Software Directed Issue Queue Power Reduction

Timothy M. Jones, Michael F.P. O'Boyle
Member of HiPEAC, School of Informatics
University of Edinburgh
t.m.jones@sms.ed.ac.uk, mob@inf.ed.ac.uk

Jaume Abella† and Antonio González†‡
†Dept. Computer Architecture, ‡Intel Labs
Universitat Politècnica de Catalunya
{jabella, antonio}@ac.upc.es

Abstract

The issue logic of a superscalar processor dissipates a large amount of static and dynamic power. Furthermore, its power density makes it a hot-spot requiring expensive cooling systems and additional packaging.

In this paper we present a novel software assisted approach to power reduction where the processor dynamically resizes the issue queue based on compiler analysis. The compiler passes information to the processor about the number of entries needed which limits the number of instructions dispatched and resident in the queue. This saves power without adversely affecting performance.

Compared with recently proposed hardware techniques, our approach is faster, simpler and saves more power. Using a simplistic scheme we achieve 47% dynamic and 31% static power savings in the issue queue with only a 2.2% performance loss. We then show that the performance loss can be reduced to less than 1.3% with 45% dynamic and 30% static power savings, outperforming all current approaches.

1. Introduction

Superscalar processors contain complex control logic in order to extract sufficient instruction level parallelism (ILP). Unfortunately, those structures used to perform out-of-order execution consume a large amount of power. This has important implications for future processor technology.

The issue logic is one of the main sources of power dissipation in current superscalar processors [11, 10]. It has been estimated that up to 25% of the energy consumed by a processor is in the issue logic [13]. Furthermore, the issue 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.

There has been much work in developing hardware techniques to reduce the power cost of the issue logic by turning off unused entries and adapting the issue queue size to the available ILP [13, 8, 2]. Unfortunately, there is inevitably a delay in sensing rapid phase changes and adjusting accordingly. 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 power.

Correctly sizing the issue queue reduces the number of non-critical in-flight instructions and so reduces the number of entries needed in the register file. This enables further power savings by simply turning off empty banks. We evaluate the power savings for both the issue queue and the register file in section 5.

1.1. Related work

Saving power by turning off unused parts of the processor has been the focus of much previous work. Bahar and Manne [4] introduce *pipeline balancing* which changes the issue width of the processor depending on the issue IPC over a fixed window size. Other papers [21, 20] propose shutting down parts of the processor in a similar manner with similar results.

Considering the issue queue (IQ) alone, Folegnani and González [13] 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.* [8] propose a similar resizing scheme, using banks which can be turned off for static power savings. Abella and González [2], like Folegnani and González, use heuristics to limit the number of instructions in the IQ. They decrease the size of the queue when the heuristic determines

Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture (HPCA-11 2005)
1530-0897/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

potential power savings. However, this at the price of a non-negligible performance loss.

There have been proposals for an IQ without wakeups which works by tracking the dependences between instructions [9]. Huang *et al.* [16] use direct-mapped structures to track dependences and allow more than one consumer per result by adding extra bits. Önder and Gupta [22] 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. Palacharla *et al.* [23] use FIFO queues to 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 [3] extend this technique so that floating-point queues do not cause a high performance loss. Other schemes have also been recently proposed [12, 15].

The majority of compiler directed approaches to power reduction have focused on embedded processors. Lee *et al.* [17] and Lorenz *et al.* [18] and Zhang *et al.* [25] all consider VLIW instruction scheduling. Other research has considered dynamic voltage scaling techniques [14, 19] and the use of compiler controlled caches for frequently executed code [5].

There are relatively fewer papers on the subject of compiler optimisations for superscalar power reduction. One notable contribution is by Toburen *et al.* [24] in which the authors present a new instruction scheduling algorithm. They associate an energy dissipation with each functional unit and only allow a certain amount to be dissipated every cycle.

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 power savings can be achieved in the IQ. 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 describes the results and is followed by section 6 which concludes this work.

2. Example

We wish to minimise the IQ size without affecting the critical path. The critical path of a program depends on the data dependence structure of a program but is also affected by the limited number of functional units and variable memory latency due to cache misses, amongst other things. For the sake of this example, we just consider data dependences that affect the critical path. Consider figure 1 which shows a basic block where all instructions belong to at least one critical path. Figure 1(a) shows the code and figure 1(b) shows the data dependence graph (DDG). There is no need for instructions `c`, `d`, `e` and `f` to be in the IQ at the same time as `a` and `b` as they are dependent on them and so cannot issue at the same time as them. Likewise, instructions `e` and `f` do not need to be in the IQ at the same time as `c` and `d`. If we limit the IQ to only 2 instructions then the code will execute in the same number of cycles, but fewer wakeups will occur and so power will be saved.

Figures 1(c) and 1(d) show the IQ in the baseline and limited cases respectively. We assume a dispatch width of 8 instructions and each instruction takes one cycle to execute. We also assume that registers `r1` and `r2` are already defined so instructions `a` and `b` can issue on the cycle after they dispatch. Finally, we also assume, as in Folegnani and González [13], that empty and ready operands do not get woken. Arrows denote whether an instruction is dispatched

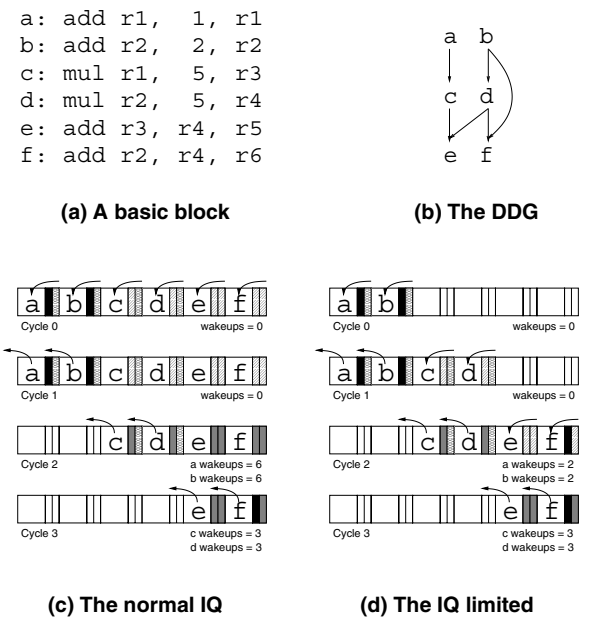


Figure 1. Issue queue power savings. 1(a) shows a basic block and 1(b) shows its DDG. In 1(c) it takes 4 cycles and causes 18 wakeups. Limiting the queue to 2 entries in 1(d) means the block still takes 4 cycles but only causes 10 wakeups.

1530-0897/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

into the IQ or issued from it. A white rectangle next to an instruction indicates an empty operand with an empty entry while a crossed diagonal rectangle shows that an operand is not needed. A rectangle with diagonal lines shows an operand yet to arrive and a grey one shows a wakeup on that operand. Finally, a black rectangle indicates an operand already obtained.

In the baseline case, figure 1(c), `a` and `b` issue on cycle 1. On cycle 2 they complete execution and cause 6 wakeups each whilst instructions `c` and `d` issue. On cycle 3, `c` and `d` cause 6 wakeups and allow `e` and `f` to issue. They write back in cycle 4 and there are 18 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 IQ at any one time. On cycle 0, only `a` and `b` are dispatched. They issue on cycle 1 and this makes way for `c` and `d` to dispatch. On cycle 2 instructions `a` and `b` cause 4 wakeups, `c` and `d` issue making way for instructions `e` and `f` to dispatch. On cycle 3, `c` and `d` cause 6 wakeups and the final two instructions issue, writing back in cycle 4. There is no slowdown and only 10 wakeups, a saving of 44%. In practice the dependence graphs are more complex and resource issues must be considered, yet this example illustrates the basic principle of our technique.

3. Microarchitecture

Our processor is an out-of-order superscalar. The compiler has to pass information to it regarding the number of IQ entries to enable. One mechanism we evaluate is based on NOOPs inserted into the code containing the IQ size. These special NOOPs consists of an opcode and some unused bits, in which the IQ size is encoded. The special NOOPs 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.

3.1. Issue queue

The issue queue is the focus of our work and has some minor additions. We introduce a second *head* pointer, named *new_head*, which allows compiler control over the youngest entries in the queue. The *new_head* 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 tail of the queue.

Our technique is based on the fact that it is relatively easy to determine the future additional requirements of a small program region. We define *max_new_range* which is the maximum number of entries between the *new_head* and *tail* pointers. In other words, *max_new_range* is the number of IQ elements needed for the next program region. The remaining instructions (between *head* and *new_head*) are from older program regions.

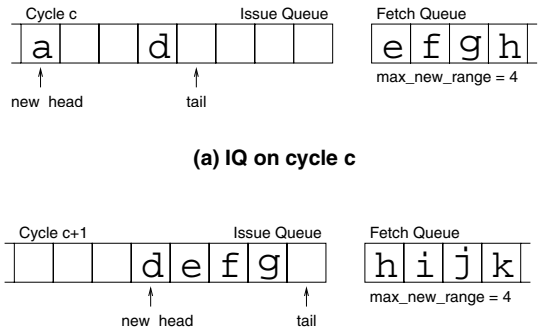


Figure 2. Operation of *new_head* pointer and *max_new_range*

The compiler sets the value of *max_new_range* and its operation, along with that of the *new_head* pointer, is demonstrated in figure 2. If instruction `a` issues, the *new_head* pointer moves up to point 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 2(b).

We assume a multiple-banked issue queue where instructions are placed in sequential order. We assume that the queue is non-collapsible as in [13, 8, 2]. Having a compaction scheme would cause a significant amount of extra energy to be used each cycle. The IQ is similar to [8] 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 wakeup logic [23].

3.2. Fetch queue

Once instructions are fetched they are placed in the fetch queue where they spend several cycles being decoded before being dispatched to the IQ. Each cycle the dispatch logic selects a number of instructions to move from the head of this queue to the issue queue. This is usually the minimum of the dispatch width (8 in our processor) and the number required to keep the distance between the *new_head* and *tail* pointers to no more than *max_new_range* entries.

In our initial implementation, we insert special NOOPs, which contain the *max_new_range* value, into the instruc-

1530-0897/05 $20.00 © 2005 **IEEE**
IEEE COMPUTER SOCIETY

tion stream, When this special NOOP is encountered in the instructions to be dispatched, it is removed and its value used as the new *max_new_range*.

4. Analysis

This section describes our complier analysis which is based on simple methods to find the critical path of a program taking into consideration data dependences and resources. There are three main structures within a program that we analyse: directed acyclic graphs (DAGs), loops and procedure calls. The aim of the compilation pass is to insert special NOOPs into the code which tell the processor the maximum number of IQ entries needed until the next special NOOP. We implemented our pass in MachineSUIF from Harvard [26], an extension to Stanford's SUIF compiler [28].

4.1. Breakdown into groups

MachineSUIF contains analysis libraries to identify the natural loops in a procedure. Where a loop has an inner loop, this is considered separately, so the inner loop's basic blocks form one loop and those that are only in the outer loop form another. This is to avoid analysis of the inner loop's blocks more than once.

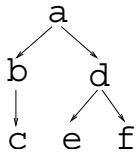
DAGs are formed from the basic blocks in the procedure using control flow analysis. The first block in a DAG is the first block in the procedure, or a block immediately following a function call.

Within each loop and DAG the DDG is constructed and its edges labelled with the latencies of the instructions for use in a more detailed analysis stage.

4.2. DAGs

We have found that fine-grained analysis of DAGs gives the most power savings and hence analyse each basic block individually. This has the added advantage that we can conservatively summarise the control flow paths leading to each block, and avoid costly analysis of each path in isolation, which could mean analysis of each basic block several times.

The algorithm used to determine the critical path is very similar to that which the scheduler in the processor uses to issue instructions. In the compiler we maintain a structure similar to the processor's issue queue. We place the first few instructions in this pseudo issue queue and then iterate over it several times, removing instructions that are able to issue, recording their writeback times and placing new ones at the tail. Dependences are tracked using the DDG so a child cannot issue earlier than the latest writeback time of its parents. We issue as many instructions as possible, to a maximum of the processor's issue width (8 in our case), and record their writeback times based on their operation latencies.



(a) DDG

Iteration 0: `a` issues
`a` oldest, `a` youngest
Needs 1 entry: `a`
Iteration 1: `b`, `d` issue
`b` oldest, `d` youngest
Needs 3 entries `b,c,d`
Iteration 2: `c`, `e`, `f` issue
`c` oldest, `f` youngest
Needs 4 entries `c,d,e,f`
Overall needs 4 entries

(b) Analysis

Figure 3. Determining the number of IQ entries needed in a DAG

In addition to the issue width we also consider another resource constraint, namely functional unit contention. Two or more instructions conflicting for a resource is modelled as an additional edge in the DDG. Cache misses also affect the critical path but as these are difficult to statically determine we currently assume that all accesses to memory are cache hits.

Knowing how instructions will issue means that the number of IQ entries needed can be determined. On each cycle, the oldest instruction in the queue is known, as is the youngest. By counting the number of instructions between the two in the basic block, we can determine the number of IQ entries needed for them to both be in the queue at the same time, and hence allow the youngest to issue.

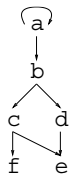
Figure 3 shows the analysis to determine how many IQ entries are needed in a basic block. We assume there are 6 in-order instructions `a,b,c,d,e,f` in this basic block. The example shows the DDG for the start of the basic block and the analysis performed on this small section. The instructions are already in the IQ at the start of the analysis. On iteration 0 only `a` issues so we need only 1 entry. On iteration 1 both `b` and `d` issue so we need 3 entries as the distance between `b` and `d` (`b,c,d`) is 3. Finally, on iteration 2, `c`, `e` and `f` issue so we need 4 entries as the distance between the oldest instruction, `c`, and the youngest, `f`, is 4. Overall, for this small piece of code, we need 4 entries available in the issue queue for everything to issue normally assuming no resource conflicts.

4.3. Loops

Out-of order execution of loops allows instructions from different loop iterations to be executed in parallel leading

1530-0897/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY



(a) DDG

$a_i = a_{i-1} + 1$
$b_i = a_i + 1$
$c_i = b_i + 1$
$d_i = b_i + 1$
$e_i = d_i + 1$
$f_i = c_i + 1$

(b) Initially

$a_i = a_{i-1} + 1$
$b_i = a_{i+1}$
$c_i = a_{i+2}$
$d_i = a_{i+2}$
$e_i = a_{i+3}$
$f_i = a_{i+3}$

(c) Finally

Figure 4. Finding equations for instructions within a loop

to a pipeline parallel execution of the loop as a whole. We therefore have to adjust our analysis accordingly.

In most loops there is a set of instructions that form a cycle of dependences: starting at one instruction and following its dependencies back to the original instruction. We call this a cyclic dependence set, or CDS. In many loops there is more than one CDS with some being subsets or supersets of others. We are interested in the CDS that has the greatest latency; it is this set of instructions which dictates how long the loop will take to execute and is therefore the critical path.

We calculate the number of issue queue entries needed to satisfy the critical dependence across loop iterations by creating an equation for each instruction. This contains both the name of the dependent instruction and the iteration it belongs to. These equations are formed from the data dependence graph and are expressed in terms of instructions within the CDS. Once all instructions have an equation, they are manipulated using a very simple scheme to determine the IQ size needed.

For example, in figure 4(a) instruction `a` is dependent on itself, forming a cycle of dependences and hence the CDS. The equation of this instruction, shown in figure 4(b), refers to a previous instance of itself, the instance of this instruction on the previous iteration of the loop. The equation for `a` means that in iteration `i`, `a` can leave the issue queue 1 cycle after `a` from the previous iteration (`i-1`). Continuing on, `b` can issue 1 cycle after `a` from the current iteration, `c` can issue 1 cycle after `b` from the current iteration, and so on. Rewriting to eliminate constants where possible gives the equations in figure 4(c) where we find that `b` actually leaves at the same time as `a` from the next iteration, `c` and `d` leave at the same time as `a` from two iterations in the future, and `e` and `f` issue at the same time as `a` from the future third iteration.

In order that `a` on iteration `i+3` can be in the issue queue at the same time as `e` and `f` from iteration `i`, 15 entries need to be available. That would allow `e` and `f` from iteration `i`, 12 instructions from iterations `i+1` and `i+2` (6 each), and `a` from iteration `i+3`. Providing this many entries would allow parallel execution of this loop without affecting the critical path.

4.4. Procedure calls

Currently we do not apply inter-procedural dependence analysis. Instead calls to and returns from a procedure are treated as the leaf nodes in the calling DAG/loop, with the called procedure responsible for analysing its IQ size requirements. Within a called procedure the first basic block is assumed to have all its dependencies available. We then apply the analysis described above to the body of the function. On returning from a function call, we restart analysing the IQ requirements for the remainder of the callee procedure. In the special cases where a library routine is called, the IQ is allowed to go to its maximum size immediately before the procedure call.

4.5. Compilation summary

The whole process of analysing a procedure is summarised in figure 5 from breaking into groups to performing the analysis described in section 4.

The complexity of our algorithm is $O(B.P_B.F)$ where B is the number of basic blocks in a procedure, P_B is the number of immediate predecessors of block B, and F is the number of function calls.

5. Results

This section describes the results obtained by our technique in terms of performance and power. First of all we evaluate IQ resizing using special NOOPs inserted into the program (section 5.2). As additional NOOPs may affect processor behaviour, an alternative method is next considered where resizing information is passed via redundant bits in the ISA, which we then evaluate in order to find out to what extent more accurate compiler analysis can improve the performance results (section 5.3).

We evaluate how our technique affects the IQ and the register file as a side effect and compare it to results published by Abella and González in [2, 1]. We compare our results to their *IqRob64* technique as this gave the most power savings and we shall henceforth refer to it as *abella*.

5.1. Simulator and benchmarks

Our processor configuration is shown in table 1 which was implemented in Wattch [6], based on SimpleScalar [7]. We used the MachineSUIF compiler from Harvard [26] to compile the benchmarks, which is based on the SUIF2 compiler from Stanford [28].

1530-0897/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

```
Find natural loops using MachineSUIF
Find DAGs where each DAG starts with the first
   basic block after a procedure call and none of
   its nodes can be part of a loop
Build the DDG for each DAG and loop

For each DAG
   Traverse the DAG breadth-first:
   For each basic block
      Work out which instructions will issue from
         the pseudo issue queue each iteration
      Determine the maximum number of IQ entries
         needed and encode in a special NOOP
   End for
End for

For each loop
   Work out the cyclic dependence set (CDS)
   Create equations such that each instruction
      relates to one in the CDS
   Make each equation's cycle offset zero
   Determine the maximum number of IQ entries
      needed and encode in a special NOOP
End for
```

Figure 5. Algorithm for analysing a procedure

We chose to use the Spec2000 integer benchmarks [27]. We did not use *eon* from this benchmark suite 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* for 100 million instructions after skipping the initialisation part and warming the caches and branch predictor for 100 million instructions. Our processor configuration is the same as *abella* however, they compiled the benchmarks using the Compaq/Alpha compiler.

5.1.1. Compilation time Our initial compiler implementation was coded in a straightforward manner with little regard to efficiency. The time taken to compile the benchmarks varied from less than a minute to over three hours on a Pentium 4, as shown in table 2.

The longest benchmark to compile was *gcc*. This was because the file produced by *bison* contains a large switch statement (374 cases) and many `gotos`, which create a complex control flow graph. As we examine all control-flow paths this leads to excessive compilation time. In a realistic implementation the number of paths examined would be heavily pruned which would be likely to reduce the compile time, in the case of *gcc*, to a few minutes.

Table 1. Processor configuration

Parameter	*Configuration*
Fetch, decode and commit width	8 instructions
Branch predictor	Hybrid 2K gshare, 2K bimodal 1K selector
BTB	2048 entries, 4-way
L1 Icache	64KB, 2-way, 32B line, 1 cycle hit
L1 Dcache	64KB, 4-way, 32B line, 2 cycles hit
Unified L2 cache	512KB, 8-way, 64B line, 10 cycles hit, 50 cycles miss
ROB size	128 entries
Issue queue	80 entries
Int register file	112 entries (14 banks of 8)
FP register file	112 entries (14 banks of 8)
Int FUs	6 ALU (1 cycle), 3 Mul (3 cycles)
FP FUs	4 ALU (2 cycles), 2 MultDiv (4 cycles mult, 12 cycles div)

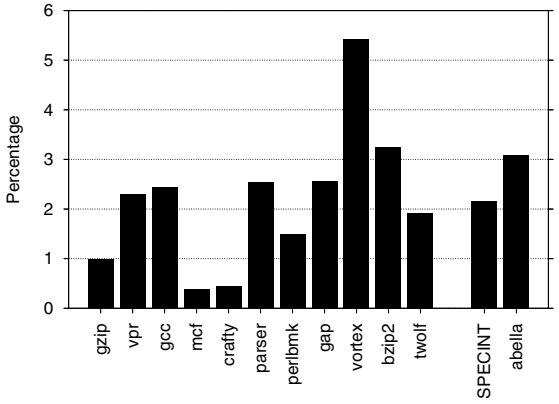


Figure 6. Normalised IPC loss for the NOOP technique

5.2. IQ resizing using special NOOPs

In this section we evaluate our resizing analysis using special NOOPs inserted into the code.

5.2.1. Performance Figure 6 shows the IPC loss for each benchmark. The average loss is 2.2% (denoted by the bar labelled SPECINT) yet *abella* suffers a greater average performance loss of 3.1% and thus our technique is faster by 0.9%.

There is a wide variation between benchmarks. The benchmark with the highest IPC loss is *vortex* at 5.4% and

1530-0897/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

Table 2. Compilation times, in minutes

	gzip	vpr	gcc	mcf	crafty	parser	perlbmk	gap	vortex	bzip2	twolf
Baseline	1	3	64	1	15	3	29	10	13	1	8
Limited	2	4	186	1	58	5	110	23	18	1	38

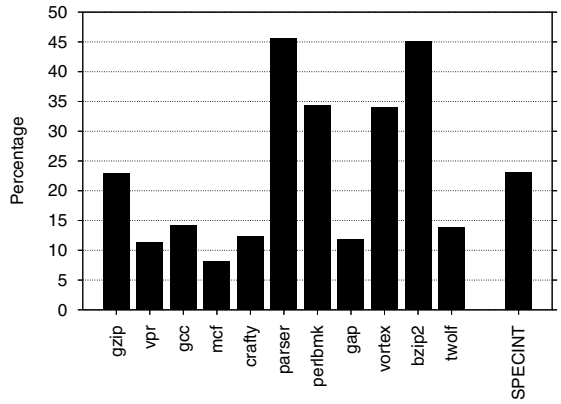


Figure 7. Normalised IQ occupancy reduction for the NOOP technique

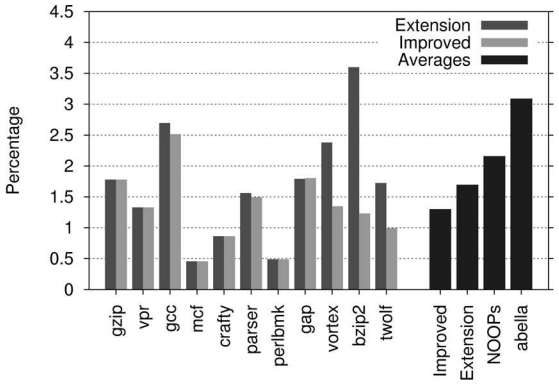


Figure 10. Normalised IPC loss for `Extension` and `Improved`

mcf the lowest with 0.4%. One of the reasons for the performance loss in *vortex* is due to functional unit contention across procedure boundaries which we currently do not analyse. In addition, the special NOOPs are not taken out of the instruction stream until the last decode stage which means that at times only 7 rather than 8 instructions are dispatched in a cycle, leading to a loss of parallelism.

5.2.2. Issue queue The reduction in the number of entries in the IQ is shown in Figure 7. Overall there is an average 23% reduction in the number of entries. This reduction increases the opportunities for power reduction. Dynamic power savings come from waking fewer operands in the IQ and also turning off banks when they are empty. Turning off banks saves static power and also prevents reads and writes to them. On average 37% of all the banks are turned off using our technique compared to 34% with *abella*.

The *nonEmpty* value in figure 8 shows the dynamic power saving achieved in the IQ if only non-empty instructions are woken. The average dynamic power achieved by our technique, however, is 47% and the average static power saving is 31%. Finally, the *abella* technique only achieves a 39% dynamic power and 30% static power reduction even though it has a greater IPC loss (figure 6).

5.2.3. Register file We only consider the integer register file as there are few floating point instructions in the benchmarks used. Delaying the dispatch of instructions (6.8% fewer in our case, 5.1% fewer in *abella*) means that fewer registers are needed simultaneously. By banking them we can turn off those banks that are not in use, saving static and dynamic power. We achieve 22% dynamic and 21% static power savings compared to 14% and 17% achieved by *abella* respectively. Our register file power savings are shown in figure 9.

5.3. Extensions

This section describes two extensions to the NOOP scheme which aim to reduce the performance loss we experience.

In our first technique we consider the case where the IQ resizing may be encoded by tagging instructions. This is beneficial because the special NOOPs are no longer needed, reducing side effects which could hamper performance, as discussed in section 5.2.1. We call this technique `Extension`.

Our second technique, called `Improved`, is derived from the `Extension` technique. We applied, by hand, improved inter-procedural analysis to some of the benchmarks which dominated IPC loss, especially *bzip2*, *vortex* and *gcc*. In particular, we considered functional unit contention across procedure boundaries for the most heavily used procedures. This would typically be available in a mature industrial implementation but is currently absent in our prototype.

Figure 10 shows the IPC loss for our two new techniques. The left-hand bars break it down by benchmark and on the

1530-0897/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

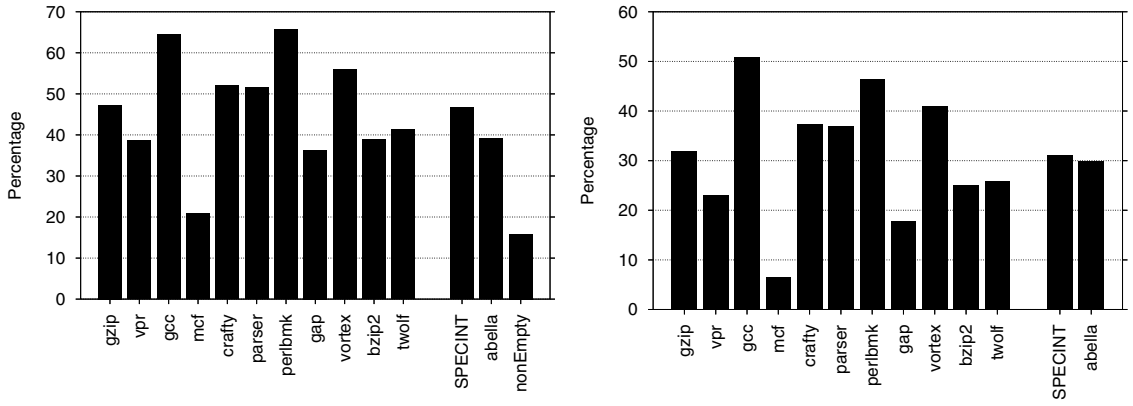


Figure 8. Normalised dynamic and static issue queue power savings for the NOOP technique

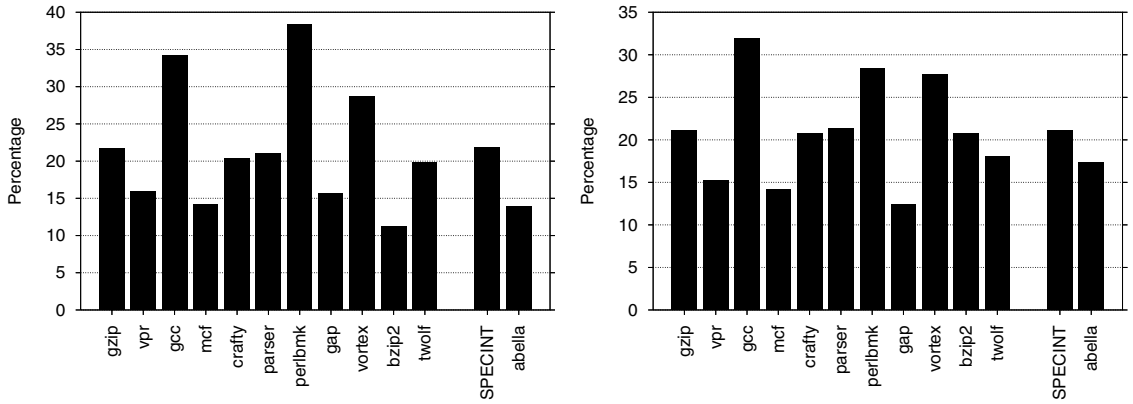


Figure 9. Normalised dynamic and static register file power savings for the NOOP technique

right are the averages. The *abella* and NOOP schemes are shown for comparison.

For `Extension`, when compared to the NOOP insertion scheme, we see that the overall performance loss drops from 2.2% to 1.7%. *Vortex* has a dramatic reduction in IPC loss from 5.4% to 2.4%. This shows that, in this case, the inserted NOOPs affect ILP.

For the `Improved` technique, the IPC loss is less than 1.3% and is primarily due to the improvement in *bzip2*. This benchmark previously had the highest IPC loss showing that inter-procedural functional unit contention was significant. *Gcc* shows little improvement and on inspection, its IPC loss is largely due to the conservative assumptions made by our analysis in the presence of its complex control paths, rather than due to lack of inter-procedural analysis.

Figure 11 shows the dynamic and static power savings from these experiments. The dynamic power savings falls slightly to 45% and static also falls slightly to 30% for both new techniques. The variation across programs is similar to those gained using the special NOOPs (figure 8).

The register file power savings are shown in figure 12. Again, there are few changes and the graphs are roughly the same shape. The dynamic power savings fall to 21% overall from 22% and static remains the same at 21% for `Extension`. For `Improved`, dynamic power savings remain at 22% and static drops to 20%

In summary, passing resize information from the program to the processor without using special NOOPs allows further reduction on IPC loss with little change in power savings. Improved compiler analysis can produce further gains in terms of reduced IPC loss.

1530-0897/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

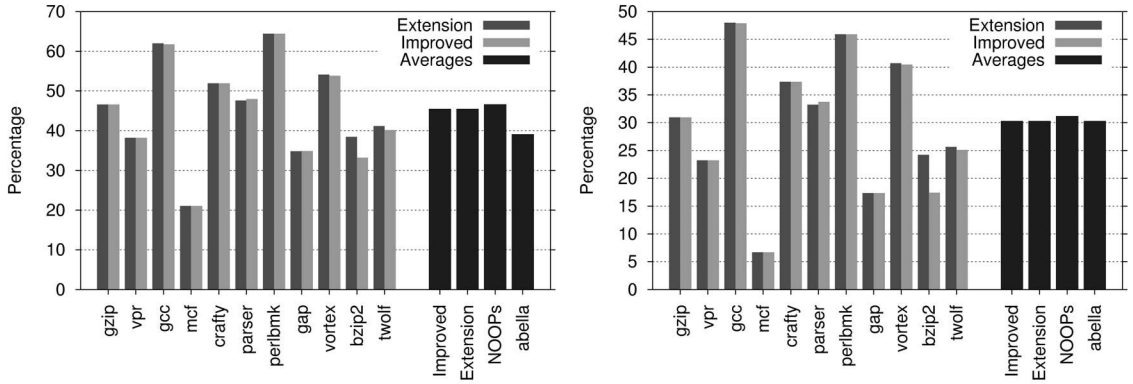


Figure 11. Normalised dynamic and static IQ power savings for `Extension` and `Improved`

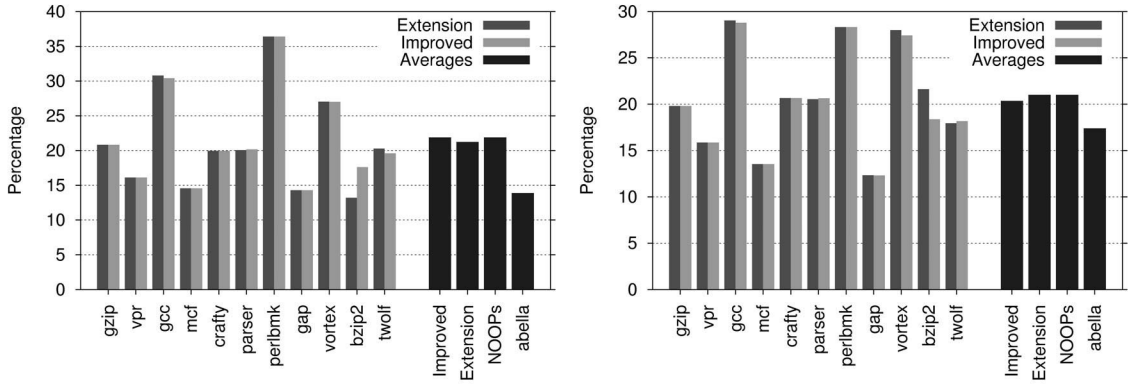


Figure 12. Normalised dynamic and static register file power savings for `Extension` and `Improved`

6. Conclusions

We have presented a novel technique to dynamically resize the issue queue using the compiler for support. The compiler analyses and determines the number of IQ entries needed by each basic block in a program and encodes this number in a special NOOP which is used to limit the number of instructions in the queue. This has the effect of reducing the issue queue occupancy and thus the amount of power dissipated.

Results from the implementation and evaluation of the proposed technique show 47% dynamic power savings and 31% static power savings in the IQ with an IPC loss of only 2.2%. This is compared to 39% and 30% dynamic and static power savings, and 3.1% IPC loss by Abella and González [2]. A side effect of limiting the number of instructions in the IQ is that fewer registers are needed and thus dynamic power savings of 22% and static power savings of 21% are also achieved in the integer register file.

By tagging instructions and using further improved analysis we can reduce the IPC loss to less than 1.3% with a 45% dynamic power and 30% static power saving. This corresponds to overall processor dynamic power savings of 11%, assuming the issue queue and integer register file consume 22% and 11% of the whole processor's power respectively.

In summary, therefore, our technique has a smaller performance loss, saves more power than state-of-the-art approaches and needs less complex hardware.

Acknowledgements

This work has been partially supported by The Ministry of Education and Science under grants TIC2001-0995-C02-

IEEE COMPUTER SOCIETY

01, TIN2004-03072, FEDER funds, and Intel Corporation.

References

[1] J. Abella and A. González. Power-aware adaptive instruction queue and rename buffers. Technical Report UPC-DAC-2002-31, Universitat Politècnica Catalunya, 2002.

[2] J. Abella and A. González. Power-aware adaptive issue queue and rename buffers. In *HiPC, LNCS2913*, 2003.

[3] J. Abella and A. González. Low-complexity distributed issue queue. In *HPCA*, 2004.

[4] R. I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *ISCA*, 2001.

[5] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis. Energy and performance improvements in microprocessor design using a loop cache. In *ICCD*, 1999.

[6] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA*, 2000.

[7] D. Burger and T. Austin. The simplescalar tool set, version 2.0. Technical Report TR1342, University of Wisconsin-Madison, 1997.

[8] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power at high performance. In *PACS*, volume 2008 of *Lecture Notes in Computer Science*. Springer, 2000.

[9] R. Canal and A. González. Reducing the complexity of the issue logic. In *ICS*, 2001.

[10] T. C. I. Center. http://bwrc.eecs.berkely.edu/cic/.

[11] J. Emer. Ev8 the post-ultimate alpha. In *Keynote at PACT*, 2001.

[12] D. Ernst, A. Hamel, and T. Austin. Cyclone: A broadcast-free dynamic instruction scheduler with selective replay. In *ISCA*, 2003.

[13] D. Folegnani and A. González. Energy-effective issue logic. In *ISCA*, 2001.

[14] C.-H. Hsu and U. Kremer. Compiler-directed dynamic voltage scaling for memory-bound applications. Technical Report DCS-TR-498, Rutgers University, 2002.

[15] J. S. Hu, N. Vijaykrishnan, and M. J. Irwin. Exploring wakeup-free instruction scheduling. In *HPCA*, 2004.

[16] M. Huang, J. Renau, and J. Torrellas. Energy-efficient hybrid wakeup logic. In *ISLPED*, 2002.

[17] C. Lee, J. K. Lee, T. Hwang, and S.-C. Tsai. Compiler optimization on instruction scheduling for low power. In *ISSS*, 2000.

[18] M. Lorenz, R. Leupers, P. Marwedel, T. Dräger, and G. Fettweis. Low-energy DSP code generation using a genetic algorithm. In *ICCD*, 2001.

[19] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *ISCA*, 2003.

[20] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *ISCA*, 1998.

[21] R. Maro, Y. Bai, and R. I. Bahar. Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In *PACS*, volume 2008 of *Lecture Notes in Computer Science*. Springer, 2000.

[22] S. Önder and R. Gupta. Superscalar execution with dynamic data forwarding. In *PACT*, 1998.

[23] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *ISCA*, 1997.

[24] M. C. Toburen, T. M. Conte, and M. Reilly. Instruction scheduling for low power dissipation in high performance microprocessors. In *ISCA*, 1998.

[25] W. Zhang, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, D. Duarte, and Y.-F. Tsai. Exploiting VLIW schedule slacks for dynamic and leakage energy reduction. In *MICRO-34*, 2001.

[26] Machine SUIF. http://www.eecs.harvard.edu/machsuif/-software/software.html.

[27] SPEC. http://www.spec.org/.

[28] SUIF. http://suif.stanford.edu/.

1530-0897/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY