Instruction Packing: Toward Fast and Energy-Efficient Instruction Scheduling

JOSEPH J. SHARKEY, DMITRY V. PONOMAREV, and KANAD GHOSE

State University of New York at Binghamton

and

OGUZ ERGIN

TOBB Economics and Technology University

Traditional dynamic scheduler designs use one issue queue entry per instruction, regardless of the actual number of operands actively involved in the wakeup process. We propose Instruction Packing—a novel microarchitectural technique that reduces both delay and power consumption of the issue queue by sharing the associative part of an issue queue entry between two instructions, each with, at most, one nonready register source operand at the time of dispatch. Our results show that this technique results in 40% reduction of the IQ power and 14% reduction in scheduling delay with negligible IPC degradations.

Categories and Subject Descriptors: C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms: Power, Performance, Superscalar Processors

Additional Key Words and Phrases: Issue queue, low power, instruction packing

1. INTRODUCTION

Dynamically scheduled out-of-order microprocessors use aggressive instruction scheduling mechanisms to maximize performance across a wide variety of applications. In such designs, instructions are typically dispatched into the issue queue (IQ) in-order after undergoing register renaming and noting the

A preliminary version of this paper entitled “Instruction Packing: Reducing Power and Delay of the Dynamic Scheduling Logic,” J. Sharkey, D. Ponomarev, K. Ghose, O. Ergin, appeared in the *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2005. This research was supported in part by the National Science Foundation, award numbers CNS 0454298 and EIA 9911099, and by the Integrated Electronics Engineering Center at SUNY-Binghamton.

Authors' address: J. Sharkey, D. Ponomarev, K. Ghose, Department of Computer Science, State University of New York, Binghamton, NY 13902-6007; email: {jsharke,dima,ghose}@cs.binghamton.edu; O. Ergin, TOBB Economics and Technology University, Ankara, Turkey; email: oergin@etu.edu.tr.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.

© 2006 ACM 1544-3566/06/0600-0156 \$5.00

Fig. 1. The percentage of dynamic instructions that enter the scheduling window with 0, 1, and 2 operands requiring comparators.

availability of their register sources, they are issued for the execution out of order as their source operands become available. The IQ tracks the availability of the source register operands for the instruction and then selects the ready instructions for the execution. The instruction scheduling logic operates in two phases—instruction wakeup and instruction selection [Palacharla et al. 1997]. During wakeup, the destination tags (register addresses) of the already scheduled instructions are broadcast across the IQ and each IQ entry associatively compares the locally stored source register tags against the broadcasted destination tag. On a match, the corresponding source is marked as ready. When all sources of an instruction become ready, the instruction wakes up and becomes eligible for selection. The selection logic then selects W out of possibly N eligible instructions, where N is the size of the IQ and W is the processor's issue width.

In order to discover and exploit a sufficient degree of instruction-level parallelism (ILP), the IQs in modern processors have to be generously sized, resulting in significant power/energy dissipations in the course of accessing the queue. A large amount of energy is dissipated when the destination tags are broadcast on the tag buses because of the charging and discharging of the wire capacitance of the tag line itself and the gate capacitance of the devices that implement the tag comparators. In addition, the power is also dissipated in the course of instruction dispatching (writing the instructions into the IQ) instruction selection, and instruction issuing (reading the instructions out of the queue). Several researchers have reported the IQ power to account for 20–25% of total chip power [Gowan et al. 1998; Wilcox et al. 1999].

In a traditional RISC-like processor where each instruction can have, at most, two register source operands, each IQ entry is implemented with two comparators, which allow the instruction located in this entry to track the arrival of both sources by monitoring the tag buses. In general, however, such a design results in a grossly inefficient usage of the comparators because (1) many instructions have only one source register operand and, therefore, do not require the use of two tags (and two comparators) in the first place, and (2) in instructions with two register source operands, a large percentage have at least one of these operands ready at the time of dispatch, again rendering the second comparator unnecessary. Our simulations showed that, on average across SPEC 2000 benchmarks, about 81% of the dynamic instructions enter the scheduling window with at least one of their source operands ready, as shown in Figure 1. Similar results were also presented in some earlier works [Ernst and

Austin 2003a; Sharkey et al. 2005]. (These results are obtained using a pipeline model depicted later in Figure 7; the nonspeculatively set register ready bits were checked to identify the readiness of source operands for this graph. More details about some intricacies associated with this terminology are presented in Section III, these specifics are not essential for understanding the material presented in Sections I and II).

In this paper, we propose a novel scheduler design, which optimizes the use of the CAM logic (comparators) within the IQ by the *opportunistic packing* of two instructions into the same IQ entry, effectively duplicating some of the RAM storage for these entries (destination register addresses, literals, and opcodes) and sharing the existing comparators. When an instruction, which has at least one of its two source register operands ready at the time of dispatch, is placed in a traditional IQ entry with two comparators, the comparators corresponding to the ready source is unused. By placing or *packing* two such instructions within the same IQ entry, such wastages can be reduced and, in many cases, totally eliminated. Instruction packing reduces the number of IQ entries. This, in turn, allows shorter bitlines and tag buses to be used, resulting in faster instruction wakeup. Instruction packing thus achieves a significant energy reduction and faster operation of the wakeup logic with negligible degradation in the IPC and just a slight increase in the delay of the selection logic.

Other researchers [Ernst and Austin 2002] previously proposed to *statically* partition the IQ into the entries with 0, 1, and 2 comparators, exploiting the same observation that only a relatively small percentage of instructions come into the pipeline with both of their source registers nonready. The key difference between the work of Ernst and Austin [2002] and this paper is that instruction packing *dynamically* allocates either a full-entry or a half-entry of the queue to an instruction and, thus, it can rapidly adapt to the characteristics of executing applications, thus encountering lower IPC degradation compared to the scheme of Ernst and Austin [2002], as we demonstrate in the results section.

We perform detailed microarchitectural and circuit-level simulations of the proposed mechanism to assess the instruction-packing scheme. Our experimental assessments show that, for the schedulers with the capacity to hold up to 64 instructions, instruction packing reduces the average overall IQ power (excluding the selection logic) by 40.2% and reduces the wakeup delay by about 26% at the cost of a commit IPC degradation of only 0.6% for a four-way machine. On an aggressive eight-way machine, this IPC loss is limited to 3.9%, on average, and can be further reduced to 2.4% using the aggressive mechanism described in the paper.

The basic instruction-packing technique, as described above, was introduced in Sharkey et al. [2005]. This submission extends this work in the following ways:

We provide a much more detailed description of the instruction selection/issue logic.

- We examine the implications of instruction packing on load-related speculation techniques. Specifically, we describe how it is possible to support the squash-recovery model (used in the Alpha 21264 processor [Compaq

Computer Corp. 1999]) in which instructions must be reissued from the IQ following a load-related misspeculation. We show that the additional constraints necessary to support this type of replay model do not significantly hinder the performance of instruction packing compared to the traditional scheduler designs.

- We present results for the entire set of SPEC 2000 integer and floating-point benchmarks using the SimPoints framework [Sherwood et al. 2002] for selecting representative instruction windows.
- We evaluate the performance of the instruction packing on a variety of processor configurations, including those with large instruction windows. We demonstrate through our results that even for the processors with a large issue width and large Reorder Buffers (ROBs), reasonably small IQs can be used (without creating a bottleneck) if the instruction packing is implemented, partially solving the IQ scalability problem with a large number of in-flight instructions.
- We propose and evaluate a modification to the basic instruction-packing scheme, where a full-size IQ entry initially allocated to an instruction with two nonready source operands is converted “on-the-fly” to a half-entry when one of the source operands becomes ready, making the other one-half of the entry available for housing a new instruction. As in many cases, there is a significant slack between the arrivals of two sources, such a modification further increases the IQ utilization and minimizes the performance losses encountered because of packing. Results show that this mechanism reduces the IPC degradation experienced because of instruction packing by nearly 50% on some configurations.

The rest of the paper is organized as follows. Section 2 details the basic instruction-packing scheme, as proposed in Sharkey et al. [2005]. The synergies of instruction packing with speculative scheduling techniques are addressed in Section 3. Our modifications to instruction packing are described in Section 4. Section 5 describes our simulation methodology, our simulation results are presented in Section 6. We describe the related work in Section 7 and offer our concluding remarks in Section 8.

2. INSTRUCTION PACKING: THE BASIC SCHEME

Figure 2a shows a format of an IQ entry used in traditional processors. A single IQ entry is comprised of the following fields: (a) entry allocated bit (A), (b) payload area (opcode, FU type, destination register address and literals), (c) tag of the first source, associated comparator (tag CAM word 1, hereafter just tag CAM 1, without the “word”) and the source valid bit, (d) tag of the second source, associated comparator (tag CAM 2), and source valid bit, and (e) the ready bit. The ready bit, used to raise the request signal for the selection logic is set by AND-ing the valid bits of the two sources.

If at least one of the source operands is ready at the time of dispatch, the tag CAM associated with this instruction’s IQ entry remains unused. To exploit this idle tag CAM, we propose to share one IQ entry between two such instructions.

Fig. 2. Formats of the traditional issue queue entry (a), and an entry of the issue queue that supports instruction packing (b).

An entry in the IQ can now hold one or two instructions, depending on the number of ready operands in the stored instructions at the time of dispatching. Specifically, if both source registers of an instruction are not available at the time of dispatch, the instruction is assigned an IQ entry of its own and makes use of both tag CAMs in the assigned entry to determine when its operands are ready. An instruction that has only one source register that is not available at the time of dispatch is assigned just one-half of an IQ entry. The remaining one-half of the IQ entry may be used by another instruction that also has one of its source registers unavailable at the time of dispatch. Sharing an IQ entry between two instructions also requires the IQ entry to be widened to permit the payload parts of both instructions to be stored, along with the addition of flags that indicate whether the entry is shared between two instructions and the status of the stored instruction(s). Figure 2b shows the format of an IQ entry that supports instruction packing. Each IQ entry is comprised of the “entry allocated” bit (A), the ready bit (R), the mode bit (MODE), and the two symmetrical halves: the left and the right half. The structure of each is identical, so we will use the left half for the subsequent explanations.

A left half of each IQ entry contains the following fields:

- Left half allocated (AL) bit. This bit is set when the left half-entry is allocated.
- Source tag and associated comparator (Tag CAM). This is where the tag of the nonready source operand for an instruction with, at most, one nonready source is stored.
- Source valid left bit (SVL). This bit signifies the validity of the source from part (b), similar to the traditional designs. This bit is also used to indicate if the instruction residing in a half-entry is ready for selection (as explained later)
- Payload area. The payload area contains the same information as in the traditional design, namely, opcode, bits identifying the FU type, destination register address, and literal bits. In addition, the payload area contains the tag of the second register source. Notice that the tag of the second source does not participate in the wakeup, because if an instruction is allocated to a half-entry, the second source must be valid at the time of dispatch. Compared to the traditional design, the payload area is increased by the number of bits used to represent a source tag.

The contents of the right half are similar. The ready bit (R) is used when an instruction with two nonready source operands is allocated into the full IQ entry, as explained below. In summary, each entry in the modified IQ is divided into a left and a right half each of which is capable of storing an instruction with, at most, one nonready source operand, or the two halves can be used in concert to house an instruction with two nonready source operands.

In general, the IQ entry can be in one of the following three states: (1) the entry holds a single instruction, both register source operands of which were not ready at the time of dispatch, (2) the entry holds two (or one with another one-half free) instructions, each of which had at least one source operand ready at the time of dispatch, or (3) the entry is free. The “mode” bit, stored within each IQ entry as shown in Figure 2b, identifies the state of the entry. If the mode bit is set to 1, then the entry maintains a two-operand instruction; otherwise, it either maintains one or two single-operand instructions or it is free.

Since each entry can hold up to two instructions, fewer IQ entries are needed by this design. However, despite the fact that each entry in the modified IQ shown in Figure 2b is somewhat wider than the traditional queue entry (because of the replication of the Payload area and three extra bits—AL, AR, and MODE), the amount of CAM logic per-entry does not change. Each entry still uses only two sets of comparators—those are either used by one instruction that occupies the full entry, or are shared by two instructions, each located in a half-entry. In the next few subsections, we describe the details of this technique.

2.1 Entry Allocation

To set up an IQ entry for an instruction, the “*entry allocated*” bits corresponding to both halves (AL and AR), as well as the global *entry_allocated* bit (A) are associatively searched in parallel with register renaming and checking the status of the source physical registers. If the instruction is determined to have at most

one nonready source operand at the time of dispatch, the lowest numbered IQ entry with at least one available half is allocated. If both halves are available within the chosen entry, then the instruction is written into the right half. Such an assignment of IQ entries avoids the fragmentation with the IQ, as the entries are assigned from one end of the queue. Consequently, whenever possible the instructions with one nonready source operand will be truly packed within the same IQ entry, as opposed to being distributed across distinct entries, thus avoiding the fragmentation.

After the appropriate half is chosen, both the “*entry_allocated*” bit of this half and the global A bit are reset. If an instruction is determined to have two nonready source operands at dispatch, then a full-sized entry is allocated, as dictated by the state of the A bits and the instruction payload is placed into the right half of the entry (for reasons discussed later). The search for a full-sized and a half-sized entry occurs simultaneously and the entry to be allocated is then chosen based on the number of nonready source operands. This IQ entry allocation process is somewhat more complicated than a similar allocation used in traditional designs, where just the A bits are associatively searched. However, there is no extra delay involved, because the searches occur in parallel. Similar issues with allocating the IQ entries are also inherent in other designs, which aim to reduce the amount of associative logic in the queue by placing the instructions into the IQ entries judiciously, based on the number of nonready operands at the time of dispatch [Ernst and Austin 2002]. We will discuss what kind of information is written into the IQ for the various instruction categories later in the paper. First, we describe how wakeup and selection operations are implemented in this scheme.

2.2 Instruction Wakeup

The process of instruction wakeup remains exactly the same as in traditional design for an instruction that occupies a full IQ entry (i.e., enters the queue with two nonready register sources). Here, the ready bit (R) is set by AND-ing the valid bits of both sources. For instructions that occupy half of an IQ entry, the wakeup simply amounts to setting of the valid bit corresponding to the source that was nonready when the instruction entered the IQ. The contents of the source valid bits are then directly used to indicate that the instruction is ready for selection (the validity of the second source is implicit in this case). The selection logic details are described next.

2.3 Instruction Selection

The process of instruction selection needs to be slightly modified to support instruction packing. To make the explanation easier, we assume that a 32-entry IQ is packed into a 16-entry structure, such that each entry is capable of holding two instructions with, at most, one nonready source each, or one instruction with two nonready sources. In a traditional design with 32-entry IQ, there are 32 request lines that can be raised by the awakened instructions—one line per IQ entry. In the instruction-packing scheme, each of the two halves of each of the 16 entries requires a request line, thus retaining the same total number of

request lines (32) and resulting in a similar complexity of the selection logic. In addition, the ready bits, used by the instructions allocated to full entries, also require request lines. Consequently, a straightforward implementation of the selection logic would require 48 (3×16) request lines, thus increasing the complexity, delay, and power requirements of the select mechanism. In other words, such an implementation would require three request lines per IQ entry—one each for R, SVR, and SVL bits.

Such an undesirable elevation in the complexity of the selection logic can be avoided by sharing one request line between the R and the SVR bits. The shared request line is raised if at least one of the bits (the R or the SVR) is set. The R and the SVR bits are both connected to the shared request line through a multiplexer, which is controlled by the “mode” bit of the IQ entry (Figure 2b). As a result, only two request lines per IQ entry are needed—twice as many as in traditional designs. However, if the total number of IQ entries is reduced by half (which is realizable with very small IPC impact), then the total number of request lines remains unchanged compared to the baseline machine.

Consequently, the overall delay of the selection logic increases only slightly—by the delay of a multiplexer. Notice also that the MUX control signal (the “mode” bit) is available in the beginning of the cycle when the selection process takes place (the “mode” bit is set when the IQ entry is allocated). The request line driven by the SVL bit is controlled by the p-device, whose gate is connected to the “mode” bit. This request line will be asserted only if the “mode” bit is set to 0 (indicating that the IQ entry is shared between two instructions) and the SVL bit is set to 1.

Note that the only part of the selection logic that is modified is the process of asserting the request lines. The rest of the selection logic remains unchanged compared to the traditional designs. The overall delay of the selection logic is thus increased by the delay of the multiplexer, whose control signal is preset (as the value of the “mode” bit is available as soon as the IQ entry is allocated). The additional delay introduced in path that generates the request signal is thus the propagation delay of a turned-on CMOS switch within the multiplexer (about 18 ps in our 0.18 μm CMOS layouts; see Section 5 for details of our methodology). Compared to the delay of a 3-level, tree-structured selection logic (about 370 ps, *neglecting wire delays*), this additional delay is negligible, only about 5%. This percentage, of course, goes down as wire delays in the selection tree are accounted for.

2.4 Instruction Issue

We define instruction issue as a process of reading the source operand tags of the selected instructions and starting the register file access (effectively moving the instruction out of the IQ). When a grant signal comes back corresponding to a request line, which was shared between the R and the SVR, the issue logic has to know which physical registers need to be read. Conventionally, this information is conveyed by the contents of the source register tag fields. However, the register tags of an instruction with two nonready sources (i.e., the

instruction that occupies full IQ entry) and the register tags of an instruction with one nonready source are generally stored in different locations within the IQ entry. In the former case, the tags are stored in the tag fields connected to both comparators—one tag is stored in the left half of the entry and the other tag is stored in the right half of the entry. In the latter case, both tags are stored in the right half of the entry, such that the tag of the nonready operand is connected to the comparator and the other tag is simply stored in the payload area. Given this disparate locations of the source register tags, how would the issue logic know which tags to use when the grant signal corresponding to a shared request line comes back?

One solution is, again, to use the contents of the “mode” bit and a few multiplexers. This will, however, slightly increase the delay of the issue/register access cycle. A better solution, which avoids the additional delays in instruction issuing altogether, is as follows. When an instruction with two nonready sources is allocated an IQ entry, the source tag, which is connected to the left-half comparator, is replicated in the payload area storage for the second tag in the right half. As a result, both tags will be present in the right half of the queue, so these tags can be simply used for register file access, without regard for the IQ entry mode.

2.5 Benefits of Instruction Packing

Instruction packing, as described in this section, has several benefits over the traditional IQ designs in terms of the layout area, the access delays, and the power consumption.

The delay of the wakeup logic is reduced, because IQs with smaller number of entries can be used with packing (as we demonstrate in the results section, the issue queue size can be reduced by one-half with virtually no loss in the IPCs). As a result, the tag buses become much shorter and the capacitive loading on these buses is also significantly reduced—the delay in driving the tag bus (which is a major component of the wakeup latency) is roughly reduced by one-half. Furthermore, shorter bitlines reduce the IQ access delays during instruction dispatching (setting up the entries) and issuing (reading out the register tags and literals). Finally, for the same reasons, the power consumptions of instruction dispatch, issue, and wakeup are also reduced. Another potential reason for the reduction in the power consumption has to do with the use of fewer comparators. In the instruction-packing, the tags of the source registers ready during dispatching are never associated with the comparators. In the traditional designs, each and every source tag is hooked up to a comparator. Unless these comparators are precharged selectively (based on whether or not a given IQ slot is awaiting for the result), unnecessary dissipations can occur when comparators associated with the already valid sources discharge on subsequent mismatches.

Finally, with instruction-packing, the area of the IQ decreases. The main reason behind this is the reduced amount of associative logic needed within the scheduler. At the same time, the amount of RAM storage remains virtually unchanged.

In the results section, we quantify these savings using detailed simulations of SPEC 2000 benchmarks and also circuit simulations of the IQ layouts. Notice that all these benefits are achieved with essentially no degradation in the IPCs (committed Instructions per Cycle). This is because most instructions (our results show 81%) have at least one of their sources ready at the time of dispatch, thus rendering the performance loss due to the smaller number of IQ entries negligible.

3. SYNERGY OF INSTRUCTION PACKING WITH SPECULATIVE SCHEDULING

For the sake of clarity in presenting the concepts of instruction-packing, our discussions in the previous sections ignored the issues that arise when the scheduling logic with reduced number of tags (such as instruction-packing, or tag elimination technique of [Ernst and Austin 2002]) is incorporated into the datapath that supports speculative scheduling based on load hit/miss prediction. In such designs, the instructions dependent on a load (and possibly also their dependents) are scheduled speculatively, assuming that the load will hit into the L1 D-cache or relying on a more elaborate hit/miss prediction information. Upon a misprediction, the prematurely scheduled instructions need to be re-executed (replayed). The comprehensive treatment of such scheduling mechanisms and various associated replay schemes is presented in Kim and Lipasti [2004]. As the number of stages between issue and execution increases in deeply pipelined high-frequency machines, it is essential that all proposed scheduling solutions support such speculation.

Schedulers with a reduced number of tags, such as the ones described in this paper, present complications in this respect, because some of the source operand tags stored in the IQ do not monitor the wakeup buses (i.e., do not have comparators associated with them) [Ernst and Austin 2003a]. The problem arises when these instructions need to be replayed following a load latency misprediction and the source operand that was (speculatively) determined to be ready in the course of the initial instruction dispatching is no longer ready after the misprediction transpires. This occurs when the source operand in question depends, directly or indirectly, on the mispredicted load. In such a situation, the wakeup buses need to be monitored to determine the readiness of the source in a nonspeculative manner. However, since the tag of this source operand does not have associated CAM logic, such monitoring is impossible. Notice that the Tag Elimination mechanism of Ernst and Austin [2002] has similar problems.

The aforementioned issue can be addressed through several possible solutions. The first solution is to flush all instructions in the pipeline that are younger than the mispredicted load and refetch them. This solution is undesirable because of the large IPC overhead. Other solutions rely on the ability to “replay” instructions following a misprediction without resorting to refetching. For such solutions, there are two generic methods for handling the issued instructions, which are still potentially subject to the load-latency related replays. One option is to keep these speculatively issued instructions in the issue queue until they become replay-free, at which point the corresponding IQ

entries are deallocated. In this option, they can be reissued from the IQ following a misprediction. Notice that as the number of cycles between issue and execution increases, a larger fraction of the issue queue is occupied by such instructions, effectively shrinking the scheduling window. An alternative solution is to deallocate the IQ entries immediately after the instruction issue and maintain the instruction in the load shadow (the ones subject to replays) in a separate buffer [Ernst and Austin 2003a; Merchant and Sager 2001] from where they can later be re-executed. In [2003], Ernst and Austin describe how such a replay mechanism can be used with tag elimination. We note that it is also possible to use instruction-packing in such an environment without any further changes. Because of the space limitation, we will not discuss this particular replay mechanism any further and instead refer the readers to Ernst and Austin [2003a] and Merchant and Sager [2001] for a more detailed discussions and the power/performance trade-offs that arise. It is likely that if instruction-packing is integrated into the scheduler where the replays are handled through a separate queue, then the resulting trade-offs will be very similar to those presented in Ernst and Austin [2003a]. Of course, the dynamic nature of queue partitioning in instruction-packing will still provide some advantages in terms of IPCs compared to the static partitioning as used by the tag elimination scheme.

For the organization where both regular and replaying instructions are issued out of the same IQ, several solutions are possible to support instruction-packing. First, the nonspeculative version of the “register ready” bit-vector can be maintained, in addition to (or instead of) speculative version. A nonspeculative ready bit associated with a physical register is only set when the instruction producing that register value is no longer subject to the load-latency related replays. Basically, the speculative and the nonspeculative ready bits will be set a few cycles apart from each other and that difference is determined by the number of stages between instruction issue and execution. If we check the nonspeculative version of the ready bits (instead of speculative) and make the decision to allocate a half or full IQ entry based on that, then instruction-packing can trivially support speculative scheduling and various forms of replays (such as squash recovery [Compaq Computer Corp. 1999] or selective recovery [Kim and Lipasti 2004; Merchant and Sager 2001]) without any problems. Of course, as the schedule-to-execute pipeline depth increases, the effectiveness of instruction-packing will decrease, as a larger percentage of instruction will require full-sized IQ entries. But then, again, in such situations the overall efficiency of using the IQ will decrease (just as it will in the traditional IQ case) as larger fraction of entries will be allocated to the issued instructions waiting for the verification of the load latency prediction. In these circumstances, the use of a separate queue for replays could be a more justifiable solution. In any case, in the results section, we investigate the performance of instruction-packing if the allocation of the IQ entries is based on the nonspeculative version of the ready bits and show that the performance and power impacts are minimal for the schedule-to-execute pipeline depth up to eight stages. (For comparison with the idealistic case that uses speculative ready bits we, of course, used perfect prediction, because otherwise speculative scheduling can not be supported with packing. For all other experiments, real predictors were

used—see the results section for details). In other words, while the speculative scheduling itself provides performance benefits, the impact of the instruction-packing mechanism (relative to the corresponding baseline model) is about the same for the processors with or without speculative scheduling.

4. AGGRESSIVE INSTRUCTION PACKING: RELEASING IQ HALVES BEFORE INSTRUCTION ISSUE

In the basic instruction-packing scheme described Section 2, a full-sized IQ entry (comprised of both a left and a right half) is allocated for an instruction with two nonready register source operands at the time of dispatch and this entry stays allocated until the instruction issues and is removed from the IQ. Such a conservative assignment of entries makes the performance of instruction-packing highly dependable on the statistics presented in Figure 1. In cases where a large percentage of instructions enter the scheduling window with two nonready register sources, the performance losses encountered because of instruction-packing could be significant. In this section, we investigate a more aggressive mechanism to further increase the efficiency of instruction-packing and make the design less sensitive to the statistics of Figure 1.

The first observation that we exploit is that very rarely both sources of an instruction become ready at the same time. In most situations, there exists a considerable slack (at least a few cycles) between the arrivals of the sources—this fact has also been exploited by the work of Kim and Lipasti [2003a]. Instead of pinning a full IQ entry for an instruction with two nonready sources throughout the entire duration of instruction's residency in the IQ, we initially allocate a full-size entry for such instruction, and then, when one of the sources arrives, we dynamically convert the status of this instruction to indicate that it has only one nonready source, and make the half of the entry corresponding to the source, that already arrived, available for future instructions.

When an instruction with two nonready register source operands is initially dispatched, the MODE bit of the corresponding IQ entry is set to 1 and the payload area is replicated in both halves. Notice that the payload area within each half stores the source tag of the register, connected to the comparator in the other half. When the first operand becomes available, the MODE bit of the entry is reset to 0, AL and AR bits are updated accordingly, and the half-entry corresponding to the matching comparator is immediately freed up for the use by newer instructions. Effectively, the instruction is converted “on-the-fly” from being treated like the one with two nonready register sources to being treated like the one with just one nonready source by the instruction-packing scheme. As a result, the efficiency of using the IQ further improves as half-entries are released at the first possible opportunity.

There are several ways of implementing the dynamic conversion and handling of an instruction from one that initially has both sources unavailable to one that has just a single source. The key observation of any such mechanism is that when the conversion is complete, the instruction must reside in a correctly formatted half-entry: including the correct state of mode and allocated bits and the correct location of the payload (in the payload left or payload right area).

Recall that the entry allocation mechanism for basic instruction-packing, as presented in the previous section, reports that the payload of an instruction allocated to a full-width entry is placed in the right-hand payload area. Therefore, the compaction mechanism must take some additional measures so as to guarantee the correct location of the payload after the compaction is complete.

The first such compaction mechanism can simply read the data out of the right-hand payload area and write it to the left-hand payload area during the “on-the-fly” conversion, and then set the mode and allocated bits appropriately. This, however, requires reading and writing of the entire payload area during the conversion and the addition of a dedicated set of wires and read/write ports on the payload areas. Such requirements are undesirable as there is an impact on both area and delay.

An alternative approach will be to allow for the decoupling of the payload area and the tag CAM areas. Here, marker bits are added to each of the two payload areas to indicate which half of the entry—left or right—each payload field is associated with and whether each half contains valid data. As in the first case, the mode and marker bits have to be updated when the entry is converted on-the-fly. In this second variation, these marker bits have to be used to steer out the payload data field to the output of the IQ when an instruction assigned to one of the halves is selected for issue. This variation, therefore, adds a multiplexing delay in the instruction issue path.

Finally, a third possible mechanism duplicates the payload RAMs for both halves of the IQ entry when the instruction with both sources unavailable is dispatched. This mechanism reduces the conversion process to simply changing the mode and allocated bits, as the payload is already present in both locations. This replication will result in increased power consumption as compared to the basic instruction-packing. However, this only occurs for instructions that are allocated a full-width entry at the time of dispatch, which is a relatively small percentage of all instructions (19%, on average, as presented in Figure 1). Furthermore, instruction issue and dispatch are otherwise unchanged from that presented in Section 2. The overall power impact of this mechanism is expected to be quite small compared to the total IQ power. In the results reported later, we present figures on this variation in which the payload is replicated, since the other approaches result in increased delay in the instruction issue path, which is probably undesirable.

The aggressive packing mechanism, as described thus far, is sufficient for the machines that either do not speculate on the loads, or make use of a separate replay queue for handling load-related mispredictions [Kim and Lipasti 2004; Ernst and Austin 2003a; Merchant and Sager 2001], or use the flush and refetch recovery. However, additional considerations, alluding to the timing of the release of the IQ entry halves, must be taken into account for the machines that support a replay model in which instructions must be reissued from the IQ [Kim and Lipasti 2004; Compaq Computer Corp. 1999]. Specifically, the valid bits for each source register in the IQ entry (VL and VR in Figure 2b) are set speculatively in such machines. If it is later determined that an instruction, which early released one of its halves must be replayed (due to a misprediction), then the released half may be needed once again to correctly implement

Table I. Configuration of the Four-Way Simulated Processor

Parameter	Configuration
Machine width	4-wide fetch, 4-wide issue, 4 wide commit
Window size	Issue queue—as specified, 48 entry load/store queue, 96-entry ROB
Function units and latency(total/issue)	4 Int Add (1/1), 2 Int Mult (3/1)/Div (20/19), 2 Load/Store (2/1), 2 FP Add (2), 2 FP Mult (4/1)/Div (12/12)/Sqrt (24/24)
Phys. registers	128 combined integer + floating-point
L1 I-cache	64 KB, 1-way set-associative, 128 byte line, 1 cycles hit time
L1 D-cache	64 KB, 4-way set-associative, 64 byte line, 2 cycles hit time
L2 cache unified	2 MB, 8-way set-associative, 128 byte line, 6 cycles hit time
BTB	2048 entry, 2-way set-associative
Branch predictor	4 K entry Gshare, 10 bit global history
Branch mispred. pen.	8 cycles minimum
Memory	128 bit wide, 150 cycles first chunk, 1 cycles interchunk
TLB	32 entry (I), 128 entry (D), fully associative

rescheduling. Therefore, simply releasing the entry half when the comparator match occurs (and the corresponding valid bit is set) is not sufficient. Instead, the entry half should be freed D cycles after the register becomes speculatively ready, where D is the number of pipeline stages between issue and execution. At this time, the corresponding source is nonspeculatively (or deterministically) ready. To implement such a delayed release of the IQ entry halves, each IQ entry half is augmented with a counter that is initialized to D . This counter is decremented every cycle after the corresponding valid bit has been set and, when it falls to zero, the IQ entry half can be released. Therefore, to support the load-related replays where instructions must be reissued from the IQ, the aggressive packing mechanism requires two such counters per IQ entry (one for each IQ entry half). This overhead is small as long as the issue to execute pipeline contains a small number of stages. For example, a two-stage issue to execute pipeline requires two-bit counters for each IQ half—amounting to four bits per IQ entry. The overhead of these additional bits is accounted for in the power and energy calculations presented in the results section. Notice that these counters can also be implemented through shift registers, especially when the schedule-to-execute pipeline is shallow.

5. SIMULATION METHODOLOGY

For estimating the performance impact of the schemes described in this paper, we used a significantly modified version of the SimpleScalar 3.0d simulator [Burger and Austin 1997]. The simulator has been modified to separately and accurately model pipeline structures, such as the issue queue, reorder buffer, and physical register file. To evaluate the proposed mechanism, two processor configurations were examined. The first configuration, presented in Table I, is a four-way superscalar processor and the second, presented in Table II, is an eight-way superscalar processor with larger ROB.

We simulated the full set of 26 SPEC 2000 integer and floating point benchmarks [Sharkey and Ponomarev 2005b], using the precompiled Alpha binaries available from the SimpleScalar website [Burger and Austin 1997]. We skipped the initialization part of each benchmark using the procedure prescribed by the

Table II. Configuration of the 8-way Simulated Processor

Parameter	Configuration
Machine width	8-wide fetch, 8-wide Issue, 8-wide commit
Window size	issue queue—as specified, 128 entry load/store queue, 256-entry ROB
Function units and latency (total/issue)	8 Int Add (1/1), 4 Int Mult (3/1)/Div (20/19), 4 Load/Store (2/1), 4 FP Add (2), 4 FP Mult (4/1)/Div (12/12)/Sqrt (24/24)
Phys. registers	300 combined integer + floating-point
L1 I-cache	64 KB, 1-way set-associative, 128 byte line, 1 cycles hit time
L1 D-cache	64 KB, 4-way set-associative, 64 byte line, 2 cycles hit time
L2 cache unified	2 MB, 8-way set-associative, 128 byte line, 6 cycles hit time
BTB	2048 entry, 2-way set-associative
Branch predictor	4 K entry Gshare, 10 bit global history
Branch mispred. pen.	8 cycles minimum
Memory	128 bit wide, 150 cycles first chunk, 1 cycles interchunk
TLB	32 entry (I), 128 entry (D), fully associative

Simpoints tool [Sherwood et al. 2002] and then simulated the execution of the following 100M instructions.

For estimating the delay, energy, and area requirements, we designed the actual VLSI layouts of the IQ and simulated them using SPICE. The layouts were created in a 0.18 μm 6-metal layer CMOS process (TSMC) using Cadence design tools. A V_{dd} of 1.8 V was assumed for all the measurements.

6. RESULTS

In this section, we first demonstrate that the use of nonspeculatively set register ready bits (as described in Section 3) to support load-latency related replays with instruction-packing does not reduce the effectiveness of the instruction-packing. To illustrate this point, we performed the set of experiments with the *perfect* load latency predictor to make it possible to implement instruction-packing based on the speculatively set ready bits. As no load-related replays are introduced in such an environment, instruction-packing can still be used with speculatively set bits and we can compare the resulting performance differences. We investigate the impact of real predictors later on in this section.

With perfect load-latency predictor, we implemented the instruction packing mechanism on both the four- and eight-way machines for various numbers of cycles between the issue and execution stages with a 32-entry instruction packing scheduler (with the maximum capacity to hold 64 instructions). While no replays occur in such a design, we still model the additional pressure put on the queue by the instructions that are issued, but have not been verified to be replay free. For all these configurations, we compared the use of speculative versus nonspeculative register ready bits for instruction packing. Figure 3 presents the average results across all benchmarks for the simulated configurations. Results are limited to the averages due to the space constraints. The legend of the graph should be understood in the following way. Each configuration is marked as X - Y , where X denotes the use of the speculative or nonspeculative bits for instruction packing and Y denotes the number of cycles between issue and execution. For example, the configuration *Nonspec-2cycles* refers to a datapath where the nonspeculative register ready bits are checked at dispatch and there

Fig. 3. Commit IPCs of the eight and four-way machines for various numbers of cycles between issue and execution.

Fig. 4. Commit IPCs of the simulated machines with eight-cycles between issue and execution for the 32-entry instruction-packing scheduler using the speculative and nonspeculative register-ready bits.

are two pipeline stages between issue and execution. As seen from this graph, the differences in performances are very small in all other examined configurations as well. In fact, the eight-way machine with eight cycle issue-to-execute delay has the largest performance loss—1.4%, on average.

Figure 4 presents the detailed per-benchmark commit IPCs for this particular configuration. As seen from the results, the performance differences are very small, even with the large number of issue-to-execute cycles assumed in these experiments. For example, the average difference is only 0.2% for the four-way machine and 1.2% for the eight-way machine. For the eight-way machine, the largest performance difference is 4.6% for *galgel* and 16 of the benchmarks have differences of less than 1%.

Next, we examine how these different configurations and the use of speculative versus nonspeculative ready bits effect the percentage of ready source operands in the dispatched instructions. Figure 5 presents the percentage of instructions entering the queue with two nonready source operands (these instructions really require full sized IQ entries with instruction-packing). The difference in this percentages among the cases that use speculative or nonspeculative ready bits is generally very small (a few percentage points) except for the case where the issue to execution latency is as large as eight cycles. As shown in Figure 4, even in that case the IPC impact is minimal. This is

Fig. 5. Percentage of instructions dispatched with two nonready source operands.

Fig. 6. Percentage of instructions dispatched with two nonready source operands.

because that despite the fact that the use of nonspeculative bits for instruction-packing increases the percentage from 12% to 21% in the worst case, the absolute percentages are still low and most of the dispatched instructions are still capable of making use of a large fraction of the IQ in the packing mode. This, coupled with the fact that the IPCs do not significantly increase in the base case as the IQ size doubles as we go beyond 32-entries, explains the negligible IPC impact seen in Figures 3 and 4. Specifically, if the IQ size is increased from 32 to 64 entries in the baseline case, then the overall IPC increase (with all other resources unchanged) is slightly more than 7%, on average, across benchmarks.

As another supporting statistic, Figure 6 presents the number of cycles that the instructions that enter the scheduling window with two nonready source operands spend in the IQ. Also shown is the number of cycles until the arrival of the first source operand. On average for the four-way machine, such instructions spend 28 cycles in the queue and the major fraction of these cycles (25 cycles) is spent waiting of the arrival of the first source. Once that source arrives, the arrival of the second source and the consecutive instruction issue happens very quickly, typically with a few cycles. These statistics explain why the percentages presented in Figure 6 do not change very much if the ready bits used by instruction packing are set a few cycles earlier or later. In other words, if an instruction has two nonready sources based on the speculative versions of ready bits it is very likely that the same will be true when the nonspeculative bits are checked.

Fig. 7. Pipeline diagram of the simulated processor.

Having presented the evaluations with the perfect load-latency prediction, we now focus on a realistic datapath and evaluate the performance of instruction-packing on the following configuration. We assume two pipeline stages between issue and execution, use a 4K-entry gShare style load-latency predictor (which gives higher than 98% average accuracy), and model the squash-recovery mechanism (as used in the Alpha 21264 [Compaq Computer Corp. 1999]) to support replays following load-latency mispredictions. In the squash-recovery model, instructions remain in the IQ after issue until they begin execution (and are no longer vulnerable to replay traps). Following the discovery of a load-latency misprediction, all instructions between the issue and execution pipeline stages are squashed and must be reissued. For more details of the recovery model as well as the logic needed to correctly reset the contents of the register status bits stored in the IQ entries, we refer to [Kim and Lipasti 2004]. Figure 7 presents the pipeline diagram assumed throughout the rest of the simulations.

The rest of this section presents the performance, power, and delay analysis of instruction-packing, assuming the configuration described in the previous paragraph and the pipeline structure shown in Figure 7. Our goal is to show that the use of instruction-packing can reduce the number of entries in the IQ by a factor of two with very little IPC degradation and provide substantial reductions in the wakeup power and delays. In all subsequent discussions, an N -entry packing queue refers to a queue that has N entries with the maximum capacity to hold $N*2$ instructions. So, for the IPC purposes, a fair point of comparison is an N -entry packing queue and a $2*N$ -entry traditional queue.

Figure 8 shows the commit IPC speedup over the 16-entry traditional case for the simulated four-way microarchitecture using four different scheduling schemes: the baseline scheduler (labeled “traditional”), instruction-packing (labeled “packing”), instruction-packing implementing the early release of comparators “on-the-fly” (labeled “packing-ER”), and the scheduler that uses tag elimination (TE) design of Ernst and Austin [2002] (labeled “TE”), which statically partitions the IQ providing some entries with two comparators, others with one comparator, and yet others with 0 comparators. The TE bar is only presented for comparison purposes—detailed qualitative comparison between instruction-packing and the TE scheme is given in Section 7. The baseline scheduler is presented for three sizes: 16, 32, and 64 entry. The scheduler implementing instruction-packing is presented for two sizes: 16 and 32 entry. The TE design uses the configurations 6/10/16 (6 two-comparator entries, 10 single-comparator entries, and 16 entries with no comparators) and 12/20/32 (12 two-comparator entries, 20 single-comparator entries, and 32 entries with no comparators). This roughly corresponds to the distribution of operands with 0, 1, and 2 nonready sources, as observed in Figure 1.

Fig. 8. Commit IPC speedup of the simulated four-way machine for various schedulers on (a) integer and (b) floating-point benchmarks.

The 16-entry scheduler with instruction-packing performs within 2.5% of a traditional 32-entry IQ, on average. This result is expected because most instructions fit in a half-entry in the queue utilizing packing, effectively doubling its capacity. The static approach of the TE-6/10/16 scheduler results in a 4.0% IPC degradation, on average. Therefore, the dynamic approach of instruction-packing significantly reduces the IPC loss incurred by the static approach of TE. Likewise, the 32-entry scheduler with instruction-packing performs within 0.6% of a traditional 64-entry IQ, on average, and the TE-12/20/32 scheduler shows 1.3% IPC degradation, on average.

Furthermore, the IPC impact of instruction-packing can be minimized by the mechanism presented in Section 6 to release comparators early on-the-fly for instructions allocated a full-width entry. When such instructions are present in the queue and one of their two register source operands becomes ready, the half of the queue entry corresponding to that source is released early. This allows for a new instruction requiring a half-width entry to enter the IQ and reside in this location. When this mechanism is used, the performance degradations drop to 1.6% and 0.3% for 16-entry and 32-entry queues, respectively. Thus, the early-release mechanism significantly reduces the IPC difference between the scheduler using instruction packing and the scheduler of twice its size (specifically, 37% reduction for the 16-entry queue and 48% reduction for the 32-entry queue).

Fig. 9. Commit IPC speedup for the simulated eight-way machine with various schedulers for (a) integer and (b) floating-point benchmarks.

Similar results are observed on the simulated eight-way machine, as presented in Figure 9. The 16-entry queue implementing instruction-packing obtains performance within 4.7% of the 32-entry traditional queue. With the addition of the early release mechanism to the instruction-packing, the performance difference drops to 2.7%: a 43% reduction in IPC loss. This is as compared to the TE-6/10/16 queue, which reduces performance by 10.1% as compared to the 32-entry traditional queue in this case. Here, the TE scheduler again results in significantly more IPC loss than the instruction-packing.

The 32-entry instruction-packing scheduler on the eight-way machine performs within 3.9% of the traditional scheduler of twice its size and within 2.4% of the 64-entry IQ when using the early release mechanism. A loss of 5.9% is experienced by the TE-12/20/32 scheduler in this case, again because of its static nature.

CMOS layouts of both the 64-entry traditional queue and the 32-entry packing queue show a 26.7% reduction in the IQ area because of the use of instruction-packing. Packing effectively reduces the number of CAM bitcells by one-half, while increasing the number of SRAM bitcells in each row (but leaving the total number of SRAM bitcells in the IQ practically unchanged). As observed in our layouts, CAM bitcells are 41% larger than their SRAM counterparts. It is the reduction in the number of CAM bitcells that accounts for the IQ area reduction.

As presented in Table III, instruction-packing achieves a 26.4% reduction in the wakeup delay. This delay reduction comes mainly from the shorter and

Table III. Wakeup Delays of a 32-Entry Queue Supporting Instruction Packing Compared to a 64-Entry Traditional Queue

	Tag-Bus Drive (ps)	Comparator Output (ps)	Final Match Signal (ps)	Total Delay (ps)
64-entry	421	236	141	798
32-entry packing	236	220	131	587
Savings:	43.9%	6.8%	7.1%	26.4%

Fig. 10. Average power savings of basic and aggressive instruction-packing on the four-way simulated processor for integer and floating-point benchmarks.

lower-capacitance tag busses and bitlines. Combined with the 5% increase in the delay of the selection logic (as conservatively estimated at the end of Section 2.3), this results in an overall reduction of 14% in the overall scheduler (wakeup/selection) delay.

Next, we examine the implications of instruction-packing on dynamic power dissipation. Instruction packing saves power because of the smaller number of tag comparators and shorter tag-busses and bitlines. Figure 10 presents the power savings of both the 32-entry basic packing queue and the 32-entry aggressive packing queue (from Section 4) on the simulated four-way superscalar processor, as extracted from the SPICE simulations. The bars represent the savings in the wakeup power, dispatch power, issue power, and the total IQ power. The power of the selection logic was not computed, but it generally represents a small percentage of the overall IQ power and does not change with the instruction-packing. The basic instruction-packing saves 40.2% of total IQ power as compared to a traditional 64-entry queue, on average, across all 26 SPEC 2000 benchmarks.

As discussed in Section 4, aggressive packing incurs additional power consumption because of the need to replicate the payload in both the left half and the right half payload areas for all instructions allocated to a full-width entry. The additional power consumption is also experienced in changing the mode bit during this conversion and clearing the corresponding allocated bit for the half of the entry being freed. While all of this additional power consumption is not negligible, it has a relatively small impact on the overall power of the queue, because it impacts only by a small percentage of instructions:

Fig. 11. Average reduction in the energy-delay-squared product (ED^2P) of basic and aggressive instruction-packing on the four-way simulated processor for integer and floating point benchmarks.

specifically those that are allocated a full-width entry. This amounts to only a small fraction of instructions, on average. As presented in Figure 10, the scheduler implementing instruction-packing and the “on-the-fly” conversion mechanism provides a 38.5% power reduction compared to the traditional queue of twice its size. Notice that this is slightly lower than the 40.2% power savings of the instruction-packing mechanism without the “on-the-fly” conversion mechanism, but the difference is somewhat small (1.7%). Therefore, this mechanism provides a means for significantly reducing the IPC losses incurred by instruction-packing (by nearly 50%), while reducing the power savings by only 2%.

Note that in all power measurements, the dynamic dissipations of the additional logic within the scheduler have been accounted for. While the explicit quantitative evaluation of the impact of the instruction-packing mechanism on static power dissipation is beyond the scope of this paper, it is likely that the static power will also reduce, because the overall number of bitcells (including both RAM and CAM cells) is smaller with instruction packing. In addition, as the dynamic power is reduced (as demonstrated in the paper), it is likely that the operating temperature will also decrease, further reducing static power.

Finally, Figure 11 shows the reduction in the energy delay-squared product (ED^2P) for both instruction packing and aggressive instruction packing with respect to the baseline machine with the 64-entry issue queue on the four-way machine. For these analyses, we assumed that the overall scheduling power, including wakeup and selection, is 20% of the total chip power. On the average, across all benchmarks, instruction-packing results in 6% improvement in the ED^2P . The aggressive-packing mechanism results in an ED^2P improvement of 5.6%, which is slightly less than that of the basic instruction-packing. This is because the IPC performance gains achieved by aggressive packing are off-set by the power overheads of the scheme, resulting in reduced energy efficiency compared to the basic packing. Therefore, the basic instruction packing provides the best energy-efficiency tradeoffs while the aggressive packing provides higher

performance at the expense of some energy efficiency (0.4% ED²P). Contrary to the average trends, for some benchmarks, such as *ammp*, *apsi*, *equake*, *six-track*, *twolf*, and *wupwise*, the energy-efficiency of aggressive packing is higher than that of the baseline packing. In any case, significant improvements in the overall energy efficiency of dynamic scheduling logic are realized using the techniques described in this paper.

7. RELATED WORK

Researchers have proposed several ways to reduce the power consumption of the issue logic. Dynamic adaptation techniques [Buyuktosunoglu et al. 2001, 2003; Folegnani and Gonzalez 2001; Ponomarev et al. 2001] partition the queue into multiple segments and deactivate some segments periodically, when the applications do not require the full IQ to sustain the commit IPCs. Energy-efficient comparators, which dissipate energy predominantly on a tag match, were proposed in [Ponomarev et al. 2003, 2004]. Also in [Ponomarev et al. 2003], the IQ power was reduced by using zero-byte encoding and bitline segmentation. In [Huang et al. 2002], the associative broadcast is replaced with indexing to only enable a single instruction to wakeup. This exploits the observation that many instructions have only one consumer. In [Aggarwal et al. 2004], the wakeup width is decreased to be smaller than the machine width, noting that the full machine width is rarely used for instruction wakeup.

The observation that many instructions are dispatched with at least one of their source operands ready is not new—it was used in Ernst and Austin [2002], where the scheduler design with reduced number of comparators was proposed. In that scheme, some IQ entries have two comparators, others have just one comparator, and yet others have zero comparators. The IPC losses coming from statically partitioning the queue are somewhat higher than in the instruction-packing, which dynamically adapts the queue to any particular instruction mix. We presented a detailed performance comparison with the scheme of Ernst and Austin [2002] (without last-tag speculation mechanism) in Figures 3 and 4 and discussed the results in Section 6. The last-tag speculation mechanism introduced in Ernst and Austin [2002] can further reduce the number of required comparators, but it requires the extra logic to maintain the predictions and also handle possible mispredictions.

In Kim and Lipasti [2003a], the tag buses were categorized into fast buses and slow buses, such that the tag broadcast on the slow bus takes one additional cycle. The design again relied on the last-arriving operand prediction to hook the last arriving operand (which actually identifies when the instruction wakes up) to the fast bus to avoid the wakeup delays. The approach [Kim and Lipasti 2003a] does not reduce or share the number of IQ entries and the length of tag buses—it just decouples one-half of the tag buses from the fast wakeup bus. While we did not perform direct comparison with the work of Kim and Lipasti [2003a] in terms of IPCs, according to the results presented in their paper, the average IPC loss was 2.2%, which is higher than what is achieved with instruction-packing. Power reduction was also not the goal of the work of Kim and Lipasti [2003a].

One approach to reducing scheduling complexity involves pipelining the scheduling logic into separate wakeup and select cycles [Stark et al. 2000; Kim and Lipasti 2003b]. Stark et al. [2000] uses the status of an instruction's grandparents to wakeup the instruction earlier in a speculative manner. [Kim and Lipasti 2003b] proposed grouping of two (or more) dependent single-cycle operations into so-called Macro-OP (MOP), which represents an atomic scheduling entity with multicycle execution latency. The concept of dataflow minigraphs [Bracy et al. 2004] is similar to Macro-Op scheduling in that groups of instructions are scheduled together.

Other proposals have introduced new scheduling techniques with the goal of designing scalable dynamic schedulers [Brekelbaum et al. 2002; Lebeck et al. 2002; Cristal et al. 2004; Raasch et al. 2002; Srinivasan et al. 2004; Sharkey and Ponomarev 2005b]. Brown et al. [2001] proposed to remove the selection logic from the critical path by exploiting the fact that the number of ready instructions in a given cycle is typically smaller than the processor's issue width.

Scheduling techniques based on predicting the issue cycle of an instruction [Ernst et al. 2003b; Hu et al. 2004; Canal and Gonzalez 2000, 2001; Abella and Gonzalez 2004; Michaud et al. 2001; Liu 2004] remove the wakeup delay from the critical path and remove the CAM logic from instruction wakeup, but need to keep track of the cycle when each physical register will become ready. In Ehrhart and Patel [2004], the wakeup time prediction occurs in parallel with the instruction fetching. In Sharkey and Ponomarev [2005a], a wakeup-free scheduler without counting and issue time-estimation logic is proposed.

8. CONCLUDING REMARKS

We proposed Instruction Packing—a novel microarchitectural technique that reduces both the delay and the power consumption of the IQ by sharing the associative part of an IQ entry between two instructions, each with, at most, one nonready source. Consequently, the number of IQ entries, and thus the length of and the capacitive loading on the tag buses and bitlines, can be reduced substantially, leading to faster access and lower power dissipation. In addition, the layout area of the IQ is also reduced.

The use of instruction-packing on the scheduler with the capacity to hold up to 64 instructions results in a 40.2% reduction in the total IQ power. In addition, the wakeup delay is reduced by 26.4%. Combined with the 5% increase in the delay of the selection logic, this results in a 14% reduction in the overall scheduler (wakeup/selection) delay. All of this comes at the cost of only 0.6% IPC degradation on the four-way machine and 3.9% for the eight-way machine on the average across all SPEC 2000 benchmarks.

The IPC degradation incurred by instruction-packing (however small it is) can be further reduced by the use of aggressive packing, where the IQ entry halves are released “on-the-fly” as the source operands become ready. However, because of the additional energy dissipations incurred in this scheme, the energy-effectiveness of this method is actually lower than that of the basic instruction-packing mechanism, if energy-delay squared product is used as a metric of power/performance efficiency. However, both of these design are

significantly more energy efficient than the baseline machine—the improvements in the ED²P are 6.0% and 5.6%, respectively.

REFERENCES

- ABELLA, J. AND GONZALEZ, A. 2004. Low-complexity distributed issue queue. In *Proceedings of the International Symposium on High Performance Computer Architecture*.
- AGGARWAL, A. ET AL. 2004. Defining wakeup width for efficient dynamic scheduling. In *Proceedings of the International Conference on Computer Design*.
- BRACY, A. ET AL. 2004. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *Proceedings of the International Symposium on Microarchitecture*.
- BROWN, M., STARK, J., AND PATT, Y. 2001. Select-free instruction scheduling logic. In *Proceedings of the International Symposium on Microarchitecture*.
- BREKELBAUM, E. 2002. Hierarchical scheduling windows. In *Proceedings of the International Symposium on Microarchitecture*.
- BURGER, D. AND AUSTIN, T. M. 1997. The simpleScalar tool set: Version 2.0. *Technical Report, Department of CS, Univ. of Wisconsin-Madison*, June and documentation for all Simple Scalar releases.
- BUYUKTOSUNOGLU, A. ET AL. 2001. A circuit-level implementation of an adaptive issue queue for power-aware microprocessors. In *Proceedings of the Great Lakes Symposium on VLSI Design*.
- BUYUKTOSUNOGLU, A. ET AL. 2003. Energy-efficient co-adaptive instruction fetch and issue. In *Proceedings of the International Symposium on Computer Architecture*.
- CANAL, R. AND GONZALEZ, A. 2000. A low-complexity issue logic. In *Proceedings of the International Conference on Supercomputing*.
- CANAL, R. AND GONZALEZ, A. 2001. Reducing the complexity of the issue logic. In *Proceedings of the International Conference on Supercomputing*.
- Compaq Computer Cor. 1999. Alpha. 21264 Microprocessor hardware reference manual. July.
- CRISTAL, A. ET AL. 2004. Out-of-order commit processors. In *Proceedings of the International Symposium on High Performance Computer Architecture*.
- EHRHART, T. AND PATEL, S. 2004. Reducing the scheduling critical cycle using wakeup prediction. In *Proceedings of the International Symposium on High Performance Computer Architecture*.
- ERNST, D. AND AUSTIN, T. 2002. Efficient dynamic scheduling through tag elimination. In *Proceedings of the International Symposium on Computer Architecture*.
- ERNST, D. AND AUSTIN, T. 2003a. Practical selective replay for reduced-tag schedulers. In *Proceedings of the 2nd Workshop on Duplicating, Deconstructing, and Debunking*, June.
- ERNST, D., HAMEL, A., AND AUSTIN, T. 2003b. Cyclone: a broadcast-free dynamic instruction scheduler with selective replay. In *Proceedings of the International Symposium On Computer Architecture*.
- FOLEGNANI, D. AND GONZALEZ, A. 2001. Energy-effective issue logic. In *Proceedings of the International Symposium on Computer Architecture*.
- HU, J., VIJAYKRISHNAN, N. AND IRWIN, M. 2004. Exploring wakeup-free instruction scheduling. In *Proceedings of the International Symposium on High Performance Computer Architecture*.
- HUANG, A. ET AL. 2002. Energy-efficient hybrid wakeup logic. In *Proceedings of the International Symposium on Low-Power Electronics and Design*.
- GOWAN, M., BIRO, L., AND JACKSON, D. 1998. Power considerations in the design of the alpha 21264 microprocessor. In *Proceedings of the 35th ACM/IEEE Design Automation Conference*.
- KIM, I. AND LIPASTI, M. 2003a. Half-price architecture. In *Proceedings of the International Symposium on Computer Architecture*.
- KIM, I. AND LIPASTI, M. 2003b. Macro-op scheduling: Relaxing scheduling loop constraints. In *Proceedings of the International Symposium on Microarchitecture*.
- KIM, I. AND LIPASTI, M. 2004. Understanding scheduling replay schemes. In *Proceedings of the International Symposium on High Performance Computer Architecture*.
- LEBECK, A. ET AL. 2002. A large, fast instruction window for tolerating cache misses. In *Proceedings of the International Symposium on Computer Architecture*.
- LIU, A. 2004. Scaling the issue window with look-ahead latency prediction. In *Proceedings of the International Conference on Supercomputing*.

- MICHAUD, A. ET AL. 2001. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings of the International Symposium on High Performance Computer Architecture*.
- MERCHANT, A. AND SAGER, D. 2001. US patent #6,212,626, assigned to Intel Corp., Computer processor having a checker. April.
- PALACHARLA, S., JOUPPI, N., AND SMITH, J. 1997. Complexity-effective superscalar processors. In *Proceedings of the International Symposium on Computer Architecture*.
- PONOMAREV, D., KUCUK, G., AND GHOSE, K. 2001. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the International Symposium on Microarchitecture*.
- PONOMAREV, D. ET AL. 2003. Energy-efficient issue queue design. In *IEEE Transactions on VLSI Systems*, November.
- PONOMAREV, D. 2004. Energy-efficient comparators for superscalar datapaths. In *IEEE Transactions on Computers*, July.
- RAASCH, S., BINKERT, N., AND REINHARDT, S. 2002. A scalable instruction queue design using dependence chains. In *Proceedings of the International Symposium on Computer Architecture*.
- SHARKEY, J. AND PONOMAREV, D. 2005a. Instruction recirculation: Eliminating counting logic in wakeup-free schedulers. In *Proceedings of the Euro-Par Conference*.
- SHARKEY, J. AND PONOMAREV, D. 2005b. Non-uniform instruction scheduling. In *Proceedings of the Euro-Par Conference*.
- SHARKEY, J., PONOMAREV, D., GHOSE, K., AND ERGIN, O. 2005. Instruction packing: Reducing power and delay of the dynamic scheduling logic. In *Proceedings of the International Symposium on Low Power Electronics and Design*.
- SHERWOOD, A. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.
- SRINIVASAN, S. ET AL. 2004. Continual flow pipelines. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.
- STARK, J., BROWN, M., AND PATT, Y. 2000. On pipelining dynamic instruction scheduling logic. In *Proceedings of the International Symposium on Microarchitecture*.
- WILCOX, K. ET AL. 1999. Alpha processors: A history of power issues and a look to the future. In *Cool-Chips Tutorial*, November.

Received August 2005; revised November 2005, February 2006; accepted February 2006