

Instruction Wake-Up in Wide Issue Superscalars

Soner Önder

Dept. of Computer Science
Michigan Technological University
Houghton, MI 49931-1295
soner@mtu.edu

Rajiv Gupta

Dept. of Computer Science
University of Arizona
Tucson, AZ 85721
gupta@cs.arizona.edu

Abstract

While the central window implementation in a superscalar processor is an effective approach to waking up ready instructions, this implementation does not scale to large instruction window sizes as those that are required by wide issue superscalars of the future. We propose a new wake-up algorithm that dynamically associates explicit wake-up lists with executing instructions according to the dependences between instructions. Instead of repeatedly examining a waiting instruction for wake-up till it can be issued, this algorithm identifies and considers for wake-up a fresh subset of waiting instructions from the instruction window in each cycle. This subset of instructions are the ones present in the wake-up lists of completing instructions. The direct wake-up microarchitecture (DWMA) that we present is able to achieve approximately 80%, 75% and 63% of the performance of a central window processor at high issue widths of 8, 16 and 32 respectively when an effective memory disambiguation mechanism [13] is employed for load speculation. In contrast, the wake-up algorithm proposed in constructing the dependence-based microarchitecture (DBMA) [15] can only achieve approximately 62%, 46% and 45% of the performance of a central window processor at issue widths of 8, 16 and 32. Thus the DWMA processor greatly outperforms the DBMA processor.

Keywords: Wide-issue superscalar, instruction window, instruction wake-up, direct matching, memory disambiguation, speculative execution.

1 Introduction

Recent trend in the design of high performance superscalar processors is to provide aggressive means for out-of-order execution of instructions. Out-of-order execution allows the processors to take advantage of instruction level parallelism extracted from an instruction stream. The above trend is motivated in part due to the demonstration by many researchers that ordinary programs have significant degrees of instruction level parallelism that can be potentially extracted by the hardware [1, 4, 7, 12, 16, 20]. However, the potential for speedups supported by the observed levels of instruction level parallelism remain far from being fully realized. In order to uncover and exploit high levels of parallelism, a number of important problems must be addressed. In particular inadequate solutions to each of the following problems act as bottlenecks in current processor designs:

1. In order to issue multiple instructions in each cycle, it is essential that the processor possess the capability of fetching a large number of instructions every cycle. This task is made difficult by the presence of frequent branches which disrupt the instruction stream.
2. In order to achieve high performance, load instructions should be held until such time that their issue will not cause memory dependence violations but not held any longer than necessary [10]. In other words it is essential that an effective memory disambiguation mechanism be provided.
3. To uncover high degree of instruction level parallelism, a large number of instructions must be continuously examined for ready instructions. In other words a large instruction window is needed from which ready instructions may be found to sustain a steady flow of instructions to the functional units for execution. In fact, the required window size increases quadratically with increasing issue width [12].

In addressing the first problem, the trace cache mechanism has yielded promising results [17, 3]. Although there is a room for improvement in this area, the trace cache approach is a significant step towards eliminating the first bottleneck. Recent innovations in the area of memory disambiguation are Moshovos and Sohi's work [9] and the development of a simple and efficient speculative memory disambiguator by Chrysos and Emer [2]. Given a high bandwidth instruction fetcher and an effective memory disambiguator, a speculative superscalar processor can uncover significant amounts of instruction level parallelism over a large instruction window.

Implementation of a large instruction window in a superscalar processor, without slowing down the processor clock, requires better techniques for identifying ready instructions than what are available today. This is because, fetching and buffering a large number of instructions in a processor is by itself not sufficient to derive the benefits of a large instruction window. The waiting instructions must be woken up at the earliest possible time that they become ready to sustain a high degree of instruction level parallelism. In other words, implementation of a large instruction window is made difficult by the need for fanning out the wake-up signal to waiting instructions and selecting for issuing the instructions which are ready. When we assume that instructions are woken up by means of broadcasting, as in a central window implementation, we are faced with significant delays which originate from wire delays, tag matching time as well as the associative logic necessary to implement the wake-up functionality. These delays increase significantly for high issue widths required to exploit high degrees of instruction level parallelism. This is because, the delay of the wake-up logic of an instruction window is a function of the window size and the issue width. These delays increase quadratically for most building blocks of the instruction window [14].

In this paper we propose a novel wake-up algorithm that dynamically associates wake-up lists with executing instructions. The wake-up list of an instruction identifies a small number of instructions that require for their execution an operand used and/or the result computed by the instruction. We present the design of a microarchitecture, the direct wake-up microarchitecture (DWMA) that implements a wake-up algorithm based upon dynamic construction of wake-up lists. In this

algorithm, at each cycle, primarily the instructions belonging to the wake-up lists of instructions that have just completed execution are considered for wake-up as only these instructions could now have become ready for execution.

A number of alternative architectures have been considered to address the wake-up problem [19, 18, 21, 6, 15]. However, only the approach taken in the design of the dependence based microarchitecture (DBMA) [15] proposes a solution to the wake-up problem in the context of a conventional superscalar architecture that is of reasonable complexity in comparison to a central window processor. We have implemented a simulator for our architecture and our results show that the DWMA out performs the DBMA architecture due to its flexibility. DWMA captures 80 %, 75 % and 63 % of the performance achievable by a central window processor with the out of order store set memory disambiguator at issue widths of 8, 16 and 32, respectively.

The remainder of the paper is organized as follows. In section 2, we present the concept of direct wake-up which is central to our novel approach to instruction wake-up. In section 3, we present the details of our proposed DWMA microarchitecture. In section 4, we thoroughly evaluate our technique and demonstrate that it achieves a significant percentage of the performance of an ideal central window processor. Finally, we summarize our results in section 5.

2 Direct Wake-up

The wake-up algorithm proposed by Palacharla et al. in [15] is based upon an important observation that if a set of instructions form a dependence chain, then the wake-up mechanism only needs to examine the first instruction in the chain since the other instructions can never be successfully woken before the first instruction. It feeds dependence chains into FIFOs and only the instructions at the heads of the FIFOs are examined for wake-up thereby reducing the complexity of wake-up. To uncover high degree of instruction level parallelism, as is needed for wide issue processors, we need to examine an increasingly larger number of instructions [12]. The window of instructions over which parallelism can be detected by the DBMA mechanism can be sometimes limited because the instruction steering algorithm requires additional empty FIFOs to proceed, and when none are available, it must stall till some FIFOs become empty. While the use of dependence information in guiding the wake-up is an effective approach, the use of FIFOs for scheduling prevents DBMA from achieving high levels of performance. We therefore propose the following approach.

We provide a large instruction window from which instruction level parallelism is extracted by examining a subset of instructions for wake-up at each cycle. However, unlike the DBMA, instead of continuously examining a waiting instruction at the head of a FIFO till it can be issued, we determine a fresh set of instructions that will be examined by the wake-up mechanism in each cycle. The dependence information is used to identify this set of instructions. If in the current cycle an instruction is expected to complete its execution, then another instruction waiting for the result of the completed instruction may become ready in the next cycle. Thus, such an instruction will be among the set of instructions examined for wake-up and if ready, it would be issued for execution in the next cycle.

In order to guide the above wake-up heuristic we must construct and maintain a *wake-up* graph dynamically. Since the graph is to be maintained in hardware, certain constraints must be placed on its form. In particular, the number of edges emanating from a node cannot be unbounded. In the hardware implementation the wake-up graph is represented by associating with each instruction an explicit wake-up list, W-LIST, which identifies the instructions that will be woken up by that instruction (i.e., the wake-up list is the representation of edges in the graph). Ideally in the W-LIST of an instruction we would like to place all instructions which use its result (i.e., all instructions to which there is a def-use edge). But since the number of entries in the W-LIST must be a fixed small number in a practical implementation, we cannot store all such instructions in the W-LIST. The approach we take to deal with this problem is as follows. We allow an instruction that computes a result to directly wake-up only the first two instructions in the instruction stream that use the result while the others are woken up indirectly. The responsibility of waking up future instructions in the instruction stream is delegated to these first two instructions. To carry out the wake-up of further instructions we introduce use-use edges from the first

two instructions and the next four instructions. This process of introducing use-use edges is repeated till all instructions have been handled. Thus, the wake-up graph contains both def-use and use-use edges. In summary an instruction can wake-up instructions connected through both def-use and use-use edges. There are at most six edges in the W-LIST of an instruction, namely, (d0,d1), (l0,l1) and (r0,r1) corresponding to the result, the left operand and the right operand.

A crucial aspect of our algorithm is that the instructions that are linked through a chain of use-use edges may execute in an order different from the order they appear in the chain. Consider a chain of use-use edges that has been created due to an operand value v . When the value v is computed by its producer, the first instruction in the chain is considered for wake-up. Let us assume that the instruction cannot issue because its other operand is unavailable. In this situation, since the value v is available, we will consider the next instruction in the chain for wake-up even though the previous instruction is still waiting. It is possible that this instruction is found to be ready and thus may execute prior to the preceding instruction in the chain. Thus, the introduction of use-use edges does not restrict out-of-order execution. Its only consequence is that all instructions waiting for the same operand are not examined simultaneously by the wake-up algorithm. Rather in each cycle only a subset of instructions waiting for the value are examined.

Clearly the wake-up graph that is constructed dynamically is influenced by the order in which the instructions appear in the code sequence. In addition, the form of the graph is also influenced by the timing of instruction execution. In fact while the initial form of the wake-up graph is determined by the original instruction ordering, the graph undergoes transformations determined by execution timing. Let us consider the modifications to the graph that may take place. When an instruction is first added to the wake-up graph, it is linked to the producer of its first unavailable operand, that is, it is added to the W-LIST of the producer of this operand. Now let us assume that this operand becomes available. This will cause the instruction waiting for the value to be considered for wake-up. At this point it may be determined that the second operand of the instruction is also unavailable. In this case the instruction would be added to the W-LIST of the producer of this missing operand. In other words, while one def-use edge has been removed, another one has been added to the wake-up graph.

Finally we would like to make the observation that an instruction may be woken up 0, 1 or 2 times before it can execute. If the instruction has all of its operands available when it is first fetched, it is issued immediately and thus never woken up. If the instruction has only one operand missing when it is first fetched, then it will be woken up once when that missing operand becomes available (either immediately using a def-use edge or in a delayed fashion through a use-use edge). If both of the operands of an instruction are missing when it is fetched then it may be woken up twice, once when the first operand becomes available, and then again when the second operand becomes available.

Now let us consider the example code sequence shown in Figure 1(a). The dependence graph for the code sequence is illustrated in Figure 1(b) and execution of this code sequence on a central window processor ((CW)) with 2 functional units is shown in Figure 1(e). We observe that using our approach the wake-up links for all the instructions except OR and SUB can be established using def-use edges, that is using d0 and d1. This is because the ADD instruction has four children and therefore two result links are not sufficient. We therefore associate XOR and SLL, the first two instructions, directly with the result of ADD through def-use edges, whereas we satisfy the remaining two children OR and SUB using use-use edges emanating from XOR. The resulting wake-up graph is shown in Figure 1(c).

The execution schedule is realized as follows. When the DIV instruction completes its execution, it wakes-up the instructions LW and MUL through its def-use edges d0 and d1. When LW completes its execution, it wakes up the ADD instruction through its d0 edge. Once ADD is completed it activates the XOR and SLL instructions also through d0 and d1 edges. When XOR is activated, it can now activate OR and SUB through its l0 and l1 links. The resulting schedule is given in Figure 1(c). As we can see, in this case, the schedule generated using our wake-up algorithm is the same as that generated by the central window algorithm. In case of the DBMA architecture, the execution of the MUL instruction can be delayed till the dependence chain from DIV to XOR completes execution and releases a FIFO.

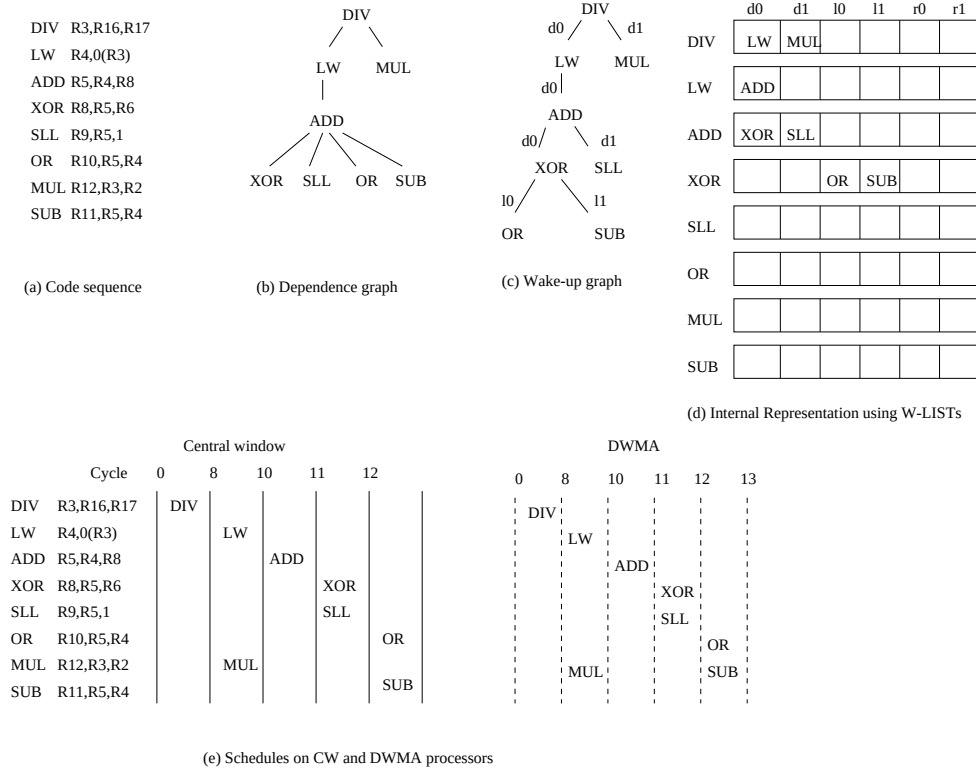


Figure 1: Example DWMA Schedule.

3 The Direct Wake-up Microarchitecture

Next we describe the direct wake-up microarchitecture (DWMA) that has been designed to dynamically construct the wake-up graph and make use of it during instruction scheduling. As shown in Figure 2, DWMA is a highly parallel decoupled superscalar with an eight stage pipeline. The construction of the wake-up graph is begun in the Decode and Graph GEN-1 stage and completed in the Rename and Graph GEN-2 stage. The graph constructed is stored in the form of W-LISTs associated with instructions in their respective reorder buffer entries. Since all instructions that have entered the processor pipeline must have a reorder buffer entry, the size of the instruction window from which instruction level parallelism is extracted is limited by the reorder buffer size. The subset of instructions that are to be considered by the wake-up mechanism are fetched and examined from the reorder buffer by the Fetch W-LIST and Wake-up stage while the updates to the wake-up graph are performed by the Register Read and W-LIST WB stage through write backs to the appropriate entries in the reorder buffer.

Each stage has a width equal to the issue width. The instructions that are executed in parallel are obtained from the heads of the FIFOs and there is a one-to-one correspondence between the FIFOs and the functional units. The instructions are put into the FIFO in the form of instruction descriptors which, in addition to the usual opcode and physical register numbers needed for the instruction's execution, also contain additional information required to carry out the wake-up activities associated with the instruction. Each instruction's descriptor is entered into the queue when it is first fetched and every time it is woken up by another instruction. Thus, the FIFOs contain only a subset of instructions from the instruction window. In particular, these are the instructions that according to our wake-up heuristic may be ready for execution.

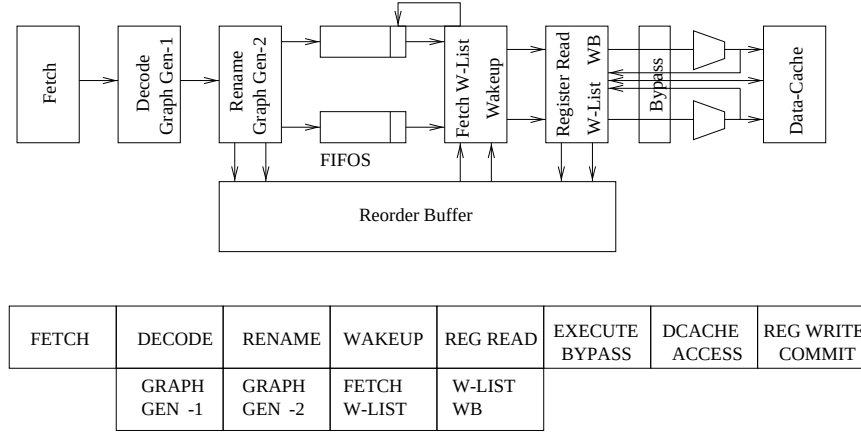


Figure 2: The Direct Wake-up Microarchitecture

Reorder Buffer. There are two significant challenges in implementing the direct wake-up technique for a superscalar processor. The first one which has already been discussed is the difficulty of bounding the number of links that emanate from each instruction. The presented solution to this problem requires that each instruction is capable of waking up instructions through its def-use edges as well as use-use edges. In essence, an instruction can wake up to 6 instructions (2 through the produced result and two instructions through each of the operands). The second problem which is related to the first is the large number of ports required to implement the desired functionality. DWMA addresses these two problems by following a unique reorder buffer design.

The reorder buffer is issue-width interleaved and its address space is divided equally among the individual FIFOs. In other words, the low order bits of the reorder buffer addresses also identify the queue (hence the functional unit) that will process the instruction. Furthermore, it selects the reorder buffer bank assigned to that functional unit. In other words, the reorder buffer index allocated to the instruction number modulo the number of functional units yields the functional unit that will execute this instruction and hence the reorder buffer bank where the wake-up list for the instruction will be stored. Since each functional unit processes a single instruction, only the wake-up list of the currently processed instruction needs to be fetched from the corresponding reorder buffer entry and each functional unit accesses its own reorder buffer bank to fetch the wake-up list of the instruction it is processing.

In order to provide sufficient number of ports without jeopardizing the processor clock, each reorder buffer bank is composed of 7 columns. The first column stores control information about the instruction as usual in any reorder buffer based superscalar implementation. The remaining 6 columns each store the descriptors that the current instruction should wake-up. As a result, each reorder buffer column element has only a single read and a single write port. As the issue width is increased, only the number of functional units and the number of reorder buffer banks increases.

It should also be noted that the reorder buffer is a collection of simple arrays of data locations since instructions fetch their data values from the provided register file. This design provides a very large instruction window without endangering the clock speed because of the distributed nature of the buffer implementation. The design of the reorder buffer is illustrated in Figure 3.

Key Pipeline Stages. Let us consider the operation of the key pipeline stages that construct and process the wake-up graph in in greater detail.

Decode and Graph GEN-1: In this stage, the instruction is decoded and in parallel the producers' W-LISTS, to which the current instruction must be added, are identified from the source register identifiers. A reorder buffer entry is

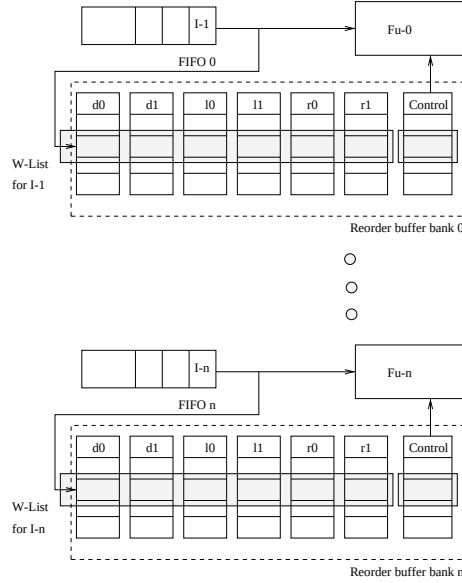


Figure 3: DWMA Reorder Buffer Design

allocated to the instruction by incrementing the tail pointer of the reorder buffer.

Rename and Graph GEN-2: The source register identifiers of the instruction are renamed to physical registers and a result register is allocated if necessary. In parallel, the current instruction and its W-LIST are written to the instruction's assigned reorder buffer entry. The instruction is assigned to a functional unit (or FIFO). The assignment is carried out in a round robin fashion and thus can be computed as the reorder buffer index modulo the issue width. An instruction descriptor is formed and sent to the appropriate FIFO.

Fetch W-LIST and Wake-up: The reservation table is accessed to determine whether the operands of the instruction descriptors at the heads of the FIFOs are ready. In parallel, the reorder buffer is accessed to fetch the W-LISTS of these instructions. The instructions corresponding to (l0,l1) are woken up if the left operand is available, the instructions corresponding to (r0,r1) are woken up if the right operand is available, and the instructions corresponding to (d0,d1) are woken up if the instruction takes a single cycle to execute. The ready woken up descriptors are pushed back to the heads of the appropriate FIFOs. The current instruction descriptors, that were initially obtained from the heads of the FIFOs, have now been processed. The ones that are found to be ready and have their functional unit free are sent to the next stage. On the other hand if the functional unit is busy, the ready descriptors are pushed back to the head of their respective FIFOs. If the instruction is not ready, then it is simply forwarded to the next stage.

Register Read and W-LIST WB: If the current descriptor is ready, the operand values are read from the register file. If it is not ready the descriptor is written back to the W-LIST of the producer of the missing operand in the reorder buffer.

As we can see from the above description, processing of W-LISTS is done in parallel with the conventional functions of the pipeline stages Decode, Rename, Wake-up and Register Read. Therefore the additional tasks necessary to implement the direct wake-up are juxtapositioned with the conventional pipeline functions in Figure 2.

We have explained the overall operation of DWMA. In the remainder of this section we will describe the detailed algorithm for generating the wake-up graph and describe how the instruction scheduling is carried by DWMA in greater detail.

Graph generation algorithm. For the purpose of graph generation, we provide an array of queues, with each logical register being associated with a single queue. Each entry in a queue is a descriptor identifying a producer for the register's data value. A descriptor contains three fields, namely the reorder buffer index of the producer, the wake-up group of the producer (left operand, right operand, or result) and a one bit counter. When an instruction is being decoded, its source register identifiers are used to access the producer queue array and the descriptor at the head of the queue is copied to the instruction's corresponding operand. The descriptor's counter is incremented, and if it now overflows, the descriptor is removed from the head of the queue. Otherwise, the updated descriptor is left at the head of the queue. This process ensures that there are at most two edges emanating from the instruction corresponding to each of its operands and its result. Two new descriptors are formed, one for each operand which can now serve as new producers of these values and they are inserted at the tail of the corresponding queues. Please note that if there is no space left in a queues, the instruction decode need not stall but can simply discard the new descriptors it just formed since there are already quite a few producers that can be used as wake-up slots by the following instructions that need the same value.

Instruction scheduling. DWMA implements the instruction scheduling uniformly by propagating instruction specific information in the form of instruction descriptors which are hardware pointers uniquely identifying the instruction. The format of the instruction descriptor is shown in Figure 4. The My-ROB-i field is the index of the reorder buffer entry allocated to this instruction. The Op-bits field indicate operand availability; there is one bit per operand. Left and Right register numbers are physical (renamed) registers for this instruction. Finally the PTR-missing field is a pointer to the W-LIST of a producer instruction. When both of the operands are missing, only one of them has to be recorded as part of the instruction descriptor since the descriptor itself is stored into the W-LIST of the other operand.

My ROB-i	Op-bits	Left Reg#	Right Reg#	PTR-Missing
----------	---------	-----------	------------	-------------

Figure 4: Instruction descriptor.

The actual processing of the instruction descriptors stored into the FIFOs is handled at a rate of one descriptor per FIFO through the Fetch W-List and Wake-up stage. Each descriptor field is used to handle the wake-up process. The processing of each of the descriptors consists of two main steps. These are determining whether or not the instruction described by the current descriptor is ready and waking up those instructions which are in the wake-up list of the current instruction. The process of determining the status of the current instruction is handled in parallel with the fetching of the wake-up list.

Determining the status of the instruction. The status of the current instruction is determined by accessing the reservation table. For this purpose, the physical register identifiers are sent to the reservation table to fetch the operand availability. Please note that only the instruction described by the descriptor needs to access the reservation table, as the ready status for all the instructions being woken-up can be easily computed from their descriptors and the operand status for the current descriptor.

Fetching of the Wake-up List. Simultaneously with the reservation table access, the stage sends the My-ROB-i field of the instruction descriptor to the reorder buffer to fetch the W-LIST associated with this instruction. Since the descriptor is at the head of the queue, it should now wake-up any instructions which are in its wake-up list.

Once the W-LIST and the operand availability information is obtained, the set of instructions which are ready are easily identified using the operand availability information for the currently processed instruction descriptor. The instruction itself is ready, if both operands are available. The dependent instructions are ready if they are only missing the data of the

source operands of the currently processed instruction (i.e., the edges are of use-use type), or if the current instruction is a single cycle integer instruction whose result would make them ready. As a result, the following operations are performed in parallel to identify the instructions which would become ready as a result of the wake-up operation carried out by the current instruction:

- if the left operand of the currently processed instruction is available, instructions which are in the left operand group l0,l1 have also one of their operands ready (i.e., through a use-use edge). To update the operand availability information for these instructions, one zero bit from the Op-bits fields of their descriptors is turned on.
- if the right operand of the currently processed instruction is available, instructions which are in the right operand group r0,r1 have also one of their operands ready. The above procedure is applied this time to those instruction descriptors found in the right operand group.
- if the currently processed instruction is a single cycle instruction, instructions which are in the result group d0,d1 will have the required data in the next cycle through a def-use edge. Therefore, for those descriptors found in the result group d0,d1, one zero bit from the Op-bits field of each descriptor is turned on.

The crucial aspect of the above operation is no additional reservation table accesses are necessary to identify those instructions which would be woken-up. Following the operation, we now have a number of descriptors, namely, the original descriptor and its dependent descriptors. We first process the dependent descriptors. If these descriptors are ready they are pushed back to the head of the FIFO. In the next cycle, they can now wake-up their dependents and issue. If they are not ready, they are written back to the reorder buffer by the next stage. If the original descriptor is ready, and the corresponding functional unit is free, the instruction is issued for execution through the Read registers and W-LIST WB stage. If the corresponding functional unit is busy, the descriptor is pushed back to the head of the FIFO.

Longer latency operations need a similar treatment with a different timing. One cycle before the completion of the result, they access their reorder buffer entry to fetch the descriptors in their result wake-up group (d0, d1). These descriptors are inserted to the head of the queue they belong. In the next cycle, these instructions can resume operation if they now have all of their missing operands.

4 Experimental Evaluation

We have implemented a simulator for the DWMA architecture described in the preceding section and evaluated its performance. In evaluating the performance we considered the experimental set up shown in Figure 5. Issue widths of 8, 16 and 32 were considered in comparing the performance of DWMA with a central window processor and the DBMA architecture. Since we did not have a trace cache implementation and a gshare based instruction fetcher cannot provide the required instruction fetch bandwidth for issue widths 16 and above, we have used a gshare based instruction fetcher for 8 issue processors and an ideal fetcher for 16 and 32 issue processors. All the processor configurations employ identical fetchers (i.e., 8 issue DWMA, DBMA and CW all use the gshare fetcher and 16, 32 issue DWMA, DBMA and CW all use ideal fetchers). All processor configurations however employ the out of order store set disambiguator [13]. Since in this implementation both branches and loads are handled speculatively, we also incorporated a recovery mechanism based upon the future file [5] in the implementation.

Under the above conditions we executed the SPEC95 benchmarks on central window, DWMA and DBMA processors for issue widths of 8, 16 and 32. The resulting IPCs for all the benchmarks are shown in Figure 6. An 8-issue DWMA architecture attains 84% and 76% of the 8-issue central window processor performance for integer and floating point benchmarks respectively. In contrast, the 8-issue DBMA architecture only achieves 70% and 54% of the performance

Issue width	8, 16, 32 instructions
Fetch width	Equal to the issue width
Retire width	Twice the issue width
Instruction Window	Issue width * 2
Number of FIFOs	Equal to the issue width
Functional Units	Issue width Symmetric Functional units.
Instruction fetch	Aggressive multi-block gshare
Dcache	Perfect
Memory ports	Issue width / 2

(a) Machine parameters

Functional Unit	Latency (cycles)
Load	2
Integer division	8
Integer multiply	4
Other integer	1
Float multiply	4
Float addition	3
Float division	8
Other float	2

(b) Functional Unit latencies

Figure 5: Machine Configurations.

that can be achieved by a central window processor. At an issue width of 16, DWMA architecture achieves 76 % of the performance of the ideal central window processor for integer benchmarks and 74 % for floating point benchmarks. In contrast, DBMA can only achieve 47 % and 46 % of the performance of the ideal central window processor for integer and floating point benchmarks respectively. The high performance of DWMA is also evident in case of 32 issue processors. With integer benchmarks the same trend is continued at 71 % of the performance of the central window processor. However, with floating point benchmarks DWMA loses some performance achieving about 55 % of the ideal processor. We have traced this performance loss to the sensitivity of the 32 issue processor to instruction schedule ordering, which can be improved upon by employing additional heuristics in the descriptor selection.

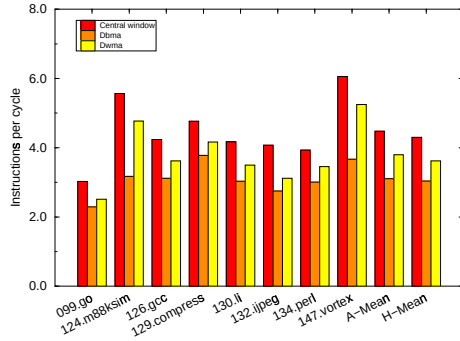
5 Conclusion

In this paper we demonstrated that there is a significant room for improvement in the DBMA wake-up algorithm when load speculation is performed. We show that it provides approximately half of the performance of a central window implementation. We developed a novel approach to wake-up, called direct wake-up, and presented the design of the DWMA architecture. Our detailed evaluation of the DWMA architecture shows that it yields performance superior to the DBMA and achieves 80%, 75% and 63% of the performance of a central window processor at issue widths of 8, 16 and 32.

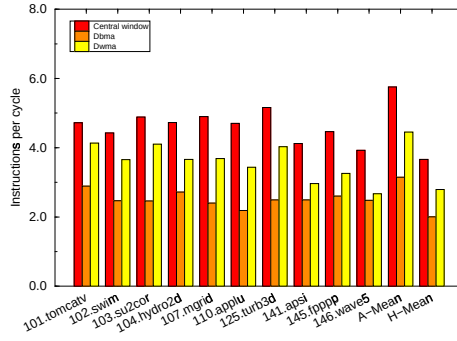
References

- [1] M. Butler, T.Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *Proceedings of the 18th International Conference on Computer Architecture*, pages 276–286, 1991.
- [2] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th International Conference on Computer Architecture*, pages 142–153, June 1998.
- [3] Daniel Holmes Friendly, Sanjay Jeram Patel, and Yale N. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. In *The 30th Annual IEEE-ACM International Symposium on Microarchitecture*, pages 24–33, December 1997.
- [4] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.
- [5] Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.
- [6] Gregory A. Kemp and Manoj Franklin. A decentralized dynamic scheduler for ilp processing. In *Proceedings of the International Conference on Parallel Processing*, volume 1, pages 239–246, 1996.
- [7] Mikko H. Lipasti and John Paul Shen. Superspeculative microarchitecture for beyond ad 2000. *IEEE Computer*, 30(9):59–66, 1997.

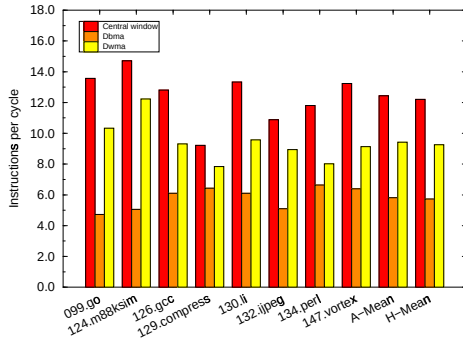
8 Issue - Integer Benchmarks



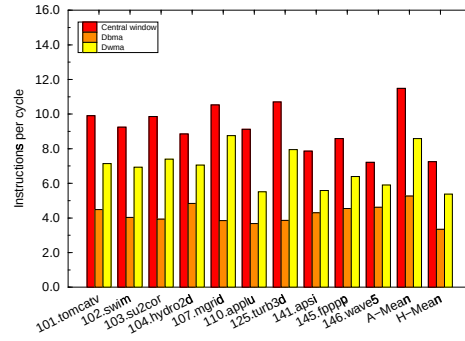
8 Issue - Floating point Benchmarks



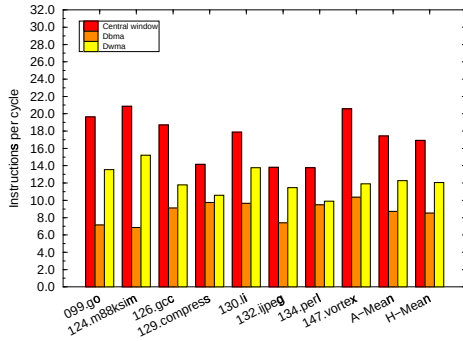
16 Issue - Integer Benchmarks



16 Issue - Floating point Benchmarks



32 Issue - Integer Benchmarks



32 Issue - Floating point Benchmarks

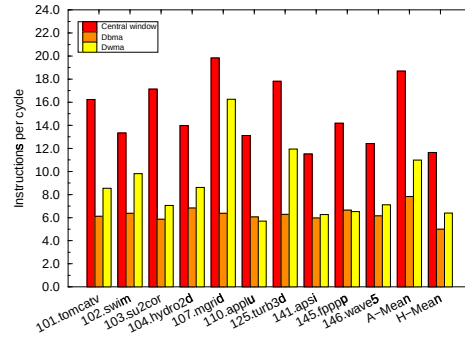


Figure 6: IPC values for DWMA

- [8] Scott McFarling. Combining branch predictors. Technical Report WRL-TN-36, Digital Western Research Laboratory, 1993.
- [9] Andreas I. Moshovos. *Memory Dependence Prediction*. PhD thesis, University of Wisconsin - Madison, 1998.
- [10] Andreas I. Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th International Conference on Computer Architecture*, pages 181–193, June 1997.
- [11] Soner Önder and Rajiv Gupta. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages*, pages 80–89, Chicago, May 1998.
- [12] Soner Önder and Rajiv Gupta. Superscalar execution with dynamic data forwarding. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 130–135, October 1998.
- [13] Soner Önder and Rajiv Gupta. Dynamic memory disambiguation in the presence of out-of-order store issuing. To appear in the 32nd Annual IEEE-ACM International Symposium on Microarchitecture, November 1999.
- [14] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Quantifying the complexity of superscalar processors. Technical Report CS-TR-96-1328, University of Wisconsin Technical Report, 1996.
- [15] Subbarao Palacharla, Norman P. Jouppi, and J.E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th International Conference on Computer Architecture*, pages 206–218, June 1997.
- [16] Yale N. Patt, Sanjay J. Patel, Marius Evers, Daniel H. Friendly, and Jared Stark. One billion transistors, one uniprocessor, one chip. *IEEE Computer*, 30(9):51–57, 1997.
- [17] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 24–34, December 1996.
- [18] James E. Smith and Sriram Vajapeyam. Trace processors: Moving to fourth generation microarchitectures. *IEEE Computer*, 30(9):68–70, 1997.
- [19] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22th International Conference on Computer Architecture*, pages 414–425, 1995.
- [20] Sriram Vajapeyam, P. J. Joseph, and Tulika Mitra. Dynamic vectorization: a mechanism for exploiting far-flung ilp in ordinary programs. In *Proceedings of the 26th International Conference on Computer Architecture*, pages 16–27, May 1999.
- [21] Sriram Vajapeyam and Tulika Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proceedings of the 24th International Conference on Computer Architecture*, pages 1–12, June 1997.