Low-Power, Low-Complexity Instruction Issue Using Compiler Assistance

Madhavi G. Valluri‡ Lizy K. John‡ Kathryn S. McKinley†
‡Department of Electrical and Computer Engineering
†Department of Computer Sciences
The University of Texas at Austin, TX 78712
{valluri,ljohn}@ece.utexas.edu,mckinley@cs.utexas.edu

ABSTRACT

In an out-of-order issue processor, instructions are dynamically reordered and issued to function units in their data-ready order rather than their original program order to achieve high performance. The logic that facilitates dynamic issue is one of the most power-hungry and time-critical components in a typical out-of-order issue processor.

This paper develops a cooperative hardware/software technique to reduce complexity and energy consumption of the issue logic. The proposed scheme is based on the observation that not all instructions in a program require the same amount of dynamic reordering. Instructions that belong to basic blocks for which the compiler can perform near-optimal sche- duling do not need any intra-block instruction reordering but require only inter-block instruction overlap. In contrast, blocks where the compiler is limited by artificial dependences and memory misses require both intra-block and inter-block instruction reordering. The proposed Reorder-Sensitive Issue Scheme *utilizes a novel compile-time analyzer to evaluate the quality of schedules generated by the static scheduler and to estimate the dynamic reordering requirement of instructions within each basic block. At the micro-architecture-level, we propose a novel issue queue that exploits the varying dynamic scheduling requirement of basic blocks to lower the power dissipation and complexity of the dynamic issue hardware.*

An evaluation of the technique on several SPEC integer benchmarks indicates that we can reduce the energy consumption in the issue queue on average by 72% with only 5% performance degradation. Additionally, the proposed issue hardware is significantly less complex when compared to a conventional monolithic out-of-order issue queue, providing the potential for high clock speeds.

1. INTRODUCTION

Modern high-performance microprocessors are typically centered around a sophisticated out-of-order execution core that can issue multiple parallel instructions each cycle. An integral component of the out-of-order execution core is the issue queue (IQ) which holds decoded instructions while they await their operand values. In each cycle, the queue selects a few instructions whose input dependences have been satisfied and issues them to the function units. A large issue queue is desirable since it allows the dynamic scheduler to examine larger portions of the instruction stream for independent instructions. However, cycle-time and power dissipation constraints restrict the actual size of the queue that can be realized in hardware. The dynamic issue logic is predicted to be a key limiting factor of clock speed in future processors [21]. Furthermore, power dissipation and energy consumption are first-order constraints in the design of today's microprocessors. Due to its highly associative nature, the issue logic is one of the most power-hungry units on the processor core today [12]. The power dissipation of this hardware grows quadratically with processor issue width and is projected to expend a significant portion of the processor's energy budget in future processors [17, 31].

This paper presents a cooperative hardware/software technique to mitigate the power and complexity bottlenecks in the issue logic. The scheme proposed in this paper attempts to harness the work done by the static compile-time instruction scheduler.

The instruction scheduling phase in the compiler reorders and packages instructions into groups of parallel instructions. The function of the static scheduler is identical to that of the dynamic hardware scheduler. Generally, static schedules are of an inferior quality when compared to dynamic schedules since the compile-time scheduler is limited by unknown memory latencies, unknown control flow, limited architectural registers, unresolved memory aliases, etc, in the program. However, on examining several SPEC2000 benchmarks, we find that although portions of general purpose programs suffer from the above impediments, a significant number of basic blocks are free of any memory misses, false-dependences or unresolved alias edges. Consequently, the static schedules of these blocks are near-optimal, and are not very different from their corresponding dynamic schedules. A simple example is shown in Figure 1.

Consider the case where the compiler is unable to resolve the addresses of two potentially independent load and store operations statically (case 1 in the figure). The instruction scheduler adds a data-dependence edge between these operations and forces them to execute sequentially. The dynamic scheduling issue logic however can dynamically disam-

ICS 'O5 June 20–22, Boston, MA, USA

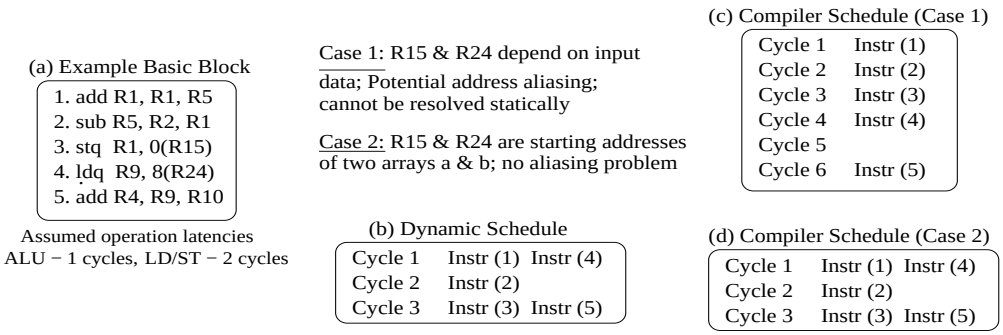

Figure 1: Illustrative Example. (a) Example basic block (b) Dynamic schedule (c) Compiler schedule when the block has an unresolved memory dependence and (d) Compiler schedule when there are no unresolved memory dependences.

biguate the addresses using run-time information to schedule the load operation ahead of the store if they are independent. The compiler-generated schedule (Figure 1(c)) is significantly worse than the schedule achieved by the dynamic issue logic (Figure 1(b)) for this case. On the other hand, if we consider the case wherein the compiler is able to resolve the addresses statically (case 2), there is no difference between the compiler-generated schedule (Figure 1(d)) and the dynamic schedule. Thus, in blocks where the compiler has perfect knowledge, instructions do not need dynamic reordering and could potentially be issued in their statically scheduled order. Furthermore, in today's wide-issue processors, it is not sufficient if the instructions within basic blocks are scheduled perfectly. Instructions from different blocks must be overlapped to fill the available issue slots.

Thus, we note that instructions in a program require two forms of reordering namely *intra-block reordering* and *inter-block overlap*. Blocks in which the compiler can perform near-optimal scheduling do not need any intra-block reordering but require inter-block overlap to limit performance loss. Blocks where the compiler is limited by artificial dependences and memory misses will require both intra-block and inter-block reordering. Based on the inherent reordering requirement of the instructions in each basic block of the program, this paper develops a novel Reorder-Sensitive instruction issue mechanism to reduce the energy consumption and complexity of the issue queue. The proposed scheme is a cooperative hardware/software scheme involving both feedback-directed compiler analysis and micro-architectural innovation.

In the reorder-sensitive issue scheme, the compiler, with the help of profile-guided hints, estimates the reorder requirements of each block and classifies it as a *low-reorder required* (LRR) block or a *high-reorder required* (HRR) block. At the micro-architecture level, the scheme uses a low- complexity, low-power, *reorder-sensitive* issue queue that exploits the varying reorder requirements of blocks in the program. The proposed Reorder-Sensitive issue logic consists of a FIFO-based low-reorder issue queue (LR Queue) that provides only inter-block overlap and a small fully associative issue queue which provides both intra- and inter- block reordering. In this scheme, each block in the program is issued to a queue tuned to the block's inherent reordering requirement. The low reorder queue uses the compiler-generated schedules and is thus suited for LRR blocks. HRR blocks are directed to the associative queue. As a natural consequence of the block selection mechanism, we find that long latency instructions such as load misses are isolated to HRR blocks. These blocks thus have low instruction-level parallelism (ILP). The associative queue thereby needs to support only low issue widths and is significantly smaller and less aggressive when compared to a conventional out-of-order issue queue that allows all instructions. The resulting issue logic is thus significantly lower in power and complexity when compared to a conventional issue queue.

We develop an integrated compiler and microarchitecture-level simulator framework called SPHINX to evaluate the proposed technique. SPHINX integrates a detailed out-of-issue processor simulator that is largely based on Simplescalar's *sim-outorder* [4] simulator with the Trimaran [32] compiler framework. SPHINX also includes power models derived from Wattch [3] for the different hardware structures in the processor. Our results show that the reorder-sensitive scheme can reduce both complexity and power of the issue queue without a significant impact on performance.

The remainder of the paper is organized as follows: In Section 2, we give a brief summary of prior work in the area of complexity and energy-effective issue queue design. The details of the proposed reorder-sensitive issue scheme are provided in Section 3. Our experimental framework is explained in Section 4. In Section 5, we present an evaluation of the proposed scheme. We present brief concluding remarks in Section 6.

2. RELATED WORK

The scheme proposed in this paper leverages the compiler to reduce the complexity and energy consumption of the dynamic scheduling logic. There are several key benefits of engaging the compiler.

One important advantage is that the compiler can configure resources more effectively at smaller granularities when compared to run-time hardware techniques. The run-time energy saving schemes typically lower the energy consumption in the issue queue by adapting the size of the queue based on the dynamic requirements of the program [2, 6, 8, 10, 13, 14, 22, 28, 29]. These techniques usually sample measurable metrics such as IPC or issue queue occupancy to estimate program computational demand and to guide adaptation. To ensure accuracy in detecting changes in computational demand, the sampling intervals are usually quite large [10, 22, 13]. In contrast, compile-time approaches can adapt at smaller intervals [14, 28, 29]. For example, the tech-

niques suggested by Unsal *et al.* [28] and Valluri *et al.* [29] perform compiler-controlled adaptation at loop boundaries. Jones *et al.* [14] resize the queue at at basic-block boundaries using compiler hints. The scheme proposed in this paper reuses static schedules at basic block granularities to save power and reduce complexity.

Similar to the scheme proposed in this paper, our previous work also suggested a technique that can harness compiler generated schedules [29]. In the Hybrid-Scheduling scheme, instructions belonging to well-structured regions of programs such as loops bypass the dynamic scheduling logic and issue directly in their statically-scheduled order, *i.e.*, in a VLIW fashion. Unlike the low reorder issue mode proposed in this paper, the low power VLIW mode provides *no* support for dynamic reordering. The Hybrid-Scheduling approach is therefore only applicable for regular programs such as media and floating-point where a significant portion of the available instruction-level parallelism (ILP) is visible and exploitable at compile-time [29]. General-purpose integer programs on the other hand, contain short basic blocks and require some dynamic reordering between basic blocks to limit performance loss. The reorder-sensitive scheme proposed in this paper is thus particularly suited for general-purpose programs.

Previous techniques have suggested reusing schedules created by the dynamic issue logic [11, 20, 27]. In these schemes, schedules previously created by the dynamic issue logic are issued directly to the function units without any dynamic reordering. However, the disadvantage of these techniques is that large cache-like structures are required for capturing the schedules at run-time. The scheme proposed by Talpes *et al.* also requires a significantly larger physical register file [27]. Seng *et al.* [25] suggest a technique to reduce power by using an in-order queue and an out-of-order issue queue for critical and non-critical instructions respectively. The scheme does not reuse compiler schedules or support. Instructions are steered to different issue queues based on dynamic critical path analysis. The TRIPS architecture also simplifies the issue logic and takes advantage of compiler knowledge to place instructions on a grid of ALUs [5]. However, the impact of this simplification on power and energy consumption is yet to be evaluated.

Another direction of related research focuses on designing complexity-effective issue queues. A majority of the previously proposed techniques reduce complexity of the issue queue by limiting the number of candidate instructions to be considered for issue [1, 7, 9, 18, 19, 21, 23, 16]. These techniques typically consist of a *pre-scheduling* phase wherein the data-dependences of instructions are analyzed. Instructions are typically held in a separate buffer and are considered for issue in their approximate data-flow order [1, 7, 9, 18, 19, 23, 16]. In some cases, after the pre-scheduling phase, instructions are steered to different low-complexity FIFOs based on their dependences with older instructions in the queues [1, 21, 23]. While these techniques alleviate the complexity of the issue logic, they often require extra hardware and/or the addition of a few pipeline stages. An advantage of employing the compiler is that it exposes mechanisms to simplify the hardware beyond what is achievable with dynamic schemes. In the proposed Reorder-Sensitive issue scheme, since the necessary analysis is performed statically, we shift the burden to the compiler and thereby eliminate the need for any auxiliary hardware resources or pipeline stages. To the best of our knowledge, our work is the first to suggest a compile-time approach to reduce the complexity of a conventional out-of-order issue queue.

3. THE REORDER-SENSITIVE INSTRUCTION ISSUE MECHANISM

Figure 2 presents a high-level view of the proposed scheme. The reorder-sensitive issue scheme has both software and hardware components. At the software-level, the scheme uses a compile-time analyzer that evaluates the quality of schedules generated by the static scheduler, estimates the reordering requirement of instructions within each basic block, and classifies blocks as LRR or HRR blocks. At the hardware level, there are two separate low-power, low-complexity queues for the two types of blocks. Blocks are directed to different queues based on their reordering requirement. We first discuss the software component of the scheme.

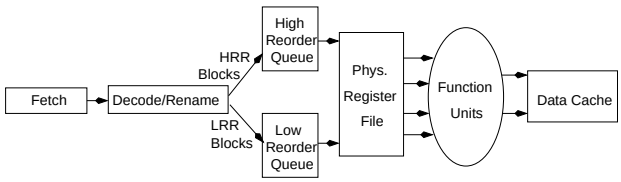

Figure 2: High-level view of the proposed reorder-sensitive instruction issue mechanism

3.1 Estimating the Inherent Reordering Requirement of Basic Blocks

For a given block of instructions, even the most advanced static schedulers fail to produce optimal schedules in the presence of the following impediments: (a) false-dependences (b) unresolved memory aliases and (c) non-uniform load latencies. Instructions within such blocks will have to be reordered dynamically to hide pipeline stalls. This section describes in detail how the compiler, with the help of profile-time statistics, evaluates the impact of each of the above constraints on the schedule quality of each basic block.

3.1.1 Impact of Anti- and Output- Dependences on Static Schedules

Due to the limited number of available architectural registers, the register allocator often assigns the same register to independent instructions. This reuse of registers results in anti- and output- dependences, which in turn limits the instruction scheduler's reordering opportunities. Dynamically scheduled processors do not suffer from this limitation since they are able to employ dynamic register renaming to remap architectural registers to hardware physical registers. With register renaming, each instruction entering the pipeline is assigned a new physical register. The number of hardware physical registers in a processor is considerably higher than the number of architectural registers exposed to the static scheduler (more than twice). Elimination of false-dependences allows the out-of-order issue scheduler greater flexibility in selecting instructions for issue each cycle.

The degradation of the schedule quality due to false register dependences can be estimated by comparing the length of the critical path in the block and the actual schedule achieved by the static instruction scheduler. The critical path is computed by considering only true dependences and

hence is the minimum schedule length of the block. If the anti- and output- dependence edges impact the schedule, the length of the schedule for the block ($Sched\_len$) will be larger than the critical path (CP_{max}). Such a block will clearly require dynamic reordering support. The degradation in schedule quality caused by anti- and output- dependences (SD_{fd}) can be estimated as:

$$SD_{fd} = \frac{Sched\_len - CP_{max}}{CP_{max}} * 100 \quad (1)$$

Note we conservatively assume that with a large enough physical register file, the dynamic issue logic can remove all the false dependences.

3.1.2 Impact of Unresolved Alias Edges

When the compiler is unable to resolve the addresses of two memory operations, it adds a dependence edge between the operations and forces them to execute sequentially. A given static schedule may perform poorly compared to its equivalent dynamic schedule if dependences that do not occur during the actual execution of the program are respected by the compiler.

To estimate the impact of alias edges on the static schedule, we collect profile information of all the memory dependences within a block that occur during program execution. For every pair of memory operations that are dependent each time they occur during execution, we convert the memory dependence edge to a true data-dependence edge during the computation of the critical path. The critical path then reflects this true memory dependence. Consequently, Equation 1 estimates the potential schedule degradation due to both false data dependences and false memory dependences.

3.1.3 Impact of Cache Misses

Load misses pose a serious performance bottleneck to in-order issue of instructions since it is difficult for the compiler to find sufficient instructions in a single basic block to fill all the issue slots, particularly in today's superscalar processors with wide issue and large cache miss latencies. To estimate the cost of cache misses (SD_{cm}), we profile the average number of L1 misses ($L1_{misses}$) and L2 misses ($L2_{misses}$) per block. Hence, the estimated performance degradation is given by:

$$SD_{cm} = \frac{L1_{misses} * L1_{eff\_cost} + L2_{misses} * L2_{eff\_cost}}{CP_{max}} * 100 \quad (2)$$

The *effective cost* in Equation 2 represents the fraction of the cache miss latency that the out-of-order issue logic can hide. The effective cost is different from the actual latency of a cache miss. It is a complex function of miss latencies, issue queue size, issue width and available ILP in the program. For example, larger issue queues can potentially hide a larger fraction of the miss stalls. In our experiments, we empirically identify the effective cost of a cache miss seen by an instruction in the LR queue.

3.1.4 Selecting and Annotating LRR Blocks

To select blocks, the compiler sets certain threshold values for acceptable schedule degradations. A block is selected as an LRR block if both SD_{fd} and SD_{cm} are less than their respective threshold values FD_{th} and CM_{th}. Similar to Jones *et. al.,* [14], we assume that the compiler annotations are conveyed to the hardware using marker instructions. We need one marker instruction for each basic block. We assume that the marker instructions are flushed after decode and do not enter the rest of the pipeline. Figure 3 shows a complete summary of the compile-time analysis.

```
Profile Collection Phase
for each basic block {
  record L1 miss rate, L2 miss rate;
  record pairs of dependent memory operations;
}

Profile Analysis Phase
for each basic block {
    for each 'always_occurs' memory edge {
        convert to true dependence edge;
    }
    Compute CP_max, SD_fd, SD_cm;
    if (SD_fd < FD_th && SD_cm < CM_th)
        Select block as LRR block;
    else
        Select block as HRR block;
    Annotate block with marker instruction;
}
```

Figure 3: Block selection algorithm

3.2 Reorder-Sensitive Issue logic

The previous section showed how the compiler identifies blocks that require less dynamic scheduling. In this section, we present an issue queue design that exploits the varying requirement of basic blocks to significantly alleviate the complexity and power of the issue logic. The proposed Reorder-Sensitive issue logic has two types of queues: namely, the LR (low reorder) queue and the HR (high reorder) queue.

The LR queue consists of several FIFO buffers, with each buffer as wide as the average ILP available in a basic block. At run-time, each new LRR block is directed to one of these buffers. For each LRR block, a FIFO is selected in a round-robin fashion. Instructions can be written only into the tail pointer of each buffer and can only be read from the head of the queue. In each cycle, instructions from the heads of the FIFOs can be selected for execution. Since there are multiple queues, instructions from different LRR blocks can be overlapped (Figure 4). Instructions in each basic block are therefore issued in their statically scheduled order but are overlapped with instructions from successive basic blocks (Note, the LR queue is conceptually similar to the region-slip-enabled issue buffer proposed by Spadini *et al.* [26]. Their proposed mechanism uses a FIFO-based issue buffer that allows a block's schedule to 'slip' into the schedule of a previous block. However, the disadvantage of the region-slip buffer is that it is a monolithic structure requiring a large number of entries and ports, making the power consumption of the buffer excessively high.).

The number of FIFOs in the LR queue required will be different for different architectures and can be chosen at design time based on the extent of overlap seen during execution for the target workloads. For example, on examining several SPECint benchmarks executing on an 8-way conventional out-of-order issue processor with a 128-entry issue queue, we found that in nearly 70% of execution cycles, instructions are issued from at most three consecutive basic blocks. Thus, three FIFO buffers can capture a significant portion of the required inter-block overlap between LR blocks.

The HR queue in the reorder-sensitive issue scheme caters to instructions that require full dynamic scheduling and hence is fully associative similar to the conventional out-of-order is-

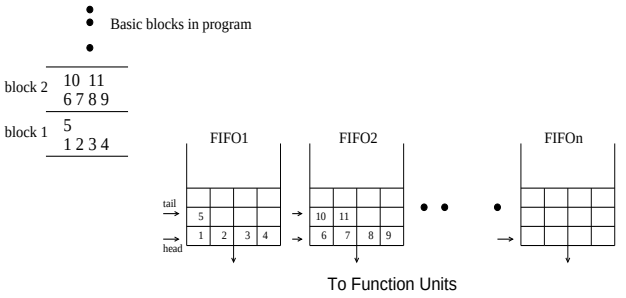

Figure 4: Internal view of the Low-Reorder (LR) issue queue

sue queue. We observe that in the block selection algorithm, blocks with a large number of cache misses are automatically deemed as HRR blocks. In consequence, we find that the available ILP in these blocks is limited. The HR buffer can thus be small and have a less aggressive issue width.

In each cycle, instructions are selected from either the out-of-order issue queue or the heads of the FIFOs for execution. The total number of instructions issued in each cycle is limited to the issue width of the processor.

The complexity of the reorder-sensitive issue logic is significantly lower than a conventional broadcast-based fully associative issue queue. The inherent complexity of the LR queue is inherently low since it is FIFO based. There are no complex CAM structures or global signals for instruction wake-up. Unlike the conventional issue window where result tags need to be broadcast to all the entries, in the LR queue, register availability only needs to be fanned out to the heads of the FIFOs [21]. Further, since each block in the program is classified and annotated by the compiler, there are practically no extraneous hardware structures required for steering instructions to different queues. Also, since the instructions in the LR buffers are issued in their statically-scheduled order, there is no extra prescheduling phase usually required in other dependence-based schedulers. The complexity of the wakeup logic in the HR queue is also significantly lower since the issue width of the queue is small. Since the HR queue is small and the number of FIFOs is not high, the arbitration logic has to select from a smaller number of candidate instructions compared to conventional scheme. Thus we see that both the LR queue and HR queue are small and simple structures. The power dissipation of the reorder-sensitive issue logic is consequently significantly lower than a conventional out-of-order issue queue.

4. EXPERIMENTAL SETUP

We develop an integrated compiler and microarchitecture-level simulator framework called SPHINX to evaluate the proposed technique. An overview of the framework is presented in Figure 5.

4.1 The SPHINX framework

SPHINX integrates a detailed out-of-issue processor simulator that is largely based on Simplescalar's *sim-outorder* simulator with the Trimaran 2.0 [32] compiler framework. Trimaran provides a rich support structure for developing state-of-the-art optimizations geared towards high performance architectures. We also incorporate power models derived from Wattch [3] in the simulator to estimate power. The SPHINX framework thus provides a unified platform for exploring a range of software and hardware techniques for low power. We briefly describe each of the modules in the framework.

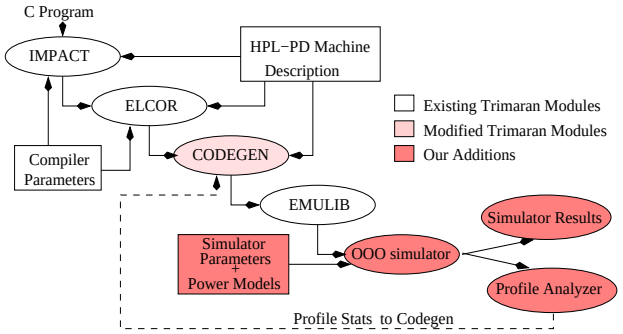

Figure 5: SPHINX compiler/simulator framework

IMPACT and ELCOR are the Trimaran front-end and back-end phases respectively. Trimaran is built around a parameterized ILP architecture called HPL-PD. A machine description language, called HMDES, allows the user to develop a machine description for the HPL-PD processor in a high-level language, which is then translated into a low-level representation for efficient use by the compiler. The HPL-PD machine description includes detailed formats of the rich set of instructions support by the HPL-PD architecture. Several machine description aspects can be varied including the register file type and size, number and type of function units, operation latencies etc. We use the machine description language to define a contemporary superscalar processor. Table 1 shows some of the important high-level HMDES parameters and their values. The code generator (CodeGen) module converts the program from its intermediate program representation into instructions that can be executed on a virtual HPL-PD machine. We implement the instrumentation code for profiling memory dependences and cache misses per block within this module. The emulation library (Emulib) of Trimaran contains an interpreter and a set of emulation routines for the HPL-PD virtual machine. We use the interpreter to generate a trace of instructions that feed into our detailed out-of-order issue simulator.

We model a detailed out-of-order issue superscalar simulator (OOO Simulator) within SPHINX. The simulator is a heavily modified version of Simplescalar's *sim-outorder* simulator. We modified the simulator to support HPL-PD instructions. Further, the Register Update Unit (RUU) in sim-outorder is replaced with a separate issue queue and physical register file. We also incorporate the branch predictor and cache modules used in *sim-outorder* in SPHINX. The power/energy estimates in the simulator are based on the suite of parameterizable power models in Wattch 1.0 [3]. The profile analyzer implements our block selection algorithm. It examines the profile statistics collected and identifies LRR/HRR blocks. This information is then provided to the CodeGen module so that the blocks can be annotated with the appropriate marker instructions.

[1]Although we support predication in our framework for completeness and for future research, none of our benchmarks used predicated instructions, mainly because we use basic-blocks not hyperblocks.

[2]BTR registers are accessed only by special *prepare-to-branch* (PBR) in the HPL-PD architecture. PBR instructions are typically used for provide hints regarding branch

Table 1: High-Level HMDES parameters

Feature	Attributes
Integer GPRs	32
Floating Point GPRs	32
Predicate Registers[1]	32
Branch Target Registers[2]	16
Rotating Registers	None
Control Registers	None
Function Units	Similar to Table 2

Table 2: Baseline Processor Configuration

Feature	Attributes
Issue Units	IQ/LSQ - 128/64 entries 128 physical registers Fetch/Decode/Issue/Commit width - 8
Cache Hierarchy	32KB 4-way L1 Dcache (2-cycle hit) 32KB DM L1 Icache (1-cycle hit) 512KB 4-way L2 (20-cycle hit)
Memory	150 cycles memory latency
Branch Pred.	4K Gshare 22 cycles extra misprediction latency
Function	6 integer ALUs, 2 FP ALUs Units 4 integer multiply units 1 FP multiply units,2 load/store units

4.2 Benchmarks

The scheme is evaluated for several SPEC95 and SPEC2000 integer benchmarks. We particularly target the integer programs since they are less amenable to compile-time optimizations due to the presence of hard-to-predict branches, pointer-intensive memory accesses and extensive use of function and library calls. The benchmarks used in the study are shown in Table 3. We use *true* profiling for the evaluation, *i.e.*, the input data used by the compile-time analyzer to classify blocks is different from the input set used to evaluate the proposed technique. MinneSPEC reduced inputs [15] are used where applicable. For *vortex*, we skip the first 100 million instructions and simulate one billion instructions.

5. EXPERIMENTAL EVALUATION

This section presents a detailed evaluation of the reorder-sensitive issue scheme. We first present the power dissipation and performance results.

5.1 Power and Performance Results

The baseline out-of-order issue processor has an 8-wide, 128 entry issue queue. The configuration of the baseline processor is detailed in Table 2. We choose a reorder-sensitive configuration to match the characteristics of typical general-purpose programs. We measured the degree of overlap in the SPEC integer benchmarks. Table 4 shows the maximum distance between instructions issued in one cycle in terms of the number of basic blocks. We observe that in nearly 75% of execution cycles, instructions are issued from at most four consecutive basic blocks. Thus, we use four 3-wide 8-entry FIFO buffers. We restrict the width of each FIFO to three entries, since in most blocks, the ILP was not higher than 3. Having 8-rows of 3 instructions each in each row provided 24-entries in a FIFO. In contrast to the baseline processor, 75% of the entries are thus in FIFOs. The remaining 25% of the entries are in a 4-wide, 32-entry associative issue queue. This is significantly less aggressive and smaller when compared to the baseline issue queue.

[1] prediction to the target architecture. The BTR file contains only replicates of values that are already held in the general-purpose register (GPR) file and does not extend the GPR file in any way.

Table 4: Instruction overlap from different blocks

# of blocks	≤ 1	≤ 2	≤ 3	≤ 4	≤ 5	≤ 6	≤ 8
% cycles	52.6	61.4	70.1	74.7	79.7	82.7	100

Table 5 shows the percentage of instructions issued from the LR queue. We set the block selection thresholds at 5%(FD_{th}) and 0.1%(CM_{th}). In Section 5.2, we present an evaluation of the technique for different threshold values. We observe that on an average, 30% of the instructions are issued from the LR queue.

Figure 6, 7 and 8 show normalized execution time, issue queue energy consumption and total energy consumption of the reorder-sensitive scheme with respect to the baseline out-of-order issue queue. Note, the energy and cycle-time overheads for fetching marker instructions are included in all our results.

Table 5: Percentage of instructions issued from the LR queue in the reorder-sensitive issue scheme

compress	gzip	li	mcf	parser	vortex	vpr	AVG
32.34	37.45	21.64	33.63	19.39	19.35	45.40	30.20

We observe that the baseline reorder-sensitive issue scheme dramatically reduces the issue queue energy consumption by 72% on average. Several factors contribute to the overall energy reduction. First, as seen in Table 5, nearly 30% of the instructions are issued from the LR FIFOs. The FIFOs consume low power since they require very few ports. In our scheme, we assign two write ports to each FIFO since the fetch width of the processor is twice that of the FIFO width. Further, since instructions are issued only from the head of the FIFO, it needs only one write port. The 32-entry HR queue in the reorder-sensitive architecture is also significantly smaller than the baseline 128-entry issue queue. More importantly, the issue width of the HR queue is half that of the baseline issue queue. The HR queue thus has significantly fewer number of ports than the baseline 8-wide issue queue. The number of dispatch (*i.e.,* writing to the issue window) ports on the HR queue is the same as in the baseline queue since the fetch width of both configurations is the same. However, the issue and wakeup (result-broadcast) ports are only four each. All the above factors contribute towards reducing the power of the reorder-sensitive issue logic. Large savings in the issue queue also lead to considerable overall power savings. The total power savings seen in the reorder-sensitive scheme is close to 18% as shown in Figure 8.

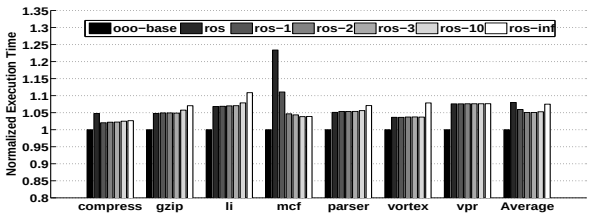

Figure 6: Normalized execution time

Figure 6 shows the performance results of the reorder-sensitive scheme. The performance degradation seen by the base reorder-sensitive issue (ros in the figures) scheme is

Table 3: Benchmarks and inputs. Minnespec reduced input sets used where applicable.

Benchmark	compress	gzip	li	mcf	parser	vortex	vpr
Profile Input Set	train	minnespec train	test	minnespec train	minnespec train	test	minnespec train
Inputs for Perf. Evaluation	220000 q 2131	minnespec ref	train	minnespec ref	minnespec ref	ref	minnespec ref

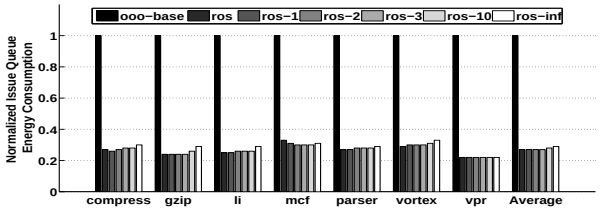

Figure 7: Normalized issue queue energy

8%. Our block selection heuristics are guided by localized estimates we make at the basic block level. Dependences between basic blocks, especially, between LRR blocks and HRR blocks are critical but are not captured by localized heuristics. When a cache miss is serviced, the instructions waiting on the data can issue immediately if they are either in the out-of-order issue queue or in the heads of FIFOs. However, if there many dependences between LRR blocks and HRR blocks with cache misses, it is unlikely that we will find all the dependent instructions at the heads of the FIFOs.

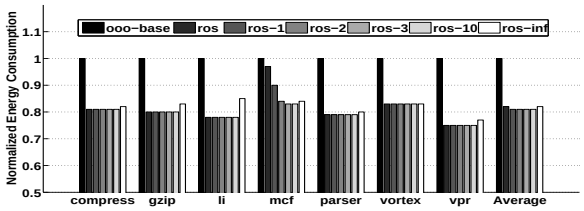

Figure 8: Normalized total energy consumption

Based on this observation, we provide a further improvement to the reorder-sensitive scheme, wherein for each HRR block with a large number of cache misses, we direct a few subsequent consecutive blocks into the out-of-order issue queue irrespective of whether they are set to be LRR blocks or HRR blocks. This helps capture the immediate dependences between LRR blocks and HRR blocks with a large number of cache misses (note all HRR blocks need not have high miss rates; some blocks are classified as HRR blocks due to false dependences).

The number of blocks redirected to the HR queue is controlled by a simple saturating counter whose maximum value can be determined and set for each benchmark. In Table 6, we show the average distance between dependent instructions in number of basic blocks. We observe that nearly all (97%) the data dependences between instructions extend to approximately 3 basic blocks. Hence, the value of saturating counter can be in the range of 1-3 blocks, ensuring that almost all dependences between HRR and LRR blocks are captured in the HR queue. Figure 6, 7 and 8 include the performance, issue queue energy savings and total energy savings of the reorder-sensitive scheme for different values of the saturating counter. We find that allowing one additional block improves performance in many benchmarks. Increasing the counter value to 2 is sufficient to capture almost all dependences and improves performance further. The benchmark *mcf* particularly shows a good improvement. Since *mcf* has an extremely poor cache hit rate, a significant fraction of dependences are critical in this benchmark, A bigger counter value captures more dependences and hence performs better. Increasing the counter value to 3 and beyond begins to provide diminishing returns. In fact, increasing the counter value to a very large value begins to negatively impact performance and issue queue energy since now too many blocks are diverted to the small HR queue leading to increased dispatch stalls in the pipeline. Figure 6 shows that the overall performance degradation reduces from 8% to 5.9% for 2 blocks and to 5% for 3 additional blocks directed to the HR queue. The percentage of instructions issued from the LR queue reduces slightly from 30% to 26% on an average (for 2 blocks) and to 25% for 3 blocks. Note that even for the best saturating counter value, there is performance loss in the reorder-sensitive scheme. This is because of two main reasons: (a) the LR queue has a limited number of FIFOs, it does not provide the ideal amount of inter-block overlap. Additionally, (b) not all of the HRR blocks are low ILP blocks. Some blocks that do not suffer from cache misses are also directed to the HRR queue. These blocks are unnecessarily penalized by the low issue HR queue.

Table 6: Dependence distance in terms of number of basic blocks

# of blocks	$= 0$	≤ 1	≤ 2	≤ 3	≤ 4	≤ 7	≤ 10
% dependences	83.6	92.6	96.0	97.5	98.4	98.7	99.9

In Figure 9, we compare the reorder-sensitive scheme with 4 and 8-wide conventional queues for varying issue queue sizes. The results demonstrate that reorder-sensitive configurations can consistently achieve high performance (within 5% of an 8-wide out-of-order issue queue), while consuming significantly lower energy (close to a 4-wide out-of-order issue queue).

5.2 Evaluation of the Block Selection Heuristics

To compute the estimated schedule degradation due to cache misses (SD_{cm}), the compiler requires good estimates for the miss hiding capability of the out-of-order issue logic (Section 3.1.3). In Figure 10 shows how the effective L1 and L2 miss costs can be empirically identified.

We observe that if we assume very small estimated costs, a larger percentage of blocks qualify as LRR blocks and hence a larger number of instructions are issued from the LR queue. However, the performance loss is also prohibitively high, indicating that we are underestimating the miss hiding capability of the out-of-order issue logic. As we progressively increase the assumed costs we find that the performance degradation and the number of instructions issued from the LR queue decrease. We note that assuming an L1 miss cost in the range of 20-30 cycles is a reasonable estimate for the L1 cost, since increasing the estimate beyond

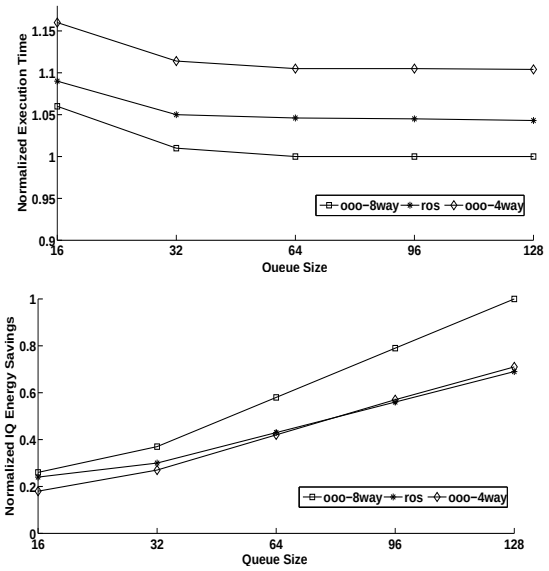

Figure 9: Performance and energy results of different ROS and conventional issue queue configurations for varying queue sizes. For the ROS schemes, queue size corresponds to the HR queue size.

this does not dramatically decrease the performance loss. If we assume that the average IPC of integer programs on an 8-way machine is 3, a 30-cycle cost can be roughly interpreted as the o-o-o issue queue being able to hide about half the L1 miss latency (10 cycles) with 3 instructions being issued each cycle. Note that if we were executing in a fully in-order machine, where it is impossible to hide any stalls due to cache misses at run-time, the cost would have been much higher. However, since the LR queue allows some degree of overlap of instructions, the *effective* cycle-time cost is lower. Another interesting observation from the figure is that the block selection heuristic is largely insensitive to the L2 miss cost. This is because the number of L2 misses are considerably lower than the number of L1 misses. We fix the effective L1 miss cost and L2 miss cost to 30 cycles for all the experiments in the paper.

The compiler selects LRR blocks based on statically computed estimates of the schedule quality degradation due to false dependences (SD_{fd}) and memory misses (SD_{cm}). The compiler classifies a block as an LRR block if its estimated schedule degradation due to false dependences and memory misses (SD_{fd} and SD_{cm}) are less than their respective threshold values (FD_{th} and CM_{th}). Figure 11 presents an evaluation of the sensitivity of our selection policy to varying threshold values. We also show the percentage of dynamic instructions that are issued from the LR queue and the corresponding performance degradation for different FD_{th} and CM_{th} values for the benchmark *compress*. (We show the sensitivity analysis for one benchmark, all other benchmarks show largely similar trends.)

Larger threshold values indicates willingness to tolerate a larger performance loss. With large threshold values the compiler selects more blocks for issue in the LR queue. Since the LR queue is extremely simple and consumes lower power when compared to the HR queue, we can operate in a power-savings mode by setting large threshold values. Alternatively, we can set lower thresholds and operate in a more

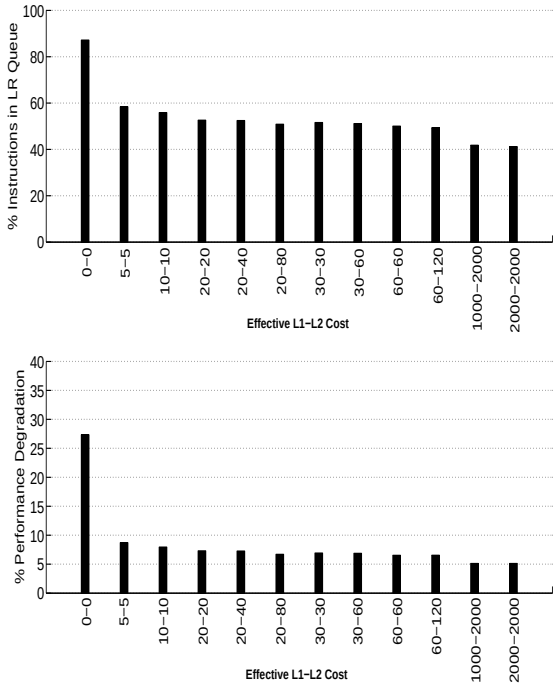

Figure 10: Performance degradation for different L1-L2 miss costs (averaged over all benchmarks). Costs expressed as `X1-Y1`, where `X1` is the effective L1 miss cost and `Y1` is the L2 miss cost in cycles. CM_{th} is fixed at 5%.

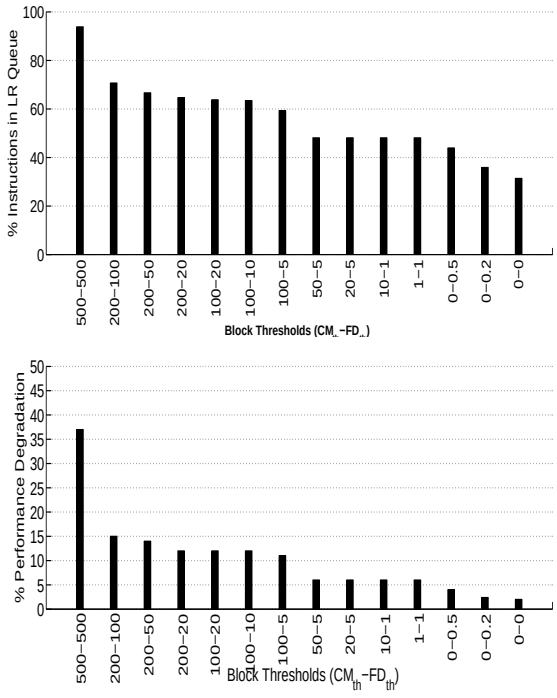

Figure 11: Percentage of instructions in LR queue and performance degradation for varying block selection thresholds

performance-sensitive mode. The actual performance degradation observed is lower than the estimated performance degradation (indicated by the threshold values). The reason for this is that the estimate is computed assuming the blocks will be issued in a strictly in-order fashion. However, since we provide some degree of overlap between blocks in the LR queue, and instructions from the heads of the FIFOs can issue in any order, we hide a good percent of the stall cycles.

An interesting observation from Figure 11 is that even when we set both thresholds to 0, there are a significant number of instructions (30%) that are selected for issue in the LR queue. This indicates that a large number of blocks do not contain any false dependences nor do they experience any memory misses. For these blocks, expending energy to reorder instructions in an out-of-order issue queue is wasteful since the compiler can generate good quality schedules. The reorder sensitive scheme thus provides us with the ability to harness compiler-generated schedules for these blocks.

5.3 Discussion

The reorder-sensitive scheme uses profile-directed feedback to select LRR blocks. We also evaluated the performance of the scheme for a 'compile-only' approach, *i.e.,* when static analysis alone is used for selecting blocks with low reorder requirements. Table 7 shows these results. For this experiment, all blocks without load instructions and false register dependences are chosen as LRR blocks. We observe that the percentage of LRR blocks drops to approximately half of those with profile-based information (see Table 5). Consequently, the performance of the compile-only scheme is lower. Profile information thus helps significantly improve the performance of the reorder-sensitive issue queue. However, as with any profile-based approach, the performance of the scheme depends on how closely a program's actual run-time tendencies match those seen during specialization. Although in our experiments, we observe that the degradation is not significant even when the input data is different from the profile input, it is desirable to have a protection mechanism when the input data deviates from the profile or a profile is not available. Unforeseen cache misses are particularly hazardous since the LR queue is quite limited in its capacity to hide the miss penalty.

Table 7: Percentage of instructions issued from the LR queue and performance degradation of *compile-only* reorder-sensitive issue scheme

Benchmark	compress	gzip	li	mcf	parser	vortex	vpr	AVG
% LRR instrns	16.42	15.31	10.39	23.87	12.77	4.64	21.27	15.5
% Perf. Degrad.	3.1	7.68	9.66	5.01	8.13	5.13	8.28	6.7

One attractive solution is to implement the reorder sensitive scheme within a hardware or software dynamic optimization framework [24, 30]. The dynamic optimizer could monitor the behavior exhibited by the running application and dynamically tune the heuristics to continually divert blocks with cache miss rates to the fully-associative queue which is better equipped to handle misses.

Another approach is to add some global loop-level or procedure level control over our localized issue policy. Prior work has shown that it is possible to predict the IPC of a program statically with reasonable accuracy using simple compile-time analysis [28]. During execution of the program, we could dynamically monitor the IPC of important program hotspots. If the observed IPC is significantly below estimated IPCs, the system can disable the reorder-sensitive issue mode and use the default conventional out-of-order issue queue.

6. CONCLUSIONS

This paper presents an instruction issue mechanism that exploits compiler-generated schedules to lower complexity and power of the issue queue. The proposed reorder-sensitive issue scheme recognizes that all instructions and all blocks in a program are not equal; some blocks are inherently easy to schedule statically, whereas others are not. We describe the compiler analysis required for identifying blocks with distinct dynamic scheduling requirements. At run-time, the scheme uses a small conventional issue window in conjunction with simple FIFO queues and basic blocks are directed to the appropriate issue queue based on their characteristics. We present a detailed analysis of the proposed architecture and the compiler heuristics employed. Our results show that we are able to save up to 72% of the issue queue energy and 18% total energy with only 5% performance degradation. Further, the proposed issue hardware is less complex when compared to a conventional out-of-order issue queue, providing the potential for much higher clock speeds. Another key contribution of this work is the SPHINX framework. With the ever increasing cycle-time and power constraints, it is becoming increasingly important to exploit every available avenue in a computer system to aid the processor. The SPHINX tool thus provides a unified platform for exploring future compiler and micro-architecture-based solutions for energy- and complexity-effective processors.

7. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments. Many thanks also to Rodric Rabbah, Ramdas Nagarajan and Nitya Ranganathan for their help with the Trimaran framework. This research is partially supported by the National Science Foundation (NSF) under grant numbers CCF-0429806, CCR-0311829, ITR CCR-0085792, CISE infrastructure grant EIA-0303609, by DARPA F33615-03-C-410, by IBM Center for Advanced Studies (CAS) awards and an IBM SUR grant.

8. REFERENCES

[1] J. Abella and A. Gonzlez. Low-complexity distributed issue queue. In *Proceedings of the 10th annual International Symposium on High Performance Computer Architecture*, Feb 2004.

[2] R. I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *27th International Symposium on Computer Architecture*, Jul 2001.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th International Symposium on Computer Architecture*, Jun 2000.

[4] D. Burger and T. M. Austin. Evaluating future microprocessors: The simplescalar tool set. Technical report, Dep. of Comp. Sci., Univ. of Wisconsin, Madison, 1997.

[5] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. Scaling to the end of silicon with EDGE architectures. In *IEEE Computer*, July 2004.

[6] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power at high performance. In *Workshop on Power-Aware Computers Systems, held in conjunction with ASPLOS*, Nov 2000.

[7] R. Canal and A. Gonzlez. A low-complexity issue logic. In *Proceedings of the 14th International Conference on Supercomputing*, pages 327–335, 2000.

[8] E. Chi, A. M. Salem, R. I. Bahar, and R. Weiss. Combining software and hardware monitoring for improved power and performance tuning. In *Workshop on Interaction Between Compilers and Computer Architecture (INTERACT)*, Feb 2003.

[9] D. Ernst, A. Hamel, and T. Austin. Cyclone: a broadcast-free dynamic instruction scheduler with selective replay. In *Proceedings of the 30th annual International Symposium on Computer Architecture*, pages 253–263, Jun 2003.

[10] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *28th International Symposium on Computer Architecture*, Jun. 2001.

[11] M. Franklin and M. Smotherman. A fill-unit approach to multiple instruction issue. In *27th Annual International Symposium on Microarchitecture*, 1994.

[12] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *Design Automation Conference*, pages 726–731, 1998.

[13] A. Iyer and D. Marculescu. Run–time scaling of microarchitecture resources in a processor for energy savings. In *Kool Chips Workshop, held in conjunction with MICRO–33*, 2000.

[14] T. Jones, M. O'Boyle, J. Abella, and A. Gonzalez. Software assisted issue queue power reduction. In *Proceedings of the 7th annual International Symposium on High Performance Computer Architecture*, 2005.

[15] A. KleinOsowski and D. J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. In *ACM Computer architecture letters*, 2002.

[16] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th annual International Symposium on Computer Architecture*, pages 59–70, Jun 2002.

[17] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *25th International Symposium on Computer Architecture*, pages 1–10, Jun 1998.

[18] P. Michaud and A. Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, Feb 2001.

[19] E. Morancho, J. M. Llaberia, and A. Olive;. Recovery mechanism for latency misprediction. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, Sept 2001.

[20] R. Nair and M. E. Hopkins. Exploiting instruction level parallelism in processor by caching scheduled groups. In *annual International Symposium on Computer Architecture*, 1997.

[21] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *24th Annual International Symposium on Computer Architecture*, Dec. 1997.

[22] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *34th International Symposium on Microarchitecture*, pages 90–101, Dec 2001.

[23] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt. A scalable instruction queue design using dependence chains. In *Proceedings of the 29th annual International Symposium on Computer Architecture*, pages 318–329, Jun 2002.

[24] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A. Mendelson. Power awareness through selective dynamically optimized traces. In *Proceedings of the 31st annual International Symposium on Computer Architecture*, page 162, 2004.

[25] J. S. Seng, E. S. Tune, and D. M. Tullsen. Reducing power with dynamic critical path information. In *34th Annual International Symposium on Microarchitecture*, Dec. 2001.

[26] F. Spadini, B. Fahs, S. Patel, and S. S. Lumetta. Improving quasi-dynamic schedules through region slip. In *Proceedings of the International Symposium on Code generation and Optimization*, pages 149–158, Mar 2003.

[27] E. Talpes and D. Marculescu. Power reduction through work reuse. In *International Symposium on Low Power Electronics and Design*, 2001.

[28] O. S. Unsal, I. Koren, C. M. Krishna, and C. A. Mortiz. Cool-fetch: A compiler-enabled IPC estimation based framework for energy reduction. In *ACM Computer architecture letters*, 2002.

[29] M. Valluri, L. John, and H. Hanson. Exploiting compiler-generated schedules for energy savings in high-performance processors. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, 2003.

[30] V.Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of Programming Language Design and Implementation*, 2000.

[31] V. V. Zyuban and P. Kogge. Inherently lower-power high-performance superscalar architectures. In *IEEE Transactions on Computers*, pages 268–285, Mar. 2001.

[32] TRIMARAN: An Infrastructure for Research in Instruction-Level Parallelism http://www.trimaran.org/