ROD instructions, on power and performance. We find that the "early issue and slow execution" of ROD instructions reduces power consumption by 4-12% while degrading performance by about 5%. On the other hand, "early issue normal execution" of ROD instructions results in 3.5% power savings with less than 1% performance loss. Further, we find that the above policies reduce the energy expended in executing wrong path instructions by about 2%.**

I. Introduction

The design considerations for high performance embedded processors and System on a Chip (SoC) solutions is not only governed by real time performance requirements but also by power dissipation (since power translates directly to heat) and cost considerations. In case of embedded microprocessors, energy consumption is important since battery life is the main design concern. An interesting trend that is emerging is the increased dependence on existing programmable processors and platform architectures to run application specific code [1]. This reduces the design time (time-to-market), but compromises on factors such as power which is typically larger in more general programmable processors. Also, due to the ever changing standards/protocols, emerging applications such as media and packet processing require programmable architectures comprising of high performance embedded microprocessors that exploit parallelism at instruction level (e.g. out-of-order superscalar, VLIWs and vector processors), thread level (e.g. multithreaded processors) and task level (homogeneous/heterogeneous chip multiprocessors). The use of such programmable processors for domain specific embedded applications opens up new possibilities for optimizing power and performance. For instance (as will be discussed later in this paper), depending on the nature of instructions, the selection policy may be tuned/customized for a set of applications.

Motivation: Instructions that are fetched are renamed to remove false dependencies and are allocated entries in the Instruction Window (IW). The number of resources (Functional Units (FU), memory ports etc) and the sizes of various queues (fetch queue, IW etc), in a typical processor is fixed based on cost, performance, power and Instruction Level Parallelism (ILP). The motivation for this work arises from the fact that processors with "limited" resources, while being power efficient, can result in instructions spending significant time in the IW waiting to be issued even *after* their dependencies are resolved. In this paper, we quantify this "delay" for different media and networking applications and examine priority schemes for reducing this delay. For the purpose of this study, we divide instructions into two classes (i) Ready On Dispatch/Decode (ROD) - instructions whose data dependencies are satisfied when they are dispatched and (ii) Not Ready On Dispatch/Decode (NROD) - instructions whose data dependencies are not satisfied when they are dispatched. Classifying instructions into ROD/NROD groups is easy to implement and is particularly suitable for *power-aware* [2] embedded systems in which cost and complexity considerations tend to influence design. We show that most ROD instructions are non critical and that 40% of ROD instructions spend more than 3 cycles in the IW. To reduce power dissipation, we evaluate an early issue policy in which ROD instructions are given higher priority but are issued to slow low power FUs. In general, we observe that delaying the execution of ROD instructions is beneficial in obtaining energy savings (with minimal performance degradation) compared to delaying other types of instructions.

II. Simulation Environment

We use Wattch [3], a performance and power analysis simulation tool that is built on top of SimpleScalar [4] and make necessary changes to the Out-Of-Order (OOO) simulator to carry out our study. Throughout this paper, we use the *cc3* style of clock gating provided by Wattch, which scales power linearly with unit usage. The base processor configuration is a 4-way superscalar processor with a combined branch predictor having 7 cycles misprediction penalty, 128(32) entry RUU(LSQ) (see [4]), 4 integer FUs, on-chip L1 instruction/data caches (8K, 2way, 32 byte block size) with 1 cycle hit latency and a L2 cache (256K, 4 way, 32 byte block size) of 12 cycle latency. These parameters roughly match those of a modern processor used in desktop as well as high performance embedded applications (it has been shown in [5] that OOO issue does not lead to unpredictable execution time and is suitable for real time applications) such as in set-top boxes, routers etc. Superscalar processors have been used in embedded media applications (e.g. NEC's V830R/AV [6]) as well as in network processors (e.g. Ezchip's TOPcore architecture - www.ezchip.com) since they achieve high performance by hiding latency effectively [7]. Architectures such as simultaneous multithreaded processors which are derived from superscalar processors are also common in the network processing domain. We chose consumer and telecom related benchmarks from MiBench [8] (*adpcm enc/dec, jpeg enc/dec, mpeg2 enc/dec, mp3 encod-*

0-7803-8346-X/04/$20.00

ing) and packet processing benchmarks from CommBench [9] (*frag, rtr, reed enc/dec, drr, dh*) for this study. We compile the benchmarks with $-O3$ optimization and collected statistics until at least 200M instructions were committed after initialization phases were skipped. Unless otherwise mentioned, all simulation results are normalized with respect to the above base processor configuration.

When an instruction is issued for execution, its tag is broadcast to all entries in the IW to wakeup dependent instructions. It has been observed that many of these broadcasts are redundant and consume significant energy [10]. Folegnani *et. al.* [10] propose a gating scheme and eliminate tag broadcasts to instructions that have one or more operands ready. In this paper we implement a limited version of the scheme proposed in [10] and eliminate unnecessary tag broadcasts to ROD instructions (since both their operands are ready). This optimization does not degrade performance and results in 1.5% power savings. All the results discussed in this paper are with this optimization in place.

III. ROD/NROD Instructions

The classification of instructions based on their readiness when dispatched, rather than their opcode has several advantages - (i) This classification is dependent on the microarchitecture and is more representative of the dynamic state of the processor (e.g. availability of resources, cache misses etc., can alter the number of ROD/NROD instructions). In other words, microarchitectural features such as instruction selection policy, branch prediction accuracy, IW size, number of FUs, issue width etc are easily "tracked" by this classification. (ii) It ensures that the instruction mix is dependent on the application and the compiler optimization. For example, a higher compiler optimization could do a better job in ensuring that dependent instructions are placed as far apart as possible in the code schedule. This results in a consuming (dependent) instruction becoming ROD as the producing instruction would have completed execution. (iii) Identifying an instruction as being ready or not is a task that is routinely carried out by the processor and is deterministic in nature. This implies that no extra hardware is necessary to implement the above classification. (iv) It helps us analyze instruction criticality from the perspective of a whole class (i.e. ROD/NROD) of instructions enabling us to focus on one particular class for future optimizations. This is possible since the above classification closely reflects the microarchitecture and dynamic behavior of instructions.

Figure 1 shows the number of ROD and NROD dynamic instructions for different benchmarks. Each bar in the figure consists of 3 portions indicated by different shades. The top most portion denoted by "All" represents instructions with all operands ready on decode i.e. ROD instructions. The remaining portions represent instructions with some and none of the operands ready on decode i.e. NROD instructions. The figure also depicts the composition of performance limiting instructions such as loads (Ld), branches (Br), multiply/divide (Mul) and others (Ot). Others (Ot), represent instructions not belonging to any of the above classes and mostly consist of integer ALU instructions. It is observed that about 25% of all issued instructions belong to the ROD class while 75% are of the NROD type. Further, it is observed that most ROD instructions belong to the ALU class of instructions.

Ready instructions spend more than one cycle in the IW if they are not immediately selected for issue by the selection logic. There are two possible reasons for this. First, if the functional unit 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 bandwidth is exhausted. For instance, with the configuration mentioned in section 2, the processor is capable of issuing at its peak rate only 58% of the time while no instructions are issued for nearly 27% of the time. One to three instructions are issued per cycle for the remaining 15% of the time. The position of an instruction in the IW gives a rough estimate of whether it will be issued immediately or not since most instructions selected for issue are from the first few entries of the IW [10].

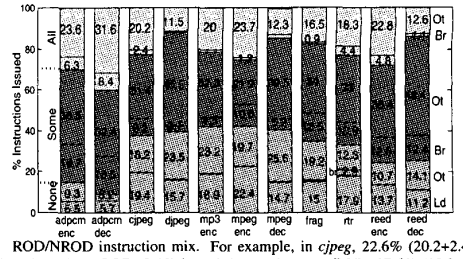

Fig. 1. ROD/NROD instruction mix. For example, in *cjpeg*, 22.6% (20.2+2.4) of all instructions issued are ROD; 2.4% branch instructions are ROD; 37.6% (18.2+19.4) of instructions issued are NROD with *none* of their operands ready when dispatched. Only contributions $\geq$ 1% are shown.

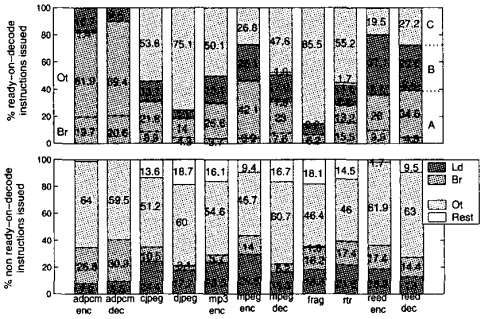

Fig. 2. Top: Fraction of ROD instructions that spend (A) 1 cycle, (B) 2 cycles, (C) >=3 cycles in IW. Bottom: Fraction of NROD instructions issued in 1 cycle. For example, in *cjpeg*, 53.8% of ROD instructions spend >=3 cycles in the IW; 13.6% of NROD instructions spend more than 1 cycle in the IW *after*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

Figure 2 (Top) shows the fraction of ROD instructions that spend 1 cycle, 2 cycles, and 3 or more cycles in the IW *after* their dependencies are satisfied. Each bar in the top figure (ROD instructions) is further subdivided into three distinct portions (having different shades) denoted by A, B, and C (see extreme right of the figure). Portion C represents ROD instructions that spend more than 3 cycles in the IW. Portion B depicts ROD instructions that spend 2 cycles in the IW while portion A represents ROD instructions that are immediately issued (in the next cycle) after dispatch. We find that for the base processor configuration nearly 60% of all ROD instructions spend more than 1 cycle in the IW while 40% spend more than 3 cycles waiting to be issued (these values are even larger for a 2-way superscalar processor). Fortunately, most ROD instructions that spend greater than 3 cycles in the IW belong to the integer ALU (Ot) class (followed by branches) which normally complete execution in 1 cycle. In other words, the execution latency of these instructions is smaller than the latency incurred due to waiting for FUs. The bottom portion of figure 2 shows the number of NROD instructions issued immediately (in the next cycle) *after* they become ready. Most NROD instructions

(this is nearly 100% for the *adpcm* benchmark) are not delayed in the IW 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. The portion indicated by the white shade (denoted by "Rest" in the legend) represents NROD instructions that spend more than 1 cycle in the IW *after* they become ready. We see that only about 11% of NROD instructions belong to this category.

Compiler optimizations such as pipeline scheduling and loop unrolling separate dependent instructions and introduce unrelated instructions, thus minimizing pipeline stalls due to true data dependencies. Figure 3 shows the waiting time of ROD/NROD instructions with *-O1* compiler optimization. A comparison between this and figure 2 (which uses *-O3* optimization) reveals that with higher optimization, the time spent by ROD instructions in the IW increases in all benchmarks (*adpcm enc/dec* is an exception). This implies that as compiler optimizations improve, pipeline stalls due to IW overflows become more pronounced especially in microarchitectures with small IW's. The fraction of NROD instructions that spend more than one cycle in the IW (denoted by "Rest" in the legend) increases for some benchmarks and decreases in others.

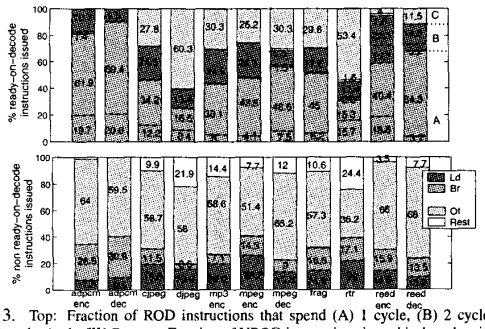

Fig. 3. Top: Fraction of ROD instructions that spend (A) 1 cycle, (B) 2 cycles (C) >=3 cycles in the IW. Bottom: Fraction of NROD instructions issued in 1 cycle with -O1 optimization.

Though several processor resources affect the time spent by instructions in the IW, we study only the impact of FUs in this paper. Figure 4 shows the waiting time of ROD/NROD instructions with two and six integer FUs. The remaining processor parameters are the same as that of the base configuration described in section 2. A comparison between figure 2 (which uses four FUs) and figure 4 indicates that the waiting time of both ROD and NROD instructions in the IW increase as the number of resources are reduced.

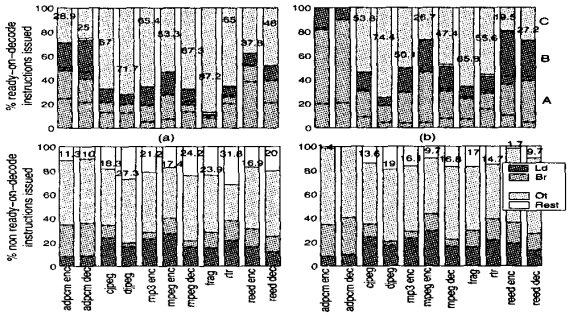

Fig. 4. Impact of FUs on ROD/NROD delays. (a) with 2 FUs (left) (b) with 6 FUs (right); Top: ROD instructions that spend (A) 1 cycle, (B) 2 cycles, (C) >=3 cycles in the IW. Bottom: NROD instructions issued in 1 cycle. Only a few values are shown for clarity.

We also evaluate the criticality of ROD and NROD instructions (see table 1) using the critical path predictor of Fields et. al [11] and observe that on average 5% and 21% of *all* instructions are *ROD and critical* and *NROD and critical* respectively. These results confirm the fact that criticality is predominantly determined by data dependencies.

TABLE I

Criticality of ROD and NROD instructions. Col 2,3 - fraction of ROD(NROD) instructions that are marked critical. Col 4,5 - fraction of *all* instructions dispatched that are ROD(NROD) critical. For example, in *cjpeg*, 31.51% of ROD and 35.18% of NROD instructions are critical. This implies that 9.3% and 24.78% of *all* dispatched instructions are *ROD and critical* and *NROD and critical* respectively.

Benchmark	% ROD critical	% NROD critical	% all ROD critical	% all NROD critical
adpcm enc	25.21	55.49	7.37	39.25
adpcm dec	26.24	25.71	11.14	14.8
cjpeg	31.51	35.18	9.3	24.78
djpeg	11.31	18.58	1.79	15.62
mp3 enc	14.57	27.15	4	19.53
mpeg enc	46.10	43.96	14.6	30
mpeg dec	13.22	14.06	2.34	11.56
frag	7.73	4.84	1.65	3.8
rtr	7.79	17.79	2	13
reed enc	18.37	18.37	6	18.65
reed dec	21.69	18.45	3.33	3.33

IV. Exploiting Delayed Execution

Now that we have analyzed the delay and criticality of ROD and NROD instructions, we examine how the above instruction classification is used for trading performance for power benefits. Previous work has focused on steering non-critical instructions to slow FUs [12] and scheduling critical instructions with higher priority in a clustered architecture[11]. This involves complicated mechanisms to detect critical path instructions. To avoid such complicated schemes, we sacrifice accuracy for simplicity and make use of the fact that most ROD instructions are non-critical and give them a higher priority to execute on slow low power FUs. These slow FUs consume low power by operating at a constant reduced voltage/frequency. Consequently, the operation and issue latencies of the slow FUs are larger (we assume these to be twice that of normal FUs in our experiments) than normal FUs. Further, we assume slow FUs of the integer ALU class since most ROD instructions are of this type i.e. only ALU instructions are executed slowly. Since the ratio of the number of ROD and NROD instructions is roughly 1 to 3, we use 3 normal FUs (that operate at 2.5V, 600MHz) and 1 slow FU (that operates at 1.75V, 300MHz). Though the above values are for a 0.35μ technology (this is the default provided by Wattch) which is quite dated, it serves to illustrate the effectiveness of our scheme. The voltage of 1.75V for the slow FU is obtained from the first order approximation of the relationship between circuit delay and supply voltage [13]. We do not use any dynamic scaling of voltage and frequency since in practice supply voltages cannot be varied on a continuous scale and each variation takes a certain finite time which can be as much as 150μ sec [14].

We use a modified instruction selection policy and examine its effect in the presence of slow and normal FUs. The combination of selection and steering policies leads to 4 cases -

Normal issue normal execution: This is the base policy used by simplescalar [4] and all values reported are normalized with respect to this scheme. The selection mechanism employed here is the oldest first policy in which a higher priority is given to load and branch instructions. There are no slow FUs used in this technique.

Normal issue slow execution: In this scheme, the issue policy

remains the same (as above) but we give a higher priority to ROD instructions to execute on the slow FU.

Early issue normal execution: In this scheme, we change the selection policy and give a higher priority to *all* ROD instructions. This scheme gives us an estimate of how a different selection policy affects performance and power. Since no slow FUs are used, any power changes incurred with this policy are due to effects induced by the early issue of ROD instructions. NROD instructions are not likely to be excessively starved of resources due to this policy since the ratio of ROD to NROD instructions is roughly 1:3.

Early issue slow execution: To reduce the waiting time of ROD instructions in the IW, we use an early selection policy so that ROD instructions are given higher priority and hence are issued earlier than normal. Further, to obtain energy gains these ROD instructions are given higher preference to execute on the slow low power FU. Early issue with slow execution of non critical ROD instructions ensures that performance degradation does not exceed acceptable levels.

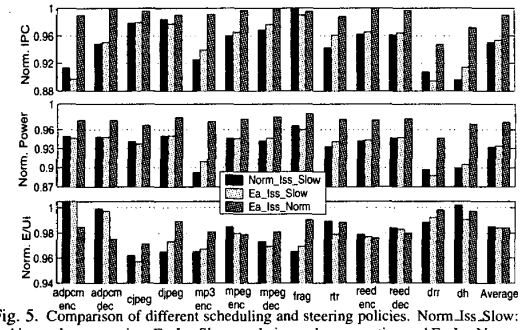

Fig. 5. Comparison of different scheduling and steering policies. Norm_Iss_Slow: Normal issue slow execution, Ea_Iss_Slow: early issue slow execution and Ea_Iss_Norm: early issue normal execution. All values are normalized w.r.t the base i.e. normal issue normal execution policy.

Figure 5 shows the impact of the above scheduling and steering policies on IPC, power and energy per useful (committed) instruction (denoted by E/UI). We observe that the *normal issue slow execution* policy (bar 1) results in 7% average power reduction with about 5% degradation in throughput. The *early issue slow execution* policy (bar 2) results in lower performance degradation for most benchmarks and performs slightly better. The *early issue normal execution* scheme (bar 3) results in the least IPC degradation (average of less than 1%) and yields 3.5% average power gains. This implies that, early issue of ROD instructions is more power efficient than the normal oldest first selection policy. This is due to the fact that early issue of ROD instructions allows dependent branch instructions to be resolved earlier and therefore leads to fewer wrong path instructions from entering the pipeline. This is confirmed by the E/UI metric which is reduced by an average of 2%. The IPC degradation in *adpcm* is very large due to the poor branch prediction accuracy for this benchmark and results in significant energy wastage over wrong path activity (note that the average values are skewed by this ill behaved benchmark). These results highlight the fact that delayed execution of instructions reduces the number of wrong path instructions that enter the pipeline thereby reducing energy wastage. This is achieved in an indirect manner through a combination of IW stalls and slow initiation of instructions into the pipeline. The relatively small improvements in energy savings are due to (i) the usage of only 1 slow low power ALU FU and (ii) less than 25% of all instructions are subjected to delayed execution (this is the average number of ROD instructions in the benchmarks evaluated).

V. Conclusions

In this paper we have analyzed the delay and criticality characteristics of instructions based on their readiness during dispatch. This evaluation of instructions as a collective set rather than individual behavior obviously reduces accuracy but provides useful insights into the kind of instructions one has to focus on for future designs. For example, our analysis indicates that as microarchitectures become larger and branch prediction accuracy improves, only a fraction of NROD instructions contribute to criticality. We have shown that instructions that are normally ready-on-dispatch (ROD) suffer from delayed selection in an oldest first issue policy and that close to 40% ROD instructions spend more than 3 cycles in the IW. These delay and criticality features of ROD instructions are exploited to achieve power benefits by issuing them to slow low power functional units. Results indicate that using slow FUs reduces energy wastage incurred in executing wrong path instructions by an average of 2%. The early issue of ROD instructions with normal execution is the best scheme and results in 3.5% power savings with less than 1% degradation in throughput. Other schmes involving slow FUs incur about 5% average performance degradation and yield as much as 12% (7% average) power savings.

We would like to emphasize that the primary goal of this paper was to bring out the properties of ROD/NROD instructions (e.g. how they spend time in the instruction window). The techniques suggested to obtain power savings are *devised to exploit these properties* and are not necessarily the best schemes. Future work will concentrate on developing better techniques that exploit "delays" faced by instructions in the IW to enable better power-performance trade-off.

References

[1] Tiwari. V et. al.: Power Analysis of Embedded Software: A First Step Toward Software Power Minimization, IEEE Trans. on VLSI Systems, vol. 2, no. 4, pp. 437-445, April 1994.

[2] Brooks. D.M et. al.: Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors, IEEE MICRO, vol. 20 No. 6, (2000).

[3] Brooks.D et. al: Wattch: A Framework for Architectural-Level Power Analysis and Optimizations, In Proc. of ISCA-27, (2000).

[4] Burger.D, Austin.T, Bennett.S: Evaluating Future Microprocessors: The SimpleScalar Tool Set, Technical report - CS-TR-1996-1308, (1996).

[5] Hughes.C.J et, al: Variability in the Execution of Multimedia Applications and Implications for Architecture In Proc. ISCA-28, (2001).

[6] Suzuki.K et. al.: V830R/AV: Embedded Multimedia Superscalar RISC Processor, IEEE Micro, vol. 18, no. 2, pp. 36-47, March 1998.

[7] Fritts. J: Architecture and Compiler Design Issues in Programmable Media Processors, Ph.D. Thesis, Dept. of Electrical Engineering, Princeton University, 2000.

[8] Guthaus.M.R et. al: MiBench: A Free, Commercially Representative Embedded Benchmark Suite, Proc of 4th Workshop on Workload Characterization, (2001).

[9] Wolf.T et. al.: CommBench - A Telecommunications Benchmark for Network Processors, Proc. Intn'l Symp. on Perf. Analysis of Systems and Software, (2000).

[10] Folegnani.D, González.A: Energy-Effective Issue Logic, In Proc. ISCA-28, (2001).

[11] Fields.B et. al: Focusing Processor Policies via Critical-Path Prediction, Proc. of ISCA-28, (2001).

[12] Seng.J.S et. al: Reducing Power with Dynamic Critical Path Information, Proc. of MICRO-34, (2001).

[13] Weglarz.E.F et. al.: Minimizing Energy Consumption for High-Performance Processing, ASPDAC, (2002).

[14] Min.R et. al.: Dynamic Voltage Scaling Techniques for Distributed Microsensor Networks, In Proc. IEEE Workshop on VLSI, (2000).