# Instruction Wake-Up in Wide Issue Superscalars

Soner Önder[1] and Rajiv Gupta[2]

[1] Michigan Technological University, Houghton, MI 49931
[2] The University of Arizona Tucson, AZ 85721

**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. 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. 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.

## 1 Introduction

Out-of-order execution allows superscalar processors to take advantage of instruction level parallelism extracted from an instruction stream. However, the potential for speedups supported by the observed levels of instruction level parallelism remain far from being fully realized. To extract high levels of parallelism one problem that must be addressed is that of designing a scalable instruction issue mechanism. A linear increase in extracted parallelism requires a quadractic increase in instruction window size [2].

The 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 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 [4].

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.

Our approach is related to the one taken in the design of the dependence based microarchitecture (DBMA) [4] which 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.

## 2 Direct Wake-up

The wake-up algorithm proposed by Palacharla et al. in [4] is based upon the 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. However, 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 limits the performance of DBMA.

Our approach provides 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 will be issued in the next cycle.

To guide the above wake-up heuristic we must dynamically construct a wake-up graph. To obtain an effective hardware design for storing the graph 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 the W-LIST of an instruction should contain all def-use edges emanating from it. But since the number of entries in the W-LIST must be a fixed small number in a practical implementation we place a bound on this number. Therefore 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 by these two instructions. The latter is achieved by introducing use-use edges from the first two instructions and the next four instructions and so on. Thus, 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.

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. Note that in the worst case an instruction may be woken up at most two times before it can execute.

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

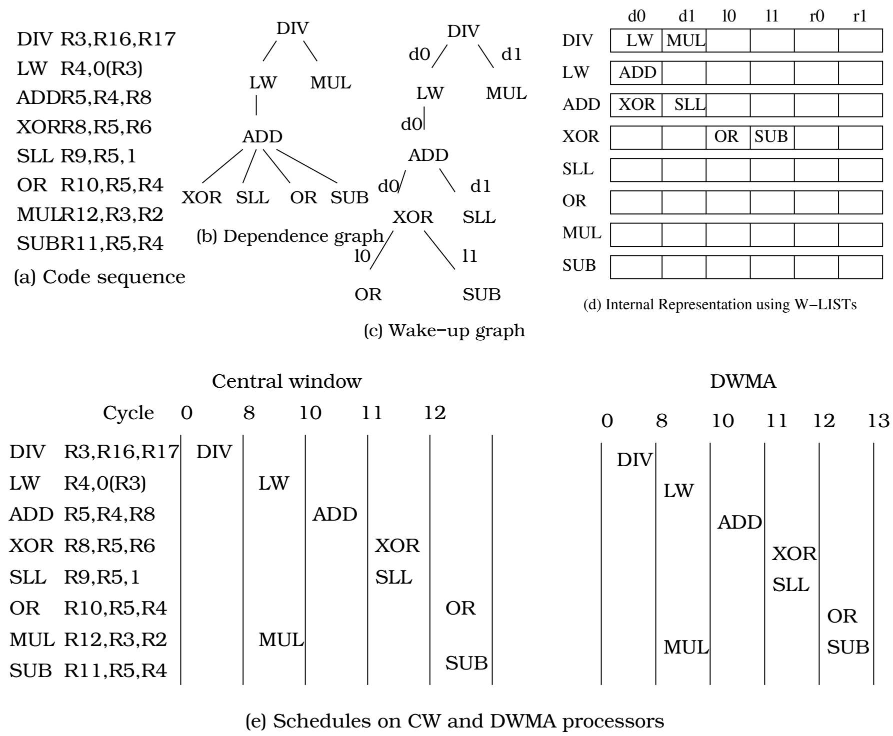
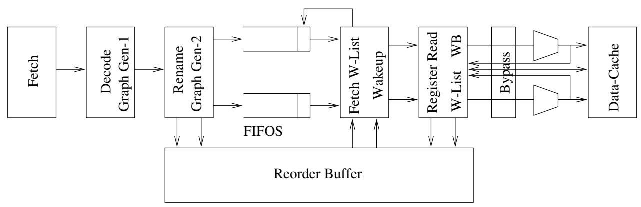



**Fig. 1.** Example DWMA Schedule.

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.

## 3 The Direct Wake-up Microarchitecture

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 is stored in the form of W-LISTs associated with instructions in their respective reorder buffer entries. Therefore 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.




| FETCH | DECODE | RENAME | WAKEUP | REG READ | EXECUTE BYPASS | DCACHE ACCESS | REG WRITE COMMIT |
|---|---|---|---|---|---|---|---|
| | GRAPH GEN -1 | GRAPH GEN -2 | FETCH W-LIST | W-LIST WB | | | |

**Fig. 2.** The Direct Wake-up Microarchitecture

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. FIFOs contain instruction descriptors which, in addition to providing the usual opcode and physical register numbers needed for an instruction's execution, also contain information required to carry out the wake-up activities associated with the instruction. Each instruction is entered into the queue when it is first fetched and every time it is woken up. The FIFOs essentially contain those instructions that are likely to be ready for execution.

*Reorder Buffer.* 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. Therefore large number of ports are required to implement the desired wake-up functionality. DWMA addresses this problem 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. Thus, 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.

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.

*Key Pipeline Stages.* Let us consider the operation of the key pipeline stages that construct and process the wake-up graph in greater detail. In the Decode and Graph GEN-1 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 allocated to the instruction by incrementing the tail pointer of the reorder buffer. In Rename and Graph GEN-2 stage 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. An instruction descriptor is formed and sent to the appropriate FIFO.

In Fetch W-LIST and Wake-up stage 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. Finally, in Register Read and W-LIST WB stage 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.

*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. If there is no space left in the queue the new descriptors are simply discarded.

*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 3. 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 |

**Fig. 3.** 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 proceeds as follows. 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. 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 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, 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 simulated the DWMA, DBMA and central window algorithms using the FAST system [1]. The experimental set up is shown in Figure 4. We have used a gshare based instruction fetcher for 8 issue processors and an ideal fetcher for 16 and 32 issue processors. We employ a modified version of the store set disambiguator described in [3]. Misspeculation recovery is based upon the future file.

| | |
|---|---|
| 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

**Fig. 4.** Machine Configurations.

The resulting IPCs for all the benchmarks in SPEC95 are shown in Figure 5. An 8/16/32-issue DWMA attains 84/76/71% and 76/74/55% of the 8/16/32-issue central window processor performance for integer and floating point benchmarks respectively. In contrast, the 8/16/32-issue DBMA architecture only achieves 70/47/50% and 54/46/42% of the 8/16/32-issue central window processor performance for integer and floating point benchmarks respectively. The DWMA is significantly more effective than DBMA.

## Acknowledgements

Supported by DARPA PAC/C Award. F29601-00-1-0183 and NSF grants CCR-0105355, CCR-0096122, and EIA-9806525 to the University of Arizona.

## References

1. S. Önder and R. Gupta. Scalable Superscalar Processing. Ph.D. Thesis, Univ. of Pttsburgh, July 1999.
2. S. Önder and R. Gupta. Superscalar execution with dynamic data forwarding. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 130–135, October 1998.
3. S. Önder and R. 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.
4. S. Palacharla, N.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.
5. S. Vajapeyam and T. 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.

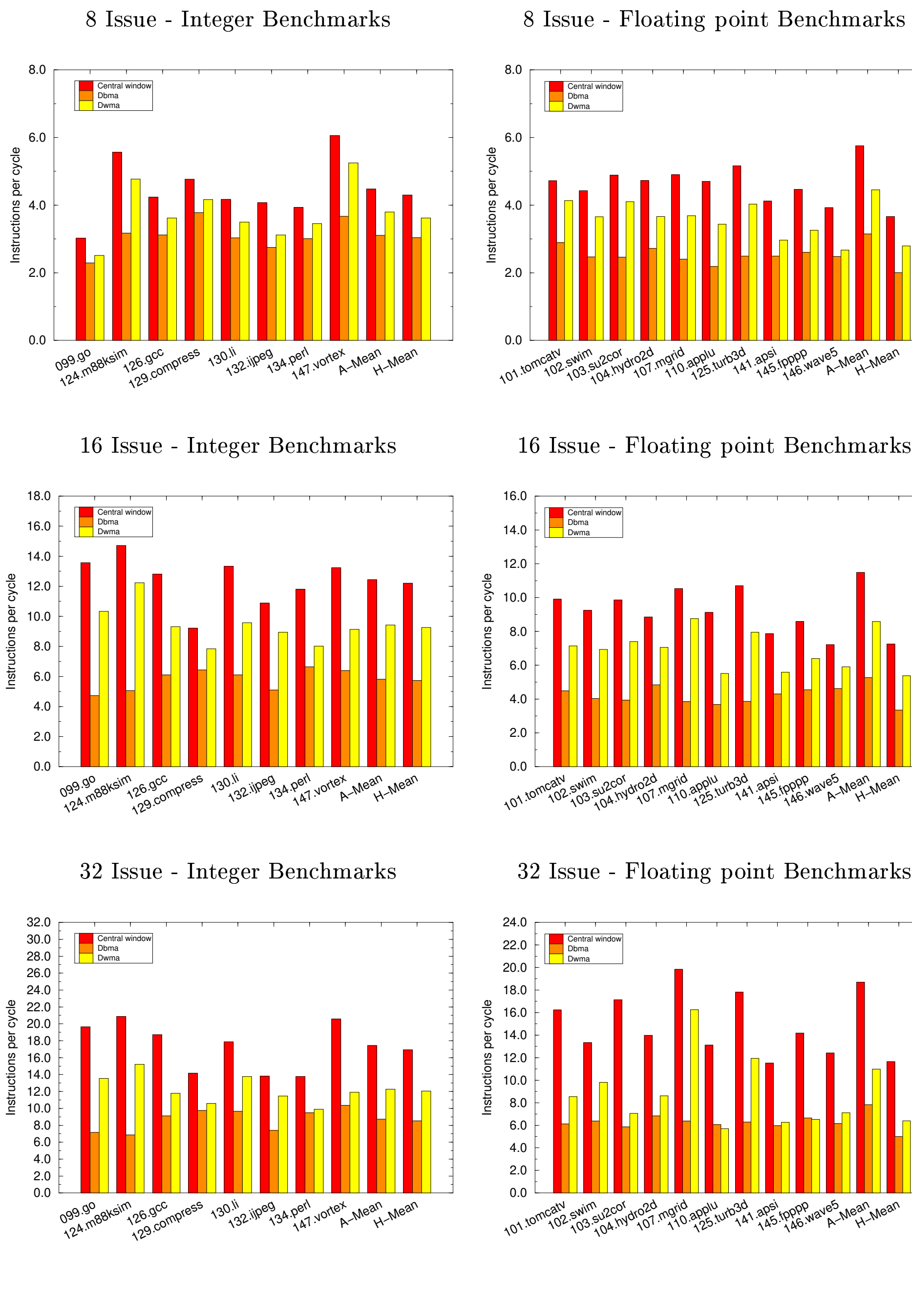



**Fig. 5.** IPC values for DWMA