Compiler Directed Issue Queue Energy Reduction

Timothy M. Jones†, Michael F.P. O'Boyle†,
Jaume Abella‡, and Antonio González‡

†Member of HiPEAC,
School of Informatics
University of Edinburgh, UK
tjones1@inf.ed.ac.uk
mob@inf.ed.ac.uk

‡Intel Barcelona Research Center,
Intel Labs - UPC,
Barcelona, Spain
jaume.abella@intel.com
antonio.gonzalez@intel.com

Abstract. The issue logic of a superscalar processor consumes a large amount of static and dynamic energy. Furthermore, its power density makes it a hot-spot requiring expensive cooling systems and additional packaging. This paper presents a novel approach to energy reduction that uses compiler analysis communicated to the hardware, allowing the processor to dynamically resize the issue queue, fitting it to the available ILP without slowing down the critical path. Limiting the entries available reduces the quantity of instructions dispatched, leading to energy savings in the banked issue queue without adversely affecting performance.
Compared with a recently proposed hardware scheme, our approach is faster, simpler and saves more energy. A simplistic scheme achieves 31% dynamic and 33% static energy savings in the issue queue with a 7.2% performance loss. Using more sophisticated compiler analysis we then show that the performance loss can be reduced to less than 0.6% with 24% dynamic and 30% static energy savings and an EDD product of 0.96, outperforming two current state-of-the-art hardware approaches.

1 Introduction

Superscalar processors contain complex logic to hold instructions and information as they pass through the pipeline. Unfortunately, extracting sufficient instruction level parallelism (ILP) and performing out-of-order execution consumes a large amount of energy, with important implications for future processors.

With up to 27% of the total processor energy consumption being consumed by the issue logic [1], this is one of the main sources of power dissipation in current superscalar processors [2]. Furthermore, this logic is one of the components with the highest power density and is a hot-spot. Reducing its power dissipation is therefore more important than for other structures. Consequently there has been much work in developing hardware schemes to reduce this energy cost by turning off unused entries and adapting the issue queue to the available ILP [1, 3, 4]. Unfortunately, there is an inevitable delay in sensing rapid phase changes and adjusting accordingly. Furthermore, these mechanisms are based on past program behaviour, rather than knowledge of the future. This leads to either a loss of IPC due to too small an issue queue or excessive power dissipation due to too large an issue queue.

This paper proposes an entirely different approach - software directed issue queue control. In essence, the compiler knows which parts of the program are to be executed in the near future and can resize the queue accordingly. It reduces the number of instructions in the queue without delaying the critical path of the program. Reducing the number of instructions in the issue queue reduces the number of non-ready operands woken up each cycle and hence saves energy. We evaluate the energy savings for the issue queue using a simplistic scheme in section 6 and with more sophisticated compiler analysis in section 7.

1.1 Related work

Saving energy by turning off unused parts of the processor has been the focus of much previous work. Bahar and Manne [5] introduce *pipeline balancing* which changes the issue width of the processor depending on the issue IPC over a fixed window size. Other papers [6, 7] propose shutting down parts of the processor in a similar manner with comparable results.

Considering the issue queue alone, Folegnani and González [1] reduce useless activity by gating off the precharge signal for tag comparisons to empty or ready operands. They then suggest ways to take advantage of the empty entries by dynamically resizing the queue. Buyuktosunoglu et al. [3] propose a similar resizing scheme, using banks which can be turned off for static energy savings. Abella and González [4] use heuristics to limit the number of instructions in the issue queue, as in [1]. They decrease the size of the queue when the heuristic determines potential energy savings. Buyuktosunoglu et al. [8] use fetch gating to control the number of instructions entering the issue queue and combine this with heuristics to limit the issue queue too. However, for both these schemes, limiting the issue queue comes at the price of a non-negligible performance loss.

There have been proposals for an issue queue without wakeups which works by tracking the dependences between instructions [9]. Huang et al. [10] use direct-mapped structures to track dependences and allow more than one consumer per result by adding extra bits. Önder and Gupta [11] implement many consumers to one producer by linking consumers together. Canal and González [9] allow set-associativity in their dependence structure for the same goal. FIFO queues are used by Palacharla et al. [12] into which instructions are dispatched in dependence chains. This means only the oldest instruction in each queue needs to be monitored for potential issue. Abella and González [13] extend this technique so that floating-point queues do not cause a high performance loss. Other schemes have also been recently proposed [14, 15].

The majority of compiler directed approaches to energy reduction have focused on embedded processors. VLIW instruction scheduling has been studied [16–19] whereas others have considered dynamic voltage scaling techniques [20] and the use of compiler controlled caches for frequently executed code [21]. For

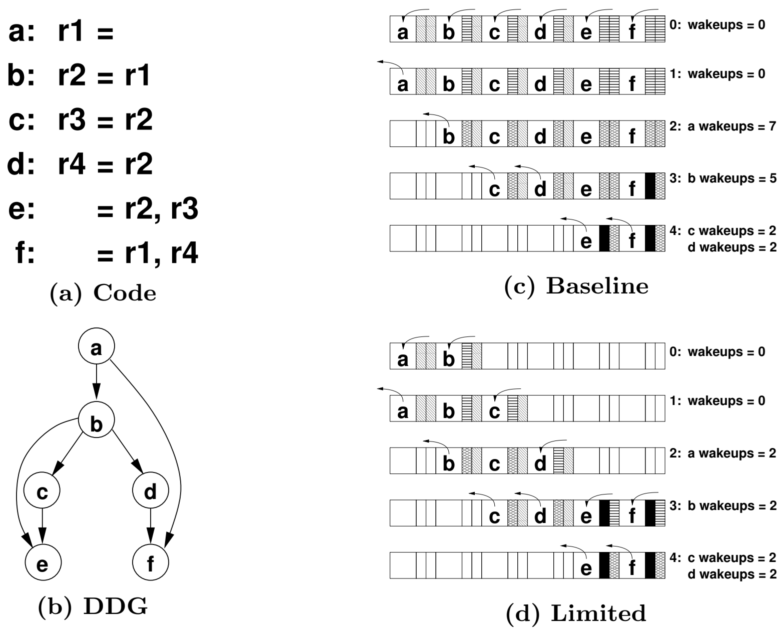


Fig. 1. Issue queue energy savings. 1(a) shows a basic block and 1(b) shows its DDG. In 1(c) it takes 5 cycles and causes 16 wakeups. Limiting the queue to 2 entries in 1(d) means it still takes 5 cycles but only causes 8 wakeups.

superscalar processors, most contributions have considered dynamic voltage scaling techniques [20, 22]. Other schemes have targeted the register file, deallocating registers early for energy savings or performance gains [23, 24]. In [25] a compiler-based technique that performs fine-grained issue queue throttling is presented. This paper performs an extensive exploration of the compiler design space, showing the improvements available through both coarse-grained and fine-grained issue queue limiting.

1.2 Contribution and structure

This paper presents a novel approach to dynamically resizing the issue queue with compiler support. To the best of our knowledge, this is the first paper to develop compiler directed analysis of issue queue size based on critical path analysis. It can be applied to any superscalar organisation and is not tuned to any hardware configuration. The rest of this paper is structured as follows. Section 2 presents an example showing how energy savings can be achieved in the issue queue. Section 3 describes the microarchitecture we use and the small changes we have made. This is followed by section 4 where we outline the compiler analysis performed on different structures in a program. Section 5 briefly describes our experimental setup. Sections 6 and 7 describe two different limiting schemes with their results and they are followed by section 8 which concludes this paper.

2 Motivation

This section describes minimising the issue queue size without affecting the critical path. For the sake of this example, only data dependences that affect the critical path are considered.

Figure 1 shows a basic block where all instructions belong to at least one critical path. To aid readability, instructions are written using pseudo-code. A fragment of assembly code is shown in figure 1(a) and its data dependence graph (DDG) is shown in figure 1(b). There is no need for instructions b, c, d, e and f to be in the issue queue at the same time as a as they are dependent on it and so cannot issue at the same time as it. Likewise, instructions c, d, e and f do not need to be in the issue queue at the same time as b. In fact, instructions e and f do not need to enter the issue queue until c and d leave. Limiting the issue queue to only 2 instructions means the code will execute in the same number of cycles, but fewer wakeups will occur and so energy will be saved.

Figures 1(c) and 1(d) show the issue queue in the baseline and limited cases respectively. A dispatch width of 8 instructions is assumed with each instruction taking one cycle to execute. It is also assumed instruction a has no other input dependences and can therefore issue the cycle after it dispatches. Finally, as in Folegnani and González [1], it is assumed that empty and ready operands do not get woken. Arrows denote whether an instruction is dispatched into the issue queue or issued from it. A white rectangle next to an instruction indicates an empty operand with an empty entry while a rectangle with diagonal lines denotes an operand that is not needed. A rectangle with horizontal lines shows an operand yet to arrive and one that is crossed diagonally shows a wakeup on that operand. Finally, a black rectangle indicates an operand already obtained.

In the baseline case, figure 1(c), all six instructions dispatch in cycle 0. Instruction a issues in cycle 1, completing in cycle 2. It causes seven wakeups and allows b to issue. In cycle 3, b finishes causing five wakeups, allowing instructions c and d to issue. They cause four wakeups in cycle 4 and finally e and f can issue. They write back in cycle 5 and there are sixteen wakeups in total.

Now consider figure 1(d) with the same initial assumptions, but with the constraint that only two instructions can be in the issue queue at any one time. Instruction c must wait unitl cycle 1 to dispatch, d until cycle 2, and instructions e and f must wait until cycle 3. There is no slowdown and only eight wakeups occur, a saving of 50%. In practice the dependence graphs are more complex and resource constraints must be considered, yet this example illustrates the basic principle of the technique.

3 Microarchitecture

This section describes the hardware changes needed to support the limiting of the issue queue. The compiler has to pass information to the processor about the number of issue queue entries needed. Throughout this paper, two methods of accomplishing this are evaluated: special no-ops and instruction tagging.

The special no-ops consist of an opcode and some unused bits, in which the issue queue size is encoded. The special no-ops do nothing to the semantics of the program and are not executed, but are stripped out of the instruction stream in the final decode stage before dispatch. Tagging assumes there are a number of unused bits within each instruction which can be used to encode the issue queue size needed. Instructions are executed as normal, but the information they contain is extracted during decode and used at dispatch. Tagging overcomes the side-effects caused by special no-ops, such increased instruction cache misses.

We have analysed the free bits within the Alpha ISA and found that many instruction have three unused bits in their encoding. One example is the operate format which is used for all register-to-register operations and contains unused function codes. For memory and branch format instructions, we can shorten the displacement field with little impact. Our analysis shows that shortening by 3 bits would affect only 2% of these instructions.

3.1 Issue queue

A multiple-banked issue queue is assumed where instructions are placed in sequential order. We assume that the queue is non-collapsible as in [1, 3, 4]. Having a compaction scheme would cause a significant amount of extra energy to be used each cycle. The queue is similar to [3] where a simple scheme is used to turn off the CAM and RAM arrays at a bank granularity at the same time. The selection logic is always on but it consumes much lower energy than the wakeup logic [12]. Empty and ready entries within the queue are prevented from being woken by gating off the precharge signal to the CAM cells, as proposed by Folegnani and González [1]. The baseline simulator performs no gating and all issue queue banks are permanently on. The schemes presented in this paper limit the number of issue queue entries allowed to contain instructions. The changes required are explained with the approaches in sections 6 and 7.

3.2 Fetch queue

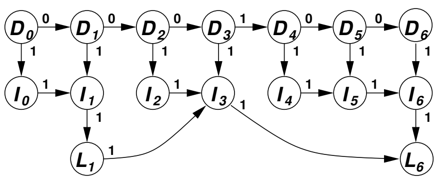
Each cycle the dispatch logic selects a number of instructions to move from the head of the fetch queue to the issue queue. The selection logic has to take into account the dispatch width of the processor, availability of issue queue entries and number of free registers, amongst other criteria. When our schemes use special no-ops to pass information from the compiler to the processor, it is removed from the instruction stream and its value used as the new issue queue limit. Although these instructions are not dispatched, their dispatch slots cannot be used by other instructions. Tagged instructions are not removed from the stream but the limiting value they contain is decoded and used as before.

4 Compiler Analysis

This section describes the compiler analysis performed to determine the number of issue queue entries needed by each program region. It is used in sections 6

```
start:  lda     $2, table
        ldq     $1, ($2)
        sll     $16, 3, $16
        addq    $1, $16, $2
        lda     $3, buffer
        addq    $3, 16, $3
        stq     $2, ($3)
```

(a) Code



(b) Dependence graph

Fig. 2. An example piece of assembly code and its dependence graph. Edges are weighted with the latency (in cycles) taken to resolve the dependence. A D node with latency 0 on its incoming edge can dispatch at the same time as its predecessor, whereas a D node with latency 1 must be dispatched a cycle after its predecessor. In this example, all operations are assumed to take just one cycle.

and 7 for coarse-grained and fine-grained issue queue throttling and is based on simple methods to find the critical path of a program taking into consideration data dependences and resources.

4.1 Program Representation

The compiler considers each procedure individually, first building the control flow graph (CFG) using basic blocks as nodes and the flow of control between them as the edges. Analysis is specialised for loops because instructions from different iterations are executed in parallel when the loop is run. Hence, new CFGs are created (as explained below) and their backward edges removed in the original to preserve the dependences between instructions within each loop and those following. The two program structures created are DAGs and loops.

4.2 Critical Path Model

The critical path is modelled as a dependence graph, similar to those proposed by Tullsen and Calder [26] and Fields et al. [27]. This is because it provides an accurate representation of the events that occur to instructions as they are dispatched and issued. However, we compute our model statically within the compiler, removing the commit nodes and adding extra nodes for events occurring in the load/store queue.

In our model, each instruction, i, is represented by nodes which correspond to events occurring within the processor. There is a D (dispatch) node for when the instruction is dispatched into the issue queue, an I (issue) node for when the instruction is issued from the queue and an L (load/store) node if the instruction is a load or store, which denotes the instruction leaving the load/store queue.

The edges connecting the graph denote dependences between nodes. Each edge is weighted with the minimum number of cycles that the dependence takes to be resolved. Unlike in Fields et al. [27], control dependences between branches

Id	*Constraint*	*Edge*	*Notes*
1	In-order dispatch	$D_p \rightarrow D_i$	If p is immediately follows i
2	IQ issue after dispatch	$D_i \rightarrow I_i$	For every instruction
3	LSQ issue after IQ issue	$I_i \rightarrow L_i$	For every load and store
4	No spec load bypass	$I_p \rightarrow L_i$	If i is a load & p is previous store
5	Data dependence	$I_p \rightarrow I_i$	Non-load p defines source reg of i
6	Data dependence	$I_p \rightarrow L_i$	Non-load p defines data reg of store i
7	Store forwarding	$L_p \rightarrow L_i$	Store p has same address as load i
8	Data dependence	$L_p \rightarrow I_i$	Load p defines source reg of i
9	Data dependence	$L_p \rightarrow L_i$	Load p defines data reg of store i

Table 1. Edges present in the critical path model.

are not modelled because it is assumed that all branches will be predicted correctly. Figure 2 shows an example piece of assembly code and the dependence graph that is formed as the critical path model.

The dependences modelled are shown in table 1. The first edge models in-order instruction dispatch. The second represents instruction issue from the issue queue at least one cycle after dispatch. the third is present for loads and stores, representing issue from the load/store queue at least one cycle after issue from the issue queue. Edge 4 models the constraint that loads cannot speculatively bypass older stores in the load/store queue. Edge 7 models the case where a load accesses the same memory address as a previous store so the data can be forwarded in the load/store queue. Finally, edges 5, 6, 8 and 9 model data dependences via registers between different nodes.

When adding edges to the dependence graph, conservative assumptions are made except in the following cases: all branches are assumed to be predicted correctly; all loads are assumed to hit in the first level data cache; and where a load follows a store and it cannot be determined that they access the same memory address, it is assumed that they do not and that the load can issue before the store once both instructions' addresses have been calculated.

A dependence graph is created for each DAG and loop within the procedure. This graph can be used to calculate the issue queue requirements of the program structure being analysed. Section 4.3 describes the analysis for DAGs and then section 4.4 explains its use for loops.

4.3 Specialised DAG Analysis

Once the dependence graph has been formed each DAG is considered separately. We first describe the specialised analysis and then provide an example of its use.

Analysis Starting with the entry point, we iterate over the DAG's dependence graph to determine the number of issue queue entries needed. We record the set of nodes reached on each iteration in the *issue set*. We traverse the dependence graph along edges from nodes in the issue set to those nodes outside. The edge weights determine the number of iterations to wait after a node has been added to the set before the edge can be traversed and the dependence satisfied.

```
1. $next\_nodes = \{(D_0, 0)\}$
2. While $next\_nodes \neq \emptyset$
   (a) $issued = 0$
   (b) $oldest\_inode =$ oldest $I$ node not in $issue\_set$
   (c) For each functional unit type $T$
       i. $used(T) = 0$
   (d) For each pair $(N, X) \in next\_nodes$
       i. If $X = 0$
          (1) If $issued < issue\_width$ and $used(FU(N)) < number(FU(N))$
              a. Then $issue\_set = issue\_set \cup N$
              b. $used(FU(N)) = used(FU(N)) + 1$
              c. $issued = issued + 1$
              d. $youngest\_inode = Younger(N, youngest\_inode)$
              e. For each edge with weight $W$ connecting $N$ with successor $S$
                 (i) $next\_nodes = next\_nodes \cup (S, W)$
       ii. Else
          (1) $X = X - 1$
   (e) $entries = MAX(entries, Distance(oldest\_inode, youngest\_inode))$

   where $FU(N)$ is the functional unit type required by node $N$
         $Younger(M, N)$ returns the younger of nodes $M, N$
         $Distance(M, N)$ returns the number of entries between nodes $M, N$
```

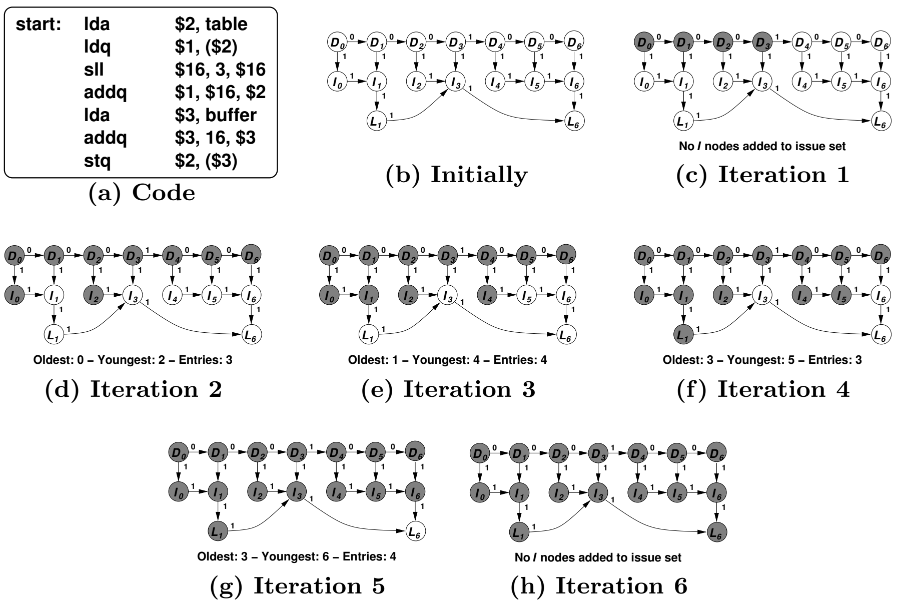
Fig. 3. Algorithm for analysing a DAG.

At this stage of the algorithm we model functional unit contention and a limited processor issue width. To model a finite issue width, we define a maximum number of I and L nodes to be added to the issue set on any given iteration. Functional unit contention is similar, except we define a maximum number of I and L nodes for each type of functional unit.

We repeatedly iterate over the whole graph until all nodes are included in the issue set. The oldest I node not in the issue set at the start of each iteration is recorded, along with the youngest that is added during the iteration. The difference between the two gives the required issue queue size on that iteration to prevent a slowdown of the critical path. The maximum size over all iterations gives the required issue queue size for the whole dependence graph. Figure 3 gives the complete algorithm for DAG analysis.

Example Figure 4 shows an example of a piece of code, its dependence graph and the analysis applied to it. The initial graph is shown in figure 4(b). On the first iteration, shown in figure 4(c) the D nodes are added to the issue set. To indicate this they are coloured grey. However, because the dispatch width allows a maximum of four instructions to enter the pipeline, node D_4 is prevented from being included so the edge $D_3 \rightarrow D_4$ has a weight of 1.

During the second iteration (figure 4(d)), the final three D nodes are added to the issue set along with nodes I_0 and I_2 which have no input dependences. At the start of the iteration I_0 is the oldest I node not in the issue set and the youngest I node to be included is I_2 with a distance of 3 between them. This



```
start:  lda     $2, table
        ldq     $1, ($2)
        sll     $16, 3, $16
        addq    $1, $16, $2
        lda     $3, buffer
        addq    $3, 16, $3
        stq     $2, ($3)
```

(a) Code
(b) Initially
No I nodes added to issue set
(c) Iteration 1
Oldest: 0 – Youngest: 2 – Entries: 3
(d) Iteration 2
Oldest: 1 – Youngest: 4 – Entries: 4
(e) Iteration 3
Oldest: 3 – Youngest: 5 – Entries: 3
(f) Iteration 4
Oldest: 3 – Youngest: 6 – Entries: 4
(g) Iteration 5
No I nodes added to issue set
(h) Iteration 6

Fig. 4. Example of DAG analysis with a dispatch and issue width of 4 and no limit on the number of functional units. With the issue queue limited to four entries, this DAG would not be slowed down at all.

continues until figure 4(h) when all nodes have been added to the issue set. Over all iterations the maximum number of entries needed in this example is just four which would allow the DAG to issue without slowing down the critical path.

4.4 Specialised Loop Analysis

Out-of order execution of loops allows instructions from different loop iterations to be executed in parallel leading to a pipeline parallel execution of the loop as a whole. The analysis, therefore, has to be adjusted accordingly.

Analysis Cycles containing I and L nodes in the dependence graph are detected and that with the longest latency chosen. This set of nodes is called the *cyclic dependence set of nodes (CDS).* The CDS dictates the length of time each loop iteration takes to be completely issued and the next started and it is this set of instructions that is the critical path through the loop.

Equations are formed for each I and L node in the loop based on the relationships within the dependence graph. The equations express the minimum number of cycles a node must wait to issue after a dependent node has issued. By substitution, these equations can be manipulated to express each I and L node in terms of a node within the CDS, meaning that relationships between CDS nodes and others within the graph are exposed. From these new equations it is possible to determine the nodes (possibly on different loop iterations) that could issue together when the loop is actually executed. The required issue queue size is calculated from the largest distance between any two I nodes issuing together.

```
1. $CDS = Find\_CDS(nodes)$
2. For each node $N$ in $nodes$
   (a) For each immediate predecessor node $P$ in $nodes$
       i. Form equation $N_0 = P_i + Latency(P_i)$
3. While there's a change in the equations and an equation not related to a CDS node
   (a) For each node $N$ in $nodes$
       i. For each equation $E$ of the form $N_i = R_j + X$
          (1) If $\exists$ equation $R_k = C_l + Y$ where $C \in CDS$
              a. Rewrite $E$ as $N_i = C_{l+j-k} + X + Y$
4. For each node $N$ in $CDS$
   (a) For each equation of the form $L_i = N_j$
       i. $oldest\_inode = Older(L_i, N_j, oldest\_inode)$
       ii. $youngest\_inode = Younger(L_i, N_j, youngest\_inode)$
   (b) $entries = MAX(entries, Distance(oldest\_inode, youngest\_inode))$

   where $Find\_CDS(graph)$ returns the cycle with the highest weight in $graph$
         $Latency(N)$ returns the latency of $N$
         $Older(M, N, O)$ returns the elder of nodes $M, N, O$
         $Younger(M, N, O)$ returns the younger of nodes $M, N, O$
         $Distance(M, N)$ returns the number of entries between nodes $M, N$
```

Fig. 5. Algorithm for analysing a loop.

Figure 5 summarises the algorithm for analysing a loop. The algorithm is guaranteed to terminate due to the condition in step 3. We do not alter the equations to model a finite issue width or functional unit contention because nodes can execute differently on each loop iteration. Instead, we factor in contention when calculating the required queue size. For example, if the maximum issue width is eight but nine instructions wish to issue then the calculated issue queue size is multiplied by 8/9.

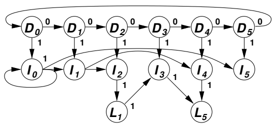
Example Figure 6 shows an example of the compiler analysis for loops. The dependence graph for the loop is shown in figure 6(b). In this graph it is easy to see that there is only one candidate cycle for the CDS, containing node I_0 with latency 1. Figure 6(c) shows the initial equations formed for the I and L nodes in the loop. Each equation relates the node on loop iteration i with one of its dependence nodes. Where a node has several dependences then equations are usually maintained for each, however only one is shown here for simplicity.

The equation for node I_0 in figure 6(c) refers to a previous instance of itself, the instance of the node on the previous iteration of the loop. The equation for I_0 means that in iteration i, I_0 can issue one cycle after I_0 from the previous iteration $(i-1)$. Continuing on, I_1 can issue 1 cycle after I_0 from the current iteration, I_2 can issue 1 cycle after I_1 from current iteration, and so on.

Once the equations have been formed they are manipulated to remove constants where possible and, hence, determine which nodes issue at the same time, producing the equations shown in figure 6(d). Considering only the I nodes, it is now trivial to compute the issue queue size needed by the loop. In order that I_0 on iteration $i+4$ can issue at the same time as I_3 on iteration i, they must be in

```
loop:  subl      $17, 1, $17
       s4addq    $17, $1, $2
       ldq       $3, ($2)
       addq      $3, 4, $3
       stq       $3, ($2)
       bne       $17, loop
```

(a) Code



(b) Dependence graph

$$\begin{aligned}
I_{0,i} &= I_{0,i-1} + 1\\
I_{1,i} &= I_{0,i} + 1\\
I_{2,i} &= I_{1,i} + 1\\
L_{2,i} &= I_{2,i} + 1\\
I_{3,i} &= L_{2,i} + 1\\
I_{4,i} &= I_{0,i} + 1\\
L_{4,i} &= I_{3,i} + 1\\
I_{5,i} &= I_{0,i} + 1
\end{aligned}$$

(c) Initially

$$\begin{aligned}
I_{0,i} &= I_{0,i-1} + 1\\
I_{1,i} &= I_{0,i+1}\\
I_{2,i} &= I_{0,i+2}\\
L_{2,i} &= I_{0,i+3}\\
I_{3,i} &= I_{0,i+4}\\
I_{4,i} &= I_{0,i+1}\\
L_{4,i} &= I_{0,i+5}\\
I_{5,i} &= I_{0,i+1}
\end{aligned}$$

(d) Finally

Fig. 6. Example of loop analysis with equations formed for the dependence graph shown. With a dispatch width of 8 and the issue queue limited to 22 entries, the critical path would not be affected.

the issue queue at the same time. This would require 22 entries to be available, allowing space for instructions corresponding to I_3, I_4 and I_5 from iteration i, 18 I nodes from iterations $i+1$, $i+2$ and $i+3$ (6 each), and I_0 from loop iteration $i+4$. Providing this many entries would allow parallel execution of this loop without affecting the critical path.

4.5 Interprocedural Analysis

One problem with our current approach is that dependence across procedure boundaries are not considered due to the limitations of our compilation infrastructure. To address this, in section 7.3, we investigate a technique that uses a small amount of extra hand-coded analysis to include these interprocedure dependences. The scheme works by first finding the call sites for each program procedure (which will be at the end of a DAG). Then, a resource list is produced for each site which gives the functional unit usage and issue set for each iteration over the DAG. In a second step, the resource lists from all possible call sites are used as initialisation at each procedure start and the analysis updated.

As an example, consider the DAG *I1* $\rightarrow$ *I2*, where *I2* is a function call. The issue set on iteration 1 will be $\{I1\}$ and on iteration 2 it will be $\{I2\}$. Furthermore, consider IALU is busy until iteration 3 and FPALU until iteration 4. This is the first stage of our approach. In the second step, we start analysing the first DAG in the called function. Assuming that this will start in iteration 3, we now know that we cannot schedule anything on FPALU until the following iteration, so we add this constraint into our analysis.

4.6 Summary

Having performed the compiler analysis, identifying the critical path and determining the issue queue requirements that would not slow it down, the information can be communicated to the processor to perform issue queue throttling.

Component	*Configuration*
Pipeline	8 instructions wide; 128 entry reorder buffer; 80 entry issue queue (10 banks of 8); 112 integer and FP registers
Branch predictor	Hybrid of 2K gshare and 2K bimodal with a 1K selector; BTB with 2048 entries, 4-way
Caches	64KB, 2-way, 32B line, 1 cycle hit L1 Insn; 64KB, 4-way, 32B line, 2 cycles hit L1 Data; 512KB, 8-way, 64B line, 10/50 cycles hit/miss L2
Functional units	6 ALU (1 cycle), 3 Mul (3 cycles) Integer; 4 ALU (2 cycles), 2 MultDiv (4/12 cycles) FP

Table 2. Processor configuration.

Section 6 describes a coarse-grained approach for each DAG or loop. Section 7 then presents a fine-grained scheme for each basic block. First, however, we describe our experimental setup.

5 Experimental Setup

This section describes the compiler, simulator and benchmarks used to evaluate our issue queue throttling schemes. Our processor configuration is shown in table 2 which was implemented in Wattch [28], based on SimpleScalar [29] using the Alpha ISA. We modelled a 70nm technology using Wattch's aggressive conditional clock gating scheme (cc3) which assumes idle resources consume 15% of their full energy. We used the MachineSUIF compiler from Harvard [30] to compile the benchmarks, which is based on the SUIF2 compiler from Stanford [31].

We chose to use the SPEC CPU2000 integer benchmark suite [32] to evaluate our schemes. However, we did not use *eon* because it is written in C++ which SUIF cannot directly compile. Similarly, we did not use any of the floating point benchmarks. Most of them cannot be directly compiled by SUIF because they are written in Fortran 90 or contain language extensions. We ran the benchmarks with the *ref* inputs for 100 million instructions after skipping the initialisation part and warming the caches and branch predictor for 100 million instructions.

Throughout this paper we evaluate our schemes in terms of performance and energy savings. For a performance metric we used instructions per cycle (IPC). For energy savings we considered dynamic (i.e. transistor switching activity) and static (i.e. the leakage energy consumed when the transistors are turned on).

6 Coarse-Grained Throttling

This section describes the first use of the analysis presented in section 4 to limit the size of the issue queue for each DAG or loop in its entirety. This is conveyed to the processor through the use of a special no-op or tag at the start of each DAG or loop. In addition a special no-op is also placed after a loop to reset the maximum queue size to the value of the surrounding DAG, allowing the queue to be fit to each structure's requirements.

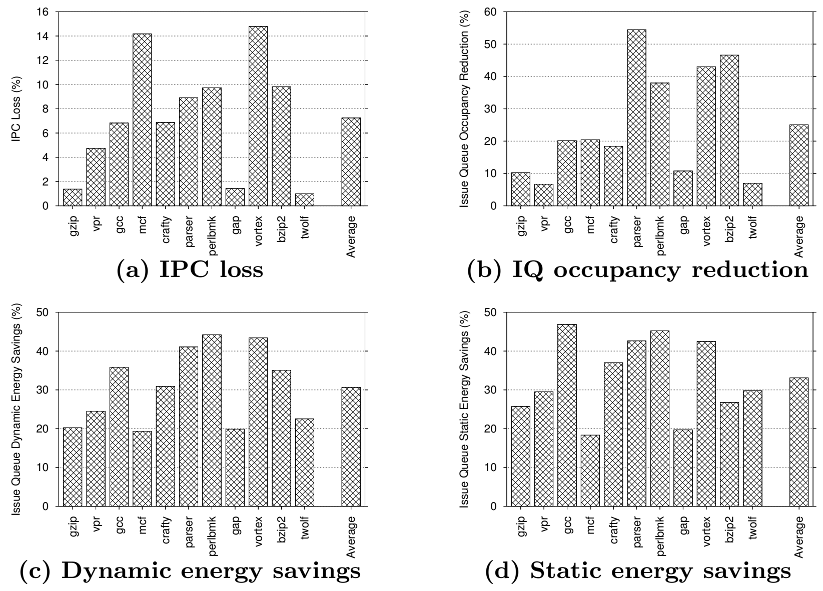


Fig. 7. Performance, issue queue occupancy reductions and issue queue energy savings when limiting using DL no-ops for coarse-grained issue queue throttling.

Section 6.1 next describes the trivial microarchitectural changes to the issue queue, then section 6.2 presents the results from using special no-ops to pass the compiler-inferred queue requirements to the processor. Section 6.3 then evaluates instruction tagging, instead of using the special no-ops.

6.1 Issue Queue

The changes to the issue queue are very minor. The limiting no-op or tag simply indicates the maximum number of entries, both full or empty, that are allowed between the *head* and *tail* pointers in the issue queue. Instructions cannot dispatch if the *tail* pointer would become further than this from the *head*. In certain situations a limiting no-op or tag will be encountered that contains a smaller number of entries than is already allowed. To rectify this, enough instructions must be issued from the *head* for the distance between the two pointers to become less than the new maximum distance before dispatch can start again.

6.2 DL No-ops

The first evaluation of this scheme was performed using special no-ops to communicate the limiting information. We called these *DL no-ops* because of the granularity at which they are placed: at the start of every DAG and loop.

Figure 7(a) shows the effect on the performance of each benchmark in terms of IPC loss. Some benchmarks are badly affected, such as *mcf* and *vortex* which lose over 14% of their performance. Others experience only a small loss, such as

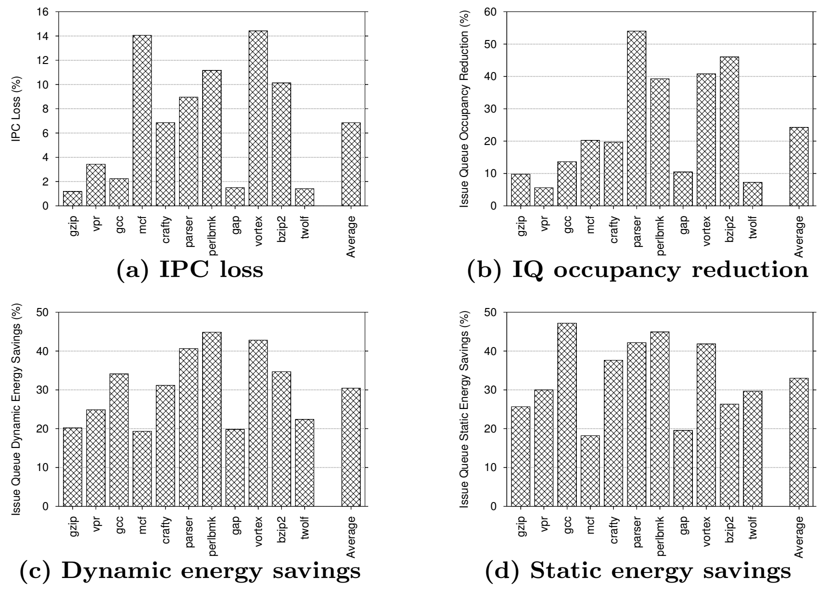


Fig. 8. Performance, issue queue occupancy reductions and issue queue energy savings when limiting using tags for coarse-grained issue queue throttling.

twolf which has a 1% drop. On average, the performance loss is 7.2% due to a reduction in the dispatch width every time a special no-op is encountered, along with an inability to alter the size of the queue at a fine-enough granularity.

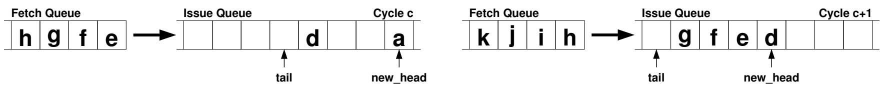
The issue queue occupancy reduction for this scheme is shown in figure 7(b). Although benchmarks that experienced a small performance loss, such as *gzip*, *gap* and *twolf*, also experience a small occupancy reduction, the benchmark that benefits the most is *parser* with a 54% drop. This leads to an average occupancy reduction of 25%. The average issue queue dynamic and static energy savings achieved through this are 31% and 33% respectively (figures 7(c) and 7(d)).

In addition to this, the presence of the DL no-ops in the binary increases the code size and means that more instructions need to be cached and fetched. On the other hand, our scheme throttles the issue queue, reducing the number of mis-speculated instructions that are actually fetched from the cache. The net result is an average reduction of 5% in the number of instruction cache accesses and a 20% increase in the size of each binary, on average.

6.3 Tags

As discussed in section 3.2, DL no-ops take up valuable dispatch resources. To reduce this problem we now consider a scheme where the first instruction in a DAG or loop is tagged with the resizing information, assuming that there were enough redundant bits in the ISA to accommodate the values needed.

The performance loss for each benchmark using these tags is shown in figure 8(a). Most benchmarks benefit to some degree from the removal of the no-



(a) Issue queue on cycle c **(b) Issue queue on cycle c+1**

Fig. 9. Operation of *new_head* pointer with a limit of four entries.

ops, *gcc* especially which loses only 2.2% performance with tags compared with 6.8% with no-ops. However, in badly performing benchmarks such as *vortex* and *bzip2*, removal of the no-ops makes little difference. When considering the effects on issue queue occupancy, shown in figure 8(b), dynamic energy (figure 8(c)) and static energy savings (figure 8(d)) there is also little change when using tagging. This shows that the no-ops have little impact on the behaviour of the issue queue but can have a major impact on performance.

In summary, the two schemes presented in this section that perform coarse-grained analysis of a whole DAG's issue queue requirements can be used to reduce the energy consumption of the queue. However, they incur a non-negligible performance loss. The following section attempts to reduce this by performing the throttling at a much finer granularity.

7 Fine-Grained Throttling

It is clear from section 6 that performing issue queue limiting over a whole DAG is too restrictive and creates significant performance losses for some benchmarks. However, the performance losses are also partly due to the fact that the throttling takes place over the whole issue queue. When the first instructions in a DAG enter the queue, instructions from a previous DAG will already be there. Without knowing about dependences in the previous DAG, the compiler analysis cannot take into account the issue queue requirements of the older instructions. Any assumptions that underestimate these requirements mean that the older instructions are in the queue longer than the compiler realises and stall the dispatch of the new DAG. Hence, this section restricts the throttling of the issue queue to only the youngest part, allowing older instructions to issue without the compiler needing to know about them and consider them during its analysis.

This section is structured as follows. Section 7.1 describes the changes to the issue queue required so that only the youngest part of the queue is throttled. Section 7.2 presents results obtained using special no-ops to pass the queue size required by the youngest part of the queue. Section 7.3 then presents the same scheme using instruction tags instead of special no-ops. The final approach is also compared to a state-of-the-art hardware scheme.

7.1 Issue Queue

The issue queue requires only minor modifications to allow the new limiting scheme to work. A second *head* pointer, named *new_head*, is introduced which allows compiler control over the youngest entries in the queue. The *new_head*

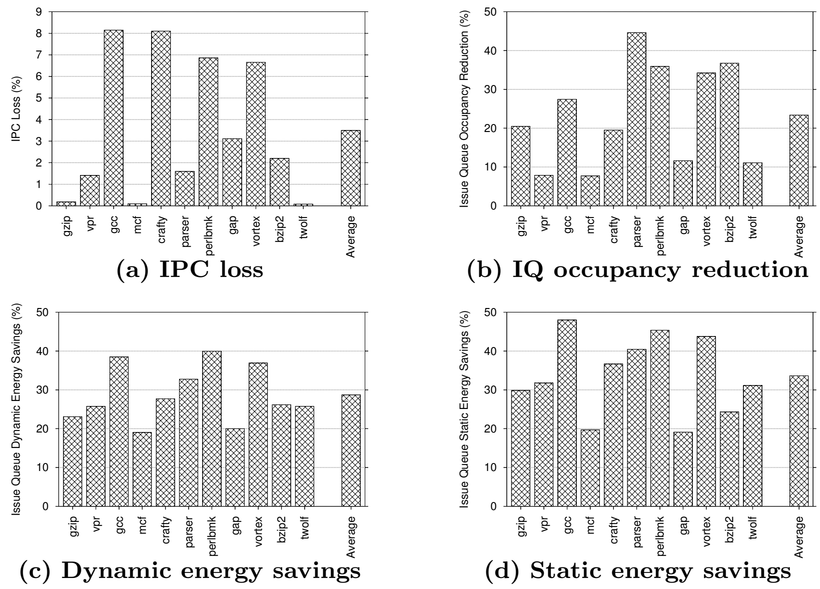


Fig. 10. Performance, issue queue occupancy reductions and issue queue energy savings when limiting using block no-ops for fine-grained issue queue throttling.

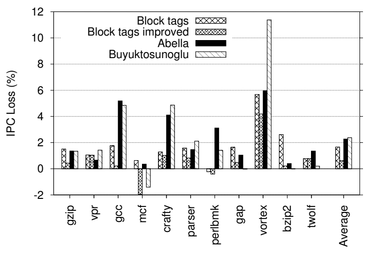
pointer points to a filled entry between the *head* and *tail* pointers. It functions exactly the same as the *head* pointer such that when the instruction it points to is issued it moves towards the *tail* until it reaches a non-empty slot, or becomes the *tail*. New instructions being dispatched are still added to the queue's *tail*.

This scheme is based on the fact that it is relatively easy to determine the future additional requirements of a small program region. Where, in the previous approach, the maximum number of entries between *head* and *tail* were limited, in this section the distance between *new_head* and *tail* is restricted. This is so that instructions from previous basic blocks (and previous DAGs) can be present in the queue without affecting the limiting of the youngest part. So, the instructions between *new_head* and *tail* are from the youngest program region whereas the rest of the instructions (between *head* and *new_head*) are from older ones.

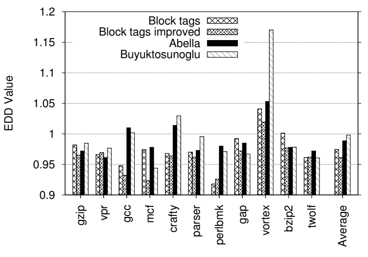
The operation of the queue is demonstrated in figure 9. If instruction a issues, the *new_head* pointer moves up to the next non-empty instruction, so three slots to d. This means that up to three more instructions can be dispatched to keep the number of entries at four or fewer. So, e, f and g can now dispatch as shown in figure 9(b).

7.2 Block No-ops

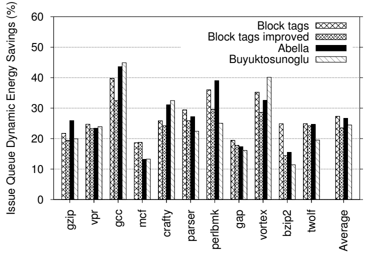
This section evaluates the new limiting scheme using special no-ops, called block no-ops, inserted into the code. The performance of each benchmark is shown in figure 10(a). In this approach benchmarks are either hardly affected (*gzip*, *mcf*



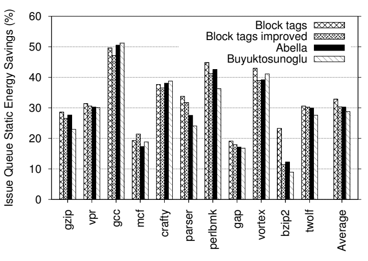
(a) IPC loss



(b) EDD product value



(c) Dynamic energy savings



(d) Static energy savings

Fig. 11. Performance reduction, EDD product values and issue queue energy savings when limiting using tags for fine-grained issue queue throttling.

and *twolf* which lose less than 0.2%) or experience large performance losses (e.g. *crafty* at over 8%). On average, however, the loss is only 3.5%.

The issue queue occupancy reduction for this scheme is shown in figure 10(b). Most benchmarks experience around a 20% reduction and *parser* gets an occupancy reduction of 45%, the average being a 23% reduction. This gets convered into dynamic and static energy savings which are shown in figures 10(c) and 10(d). The average dynamic and static energy savings are 29% and 34% respectively. The best savings come from *gcc* which experiences 38% dynamic and 48% static energy savings. All benchmarks achieve 19% dynamic energy savings whilst most see their static energy reduced by at least 30% too.

With this approach there is also an increase in code size of 44% on average. This also increases the number of instructions that need to be fetched but, as described in section 6.2, this is offset by throttling the issue queue. Overall, there is an increase of 1% in the number of instruction cache accesses on average.

7.3 Tags

This section evaluates the use of tags instead of no-ops to convey the limiting information for the youngest part of the issue queue. In each graph three schemes are shown. The first is the approach that simply uses tags instead of no-ops to pass limiting information to the processor. This scheme is called *Block tags*.

The second scheme, called *Block tags improved*, is derived from the *Block tags* technique. By hand, limited extra analysis was applied to all benchmarks to reduce functional unit contention and prevent dispatch stalls causing under-utilisation of resources when useful work could be performed. It is described in more detail in section 4.5. This interprocedural analysis would typically be available in a mature industrial compiler but is absent in our SUIF prototype.

For comparison, we implemented two state-of-the-art hardware approaches. The first is from papers published by Abella and González [4, 33], the second from Buyuktosunoglu et al. [8]. We implemented these within the same simulation infrastructure as all other experiments. We compare to the *IqRob64* scheme from [4] as this gave the most energy savings and is henceforth referred to as *Abella*. We call the technique from [8] *Buyuktosunoglu*.

Results As can be seen from figure 11(a), the compiler schemes lose less performance than *Abella* and *Buyuktosunoglu*, *perlbmk* and *mcf* even gain slightly (0.4% and 1.9% respectively). This is due, in part, to a reduced number of branch mispredictions, but also because young instructions on the critical path are sometimes prevented from executing due to functional unit contention with older, non-critical instructions. With the issue queue throttling scheme here, these older instructions are executed on a different cycle and thus the contention does not occur. All approaches perform badly on *vortex* (especially *Buyuktosunoglu*) but the two hardware schemes cannot cope well with *gcc*, *crafty* or *perlbmk*. The extra analysis performed in *Block tags improved* considerably reduces the performance loss of most benchmarks. On average *Block tags* loses 1.7%, *Block tags improved* 0.6%, *Abella* 2.3% and *Buyuktosunoglu* 2.4% performance.

The energy-delay-squared product for each of the schemes is shown in figure 11(b), where it is assumed that the issue queue consumes 20% of the total processor energy, consistent with the findings of Folegnani and González [1]. We have assumed that leakage accounts for 25% of total energy as described in [34]. The compiler-directed schemes have better EDD products than *Abella* and *Buyuktosunoglu*, with the best, *Block tags improved*, achieving 0.96, compared to 0.99 for both hardware approaches.

These EDD values are achieved through savings in both dynamic and static energy (figures 11(c) and 11(d)). The static energy of the issue queue is completely dependent on the number of banks that are on each cycle whereas the dynamic energy consumption is dependent on the number of instructions waking others and reads and writes to the queue, as well as the occupancy. The average dynamic energy savings of the *Block tags* scheme is the same as *Abella* (27%) whereas it is reduced slightly to 24% in *Block tags improved* and *Buyuktosunoglu*. The static energy reduction is, on average, better than the hardware approaches in both our schemes. The *Block tags* approach reduces it by 33%, *Block tags improved* by 30%, *Abella* by 30% and *Buyuktosunoglu* by 29%.

8 Conclusions

This paper has presented novel techniques to dynamically resize the issue queue using the compiler for support. The compiler analyses and determines the number of issue queue entries needed by each program region and encodes this number in a special no-op or a tag with the instruction. The number is extracted at dispatch and used to limit the number of instructions in the queue. This reduces the issue queue occupancy and thus the amount of energy consumed.

Results from the implementation and evaluation of the proposed schemes show 31% dynamic energy savings with a 7.2% average IPC loss for a basic scheme which attempts to determine the whole queue size needed. By only determining the requirements of the youngest part of the queue, the performance loss can be reduced to just 3.5% when using special no-ops to convey the information. Tagging instructions and using improved analysis reduces this further to just 0.6%, compared with 2.3% and 2.4% for two state-of-the-art hardware schemes [4, 8]. Both compiler and hardware schemes save similar amounts of static and dynamic energy in the issue queue.

Future Work One of the downsides of our current approach is that it only works on single-threaded processors. On a simultaneous multithreaded (SMT) architecture, the schemes presented in this paper that throttle the issue queue considering only one thread at a time could be detrimental to other processes sharing the resources. Future work will consider issue queue limiting in this type of environment, using compiler analysis of future program requirements and hardware knowledge of the current system state to adapt the issue queue size for the benefit of all executing threads.

Acknowledgements This work has been partially supported by the Royal Academy of Engineering, EPSRC and the Spanish Ministry of Science and Innovation under grant TIN2007-61763.

References

1. Folegnani, D., González, A.: Energy-effective issue logic. In: ISCA-28. (2001)
2. Emer, J.: Ev8: The post-ultimate alpha. In: Keynote at PACT. (2001)
3. Buyuktosunoglu, A., Schuster, S., Brooks, D., Bose, P., Cook, P., Albonesi, D.: An adaptive issue queue for reduced power at high performance. In: PACS. Volume 2008 of LNCS. (2000)
4. Abella, J., González, A.: Power-aware adaptive issue queue and rename buffers. In: HiPC, LNCS2913. (2003)
5. Bahar, R.I., Manne, S.: Power and energy reduction via pipeline balancing. In: ISCA-28. (2001)
6. Maro, R., Bai, Y., Bahar, R.I.: Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In: PACS. Volume 2008 of Lecture Notes in Computer Science., Springer (2000)
7. Manne, S., Klauser, A., Grunwald, D.: Pipeline gating: Speculation control for energy reduction. In: ISCA-25. (1998)
8. Buyuktosunoglu, A., Karkhanis, T., Albonesi, D.H., Bose, P.: Energy efficient co-adaptive instruction fetch and issue. In: PACS. Volume 2008 of LNCS. (2000)
9. Canal, R., González, A.: Reducing the complexity of the issue logic. In: ICS-15. (2001)
10. Huang, M., Renau, J., Torrellas, J.: Energy-efficient hybrid wakeup logic. In: ISLPED. (2002)

11. Önder, S., Gupta, R.: Superscalar execution with dynamic data forwarding. In: PACT. (1998)
12. Palacharla, S., Jouppi, N.P., Smith, J.E.: Complexity-effective superscalar processors. In: ISCA-24. (1997)
13. Abella, J., González, A.: Low-complexity distributed issue queue. In: HPCA-10. (2004)
14. Ernst, D., Hamel, A., Austin, T.: Cyclone: A broadcast-free dynamic instruction scheduler with selective replay. In: ISCA-30. (2003)
15. Hu, J.S., Vijaykrishnan, N., Irwin, M.J.: Exploring wakeup-free instruction scheduling. In: HPCA-10. (2004)
16. Lee, C., Lee, J.K., Hwang, T., Tsai, S.C.: Compiler optimization on instruction scheduling for low power. In: ISSS-13. (2000)
17. Lorenz, M., Leupers, R., Marwedel, P., Dräger, T., Fettweis, G.: Low-energy DSP code generation using a genetic algorithm. In: ICCD-19. (2001)
18. Zhang, W., Vijaykrishnan, N., Kandemir, M., Irwin, M.J., Duarte, D., Tsai, Y.F.: Exploiting VLIW schedule slacks for dynamic and leakage energy reduction. In: MICRO-34. (2001)
19. Toburen, M.C., Conte, T.M., Reilly, M.: Instruction scheduling for low power dissipation in high performance microprocessors. Technical report, North Carolina State University (1998)
20. Magklis, G., Scott, M.L., Semeraro, G., Albonesi, D.H., Dropsho, S.: Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In: ISCA-30. (2003)
21. Bellas, N., Hajj, I., Polychronopoulos, C., Stamoulis, G.: Energy and performance improvements in microprocessor design using a loop cache. In: ICCD-17. (1999)
22. Hsu, C.H., Kremer, U., Hsiao, M.: Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In: ISLPED. (2001)
23. Jones, T.M., O'Boyle, M.F.P., Abella, J., González, A., Ergin, O.: Compiler directed early register release. In: PACT. (2005)
24. Lo, J.L., et al.: Software-directed register deallocation for simultaneous multithreaded processors. IEEE TPDS **10**(9) (1999)
25. Jones, T.M., O'Boyle, M.F.P., Abella, J., González, A.: Software directed issue queue power reduction. In: HPCA-11. (2005)
26. Tullsen, D.M., Calder, B.: Computing along the critical path. Technical report, University of California, San Diego (1998)
27. Fields, B., Rubin, S., Bodík, R.: Focusing processor policies via critical-path prediction. In: ISCA-28. (2001)
28. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: A framework for architectural-level power analysis and optimizations. In: ISCA-27. (2000)
29. Burger, D., Austin, T.: The simplescalar tool set, version 2.0. Technical Report TR1342, University of Wisconsin-Madison (1997)
30. Smith, M.D., Holloway, G.: The Machine-SUIF documentation set. http://-www.eecs.harvard.edu/machsuif/software/software.html (2000)
31. The Stanford SUIF Compiler Group: The suif compiler infrastructure. http://suif.stanford.edu/
32. The Standard Performance Evaluation Corporation (SPEC): CPU 2000. http://www.spec.org/cpu2000/
33. Abella, J., González, A.: Power-aware adaptive instruction queue and rename buffers. Technical Report UPC-DAC-2002-31, UPC (2002)
34. Aygün, K., Hill, M.J., Eilert, K., Radhakrishnan, K., Levin, A.: Power delivery for high-performance microprocessors. Intel Technology Journal **9**(4) (2005)