Macro-op Scheduling: Relaxing Scheduling Loop Constraints

Ilhyun Kim and Mikko H. Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin−Madison
{ikim,mikko}@ece.wisc.edu

Abstract

Ensuring back-to-back execution of dependent instructions in a conventional out-of-order processor requires scheduling logic that wakes up and selects instructions at the same rate as they are executed. To sustain high performance, integer ALU instructions typically have single-cycle latency, consequently requiring scheduling logic with the same single-cycle latency. Prior proposals have advocated the use of speculation in either the wakeup or select phases to enable pipelining of scheduling logic to achieve higher clock frequency. In contrast, this paper proposes macro-op scheduling*, which systematically removes instructions with single-cycle latency from the machine by combining them into macro-ops, and performs nonspeculative pipelined scheduling of multi-cycle operations. Macro-op scheduling also increases the effective size of the scheduling window by enabling multiple instructions to occupy a single issue queue entry. We demonstrate that pipelined 2-cycle macro-op scheduling performs comparably or even better than atomic scheduling or prior proposals for select-free scheduling.*

1. Introduction & Motivation

A major obstacle to building high-frequency out-of-order microprocessors is the instruction scheduling logic because wakeup and select operations are not easily pipelined in conventional designs. This difficulty is compounded by the trend of deeper pipelining and the ever-increasing performance gap between memory system and processor core, which requires larger instruction queue structures to tolerate long latency operations; this may prohibitively increase the complexity and circuit delay of scheduling logic. Given these constraints, naively scaling conventional scheduling logic is impractical, as it will fail to meet both architectural and circuit requirements at the same time. To address this problem, many researchers have proposed several techniques for either reducing scheduling logic complexity or scaling the instruction window without severely affecting the cycle time.

This paper also proposes a technique to achieve similar benefits, but looks at the problem from a different perspective. Specifically, we explore the design space of instruction scheduling logic, evaluate benefits when varying the *granularity* of the scheduling unit in the scheduling logic, and expose a greater opportunity at a different level by relaxing the constraints imposed by *instruction-centric* design. Figure 1 depicts a spectrum of the scheduling logic design space at different granularities. In conventional designs, scheduling decisions occur at instruction boundaries and issue queue entries are allocated per instruction. As a finer-grained approach, there have been proposals for *operand-granular* scheduling logic that prioritizes operand wakeups and decouples half of the tag matching logic from the wakeup bus to reduce the load capacitance [4][5]. These techniques enable the wakeup logic to operate at a higher frequency with minimal performance impact. As a coarser-grained approach in the opposite direction, the AMD K7 and the Intel Pentium M have adopted techniques to allow an issue queue entry to accommodate multiple micro-ops as a form of fused operations for certain types of x86 instructions [6][18]. Original micro-ops are loosely coupled in a fused operation from the scheduler's perspective; they are scheduled individually according to the readiness of corresponding source operands. This approach is effective in reducing contention in issue queue as well as other portions of pipeline.

Going a step further from reduced issue queue pressure achievable at a coarser-level queue management, the atomicity of wakeup and select operations can be relaxed by increasing the scheduling granularity from single to multiple instructions. We propose *macro-op scheduling* that transforms a series of instructions into a multi-cycle scheduling unit that we call *macro-op* (*MOP*), and performs pipelined scheduling of multi-cycle operations while the processor core still executes dependent instructions consecutively. Combined with the relaxed scalability constraint due to reduced issue queue pressure, this technique can achieve comparable or even better performance than atomic scheduling while enabling the reduced complexity of pipelined scheduling logic.

The rest of the paper is organized as follows. Section 2 describes the base pipeline and scheduling model used in this paper. Section 3 presents an overview of macro-op scheduling. Section 4 characterizes the suitability of instructions to be grouped into MOPs. Section 5 gives a detailed explanation of macro-op scheduling logic. Section 6 provides a performance evaluation of macro-op scheduling. Section 7 reviews other related work. The main conclusions are summarized in Section 8.

2. Machine Model

2.1. Pipeline Overview

The base machine model is a conventional superscalar out-of-order processor. Figure 2 illustrates the pipeline structure used in this paper. After instructions are fetched and decoded, the source and target register identifiers are

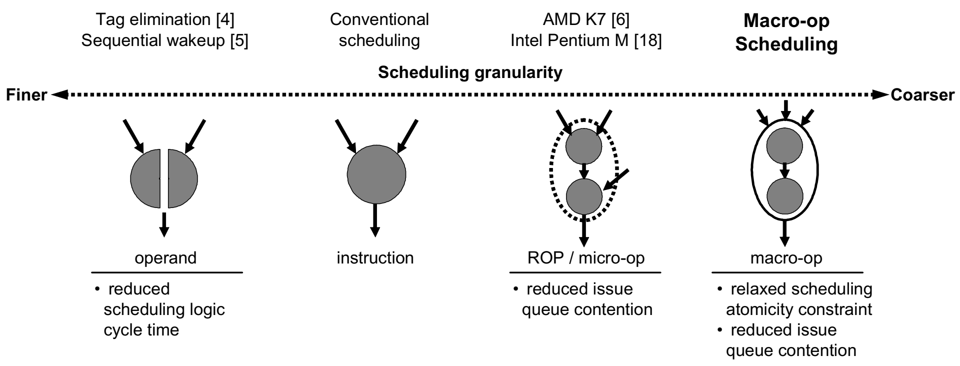


Figure 1. A spectrum of the scheduling logic design space at different scheduling granularities.

Fetch	Decode	Rename	Rename	Queue	Sched	Disp	Disp	RF	RF	Exe	WB	Commit

Figure 2. Processor pipeline.

renamed and physical registers are assigned to remove false dependences. In the queue stage, instructions check the most recent ready status of input operands, and are inserted into free issue queue entries. In the scheduling stage, a set of wakeup and select operations links data dependences among instructions and speculatively issues instructions. In order to reduce the scheduling logic complexity, the payload RAM [8] is located next to the scheduling stage and the actual register identifiers and opcodes are accessed from this separate structure. At the same time, issued instructions are dispatched to the execution pipeline. Since load latency is not deterministic, instructions dependent on loads are scheduled assuming the common case cache hit latency. If dynamic events, e.g. cache misses, incur unexpected delays, load-dependent instructions that have been issued within the load shadow [1] are selectively invalidated and replayed with correct inputs after the misscheduling condition is resolved. If the scheduling is correct and instructions are successfully executed, they release issue queue entries which are reused for newer instructions.

For memory operations, a load instruction that finishes its address calculation proceeds to the memory stage (not shown in the figure), which is located next to the execution stage. A store instruction is decoded as two separate operations (e.g. an effective address generation and actual store operation), and writes the store data into the memory system when the instruction is committed. This configuration is similar to the one used in the Pentium 4 [2].

2.2. Baseline Scheduling Logic

The function of scheduling logic is to wake up instructions dependent on the instructions issued in the previous cycle, and to select the next issue candidates from the pool of ready instructions. This set of wakeup and select operations is performed every clock cycle to issue dependent instructions consecutively.

In this paper, we study macro-op scheduling built on two different styles of wakeup logic arrays: conventional CAM-style and wired-OR-style [8][12]. CAM-style wakeup logic usually has two tag comparators to support up to two source operands for each instruction. Many conventional processor implementations use physical register specifiers as tags. A scheduling cycle starts when an issued instruction broadcasts its tag through the wakeup bus. Other instructions in the issue queue compare the tags of their source operands, and set ready bits if they match. When both source operands become ready, the instruction sends a request signal to select logic that selects ready instructions to issue considering the available resources and the priorities of instructions. The selected instructions are issued and broadcast their destination tags; At this point, a cycle of scheduling is completed.

The basic operations of wired-OR-style wakeup logic are similar to those of CAM-style wakeup logic, except that ready status and dependence tracking is managed in a dependence vector form. Each bit in the vector represents a dependence on a parent instruction at the corresponding bit location. In order to reduce the number of wires running vertically through the wakeup array, dependence tracking in this wakeup array is managed in a separate name space (i.e. issue queue entry number) from physical register identifiers. This can be enabled by performing a process similar to register renaming, i.e. register to issue queue entry name conversion. Each instruction monitors the readiness of source operands every clock cycle by checking if all wakeup lines of matching dependence bits are asserted, and sends a request signal to select logic. When an instruction is issued, it asserts the wakeup line corresponding to its own issue queue entry. This process in turn wakes up dependent instructions that have matching bits in their dependence vectors.

3. Macro-op Scheduling Overview

A naive relaxation of the scheduling atomicity constraint (i.e. atomic wakeup / select within a single clock cycle) may lead to a significant performance loss because dependent instructions cannot execute in consecutive clock cycles. This constraint is imposed by the minimal execution latency of instructions; the vast majority of ALU operations execute in a single clock cycle and hence scheduling of dependent instructions should be fast enough to keep up

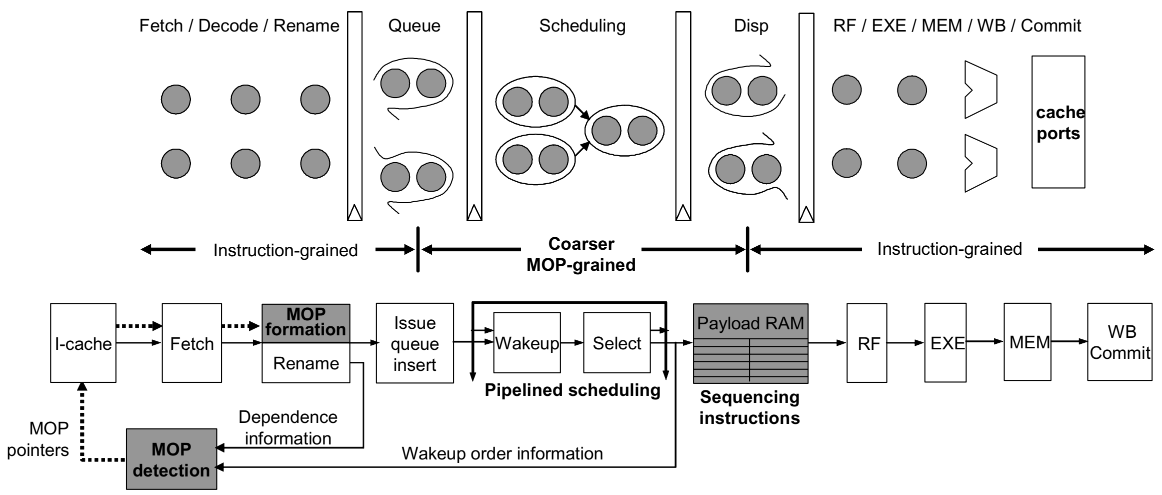


Figure 3. Coarser-grained macro-op scheduling.

with executing them. If the execution latencies of all types of instructions were greater than one clock cycle, the scheduling loop would be no longer restricted to one clock cycle, and the wakeup and select operations could expand over multiple clock cycles with respect to the minimal execution latency. However, it is hard to imagine that hardware designers would give up single-cycle operations even in future microprocessors running at an extremely high clock speed because instructions can still be issued consecutively by using e.g. staggered adders [2][17] that allow dependent computations to overlap.

Instead, the atomicity constraint can be relaxed by increasing the scheduling granularity from single to multiple instructions. Macro-op scheduling groups multiple instructions, converts them into MOPs with multi-cycle latencies, and forces scheduling decisions to occur at MOP boundaries.

Figure 3 illustrates the basic concept of macro-op scheduling and its corresponding pipeline stages. The *MOP detection* logic located outside the processor's critical path examines register dependences among instructions and creates *MOP pointers*. A MOP pointer is stored in the instruction cache, and specifies which instructions can be grouped. When MOP candidate instructions are located based on MOP pointers, the *MOP formation* logic converts them into a MOP which occupies only a single issue entry. The instruction scheduler performs pipelined scheduling of multi-cycle MOPs and issues them when all source operands become ready. An issued MOP accesses the payload RAM, which sequences the original instructions in the instruction-grained execution pipelines.

3.1. Benefits of Macro-op Scheduling

Figure 4 shows an example of macro-op scheduling in which each MOP can contain two instructions. The original data dependence graph was taken from *gzip*. All instructions in the figure are single-cycle operations.

Instructions grouped in a MOP behave in the scheduler as a single unit; a MOP can be issued only when all source dependences are satisfied and it incurs only one tag broadcast. For these coarser-level controls over instructions, the source and destination dependences of original instructions need to be coalesced as MOPs are created. When two dependent instructions are grouped, the maximum number of source dependences is three, assuming an instruction in this architecture can have up to two source operands. Conventional CAM-style wakeup logic may lose some grouping opportunities if each issue queue entry has only two source comparators. However, wired-OR-style wakeup logic does not have this restriction because the bit vector can represent more than two source dependences by marking extra bit locations. In order to handle multiple destination dependences, they are merged into one MOP dependence and hence the dependence between the *MOP head* (the first instruction of a MOP) and the *MOP tail* (the last instruction of a MOP) does not incur a tag broadcast. These dependence conversions remove dependence edges or replace them with false edges, abstracting the data dependence graph without violating the true dependences. However, they simply alter the way instructions are scheduled and still ensure correctness of execution since register values are accessed based on the original data dependences.

Macro-op scheduling relaxes the atomicity constraint of instruction scheduling, enabling pipelined scheduling logic that issues dependent instructions consecutively. In the base case (Figure 4a), the wakeup and select operations must be performed within a single clock cycle (*1-cycle scheduling*) to achieve consecutive execution of dependent instructions. In contrast, a MOP has a two-cycle latency and hence macro-op scheduling (Figure 4b) can perform a set of wakeup and select operations every two clock cycles (*2-cycle scheduling*). Instructions not grouped into MOPs (instruction 6 and 7 in the figure) due to no matching pair behave as in conventional 2-cycle scheduling, and dependent instructions cannot be issued consecutively. The dependence tree depth for the example increases from 9 to only 10 clock cycles in macro-op scheduling, while it becomes 17 clock cycles in conventional 2-cycle scheduling.

Macro-op scheduling increases the effective size of the window because multiple instructions are processed as a single unit in the scheduler and hence an issue queue entry

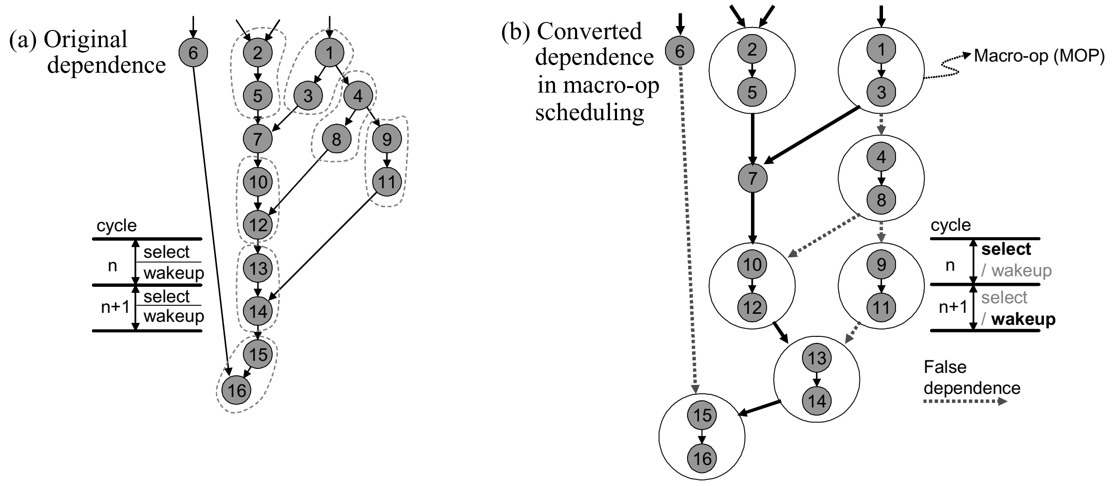
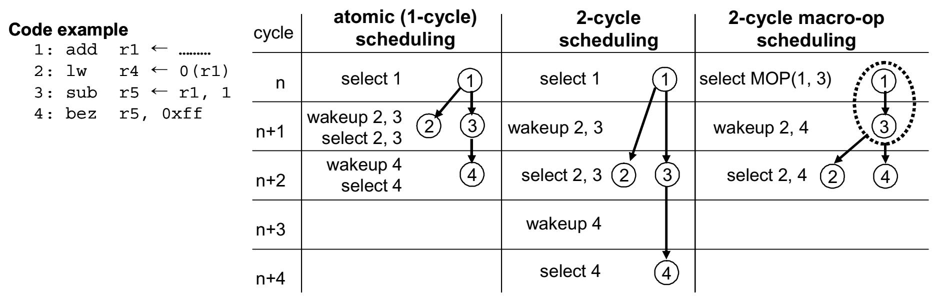


Figure 4. An example of macro-op scheduling.



Code example

```
1: add  r1 ← ........
2: lw   r4 ← 0(r1)
3: sub  r5 ← r1, 1
4: bez  r5, 0xff
```

cycle	atomic (1-cycle) scheduling	2-cycle scheduling	2-cycle macro-op scheduling
n	select 1	select 1	select MOP(1, 3)
n+1	wakeup 2, 3 select 2, 3	wakeup 2, 3	wakeup 2, 4
n+2	wakeup 4 select 4	select 2, 3	select 2, 4
n+3		wakeup 4	
n+4		select 4	

Figure 5. Wakeup and select timings.

can logically hold multiple original instructions. In the example, the macro-op scheduler consumes only 9 issue queue entries for 16 instructions. This enables the scheduler to better tolerate long latency events with the same number of issue queue entries.

Figure 5 presents detailed scheduling timings in conventional 1-cycle, 2-cycle, and 2-cycle macro-op scheduling. In 2-cycle scheduling, the minimal latency of dependence edges is two clock cycles and the critical path of the data dependence graph is negatively affected due to wakeup delay. In macro-op scheduling, many 2-cycle dependence edges can be shortened through grouping instructions into a MOP. In the macro-op scheduling example, instructions 1 and 3 are grouped; the issued MOP sequences the two instructions so they are effectively scheduled as if 1-cycle scheduling is performed. However, the MOP itself has a 2-cycle latency from the perspective of scheduling logic. Instructions dependent on the MOP head perform as in conventional 2-cycle scheduling (instruction 2 in the example); hence the issue timing is the same as 2-cycle scheduling. Note that instructions dependent on the MOP tail are scheduled consecutively (instruction 4 in the example) since the wakeup operation can be hidden behind the execution latency of the MOP.

4. Issues in MOP Formation

Macro-op scheduling benefits from shortening dependence edge latency and reducing issue queue contention by grouping dependent instructions. Since the processor pipeline processes instructions in program order before they are inserted into the out-of-order window, MOP candidate instructions should also be located near each other to be detected and grouped within a reasonable scope. In this section, we characterize the dynamic instruction distance between MOP candidate instructions, measure effectiveness of MOP formation, and determine the MOP policies to be used for the remainder of this study.

4.1. Candidate Instruction Types

Since our primary goal is to relax the atomicity of the scheduling loop, macro-op scheduling targets single-cycle operations: single-cycle ALU, store address generation, and control (e.g. branch) instructions. Among these *MOP candidates*, instructions that generate register values and hence can have dependent instructions will be referred to as *value-generating candidate* instructions, which potentially degrade performance in 2-cycle scheduling by delaying issue of their dependent instructions. Other types such as long-latency integer ALU (e.g. multiply), loads, and floating-point operations already have multi-cycle latencies and therefore do not require 1-cycle scheduling.

4.2. MOP Formation Scope

Macro-op scheduling groups a chain of dependent instructions and converts them into a multi-cycle latency MOP. In order to determine the scope for MOP formation, we characterize the dependence edge distance measured in

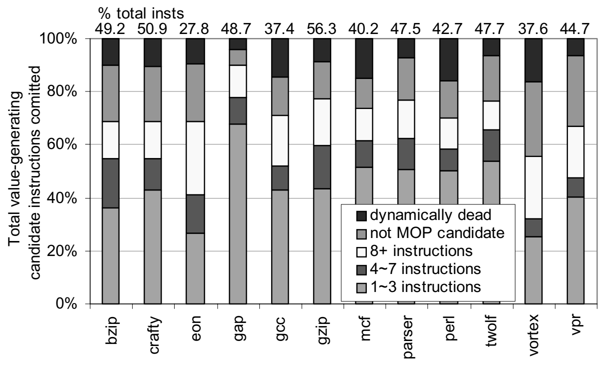


Figure 6. Characterization of dependence edge distance between two candidate instructions.

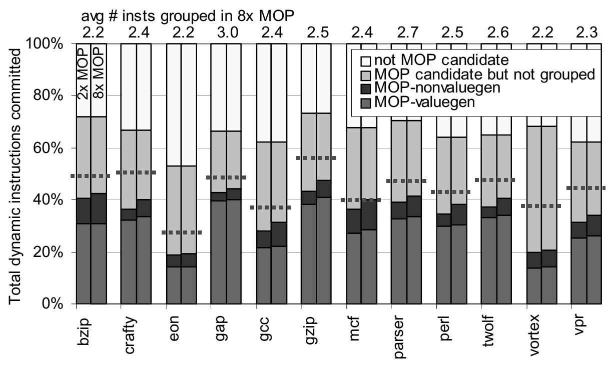


Figure 7. Characterization of instructions groupable into different MOP sizes.

terms of instruction count between two candidate instructions in Figure 6. In the graph, the y-axis represents all potential MOP heads (i.e. value-generating candidate instructions). The percentage out of total committed instructions is shown on the top of each bar (*% total insts*). The stacked bars in each benchmark show the distance between each MOP head and the nearest potential MOP tail (i.e. dependent single-cycle instruction). If there is no MOP tail, we count the MOP head as either *dynamically dead* (when there is no dependent instruction) or *not MOP candidate* (when the dependent instruction is not a MOP candidate). We note that the data shown here illustrates program characteristics, and is not dependent on machine configuration.

On average, 73% of MOP heads have at least one potential MOP tail across the benchmarks we tested. In general, an 8-instruction scope (shown as *1~3* and *4~7 instructions*) captures most cases and is therefore the focus of our work.

Dependence edge distances vary noticeably from benchmark to benchmark. For instance, *gap* has short dependence edges between two MOP candidates and 87% of MOP pairs are detected within 8 instructions in program order. On the other hand, *vortex* has relatively longer dependence edges so only 54% of MOP candidate pairs are detected in the same scope. The performance effects of this variability are discussed further in Section 6.4.

4.3. MOP Size

Given an 8-instruction scope determined in the previous section, we characterize the number of instructions in dependence chains that can be grouped into a MOP. Again, we note that the data shown in this section is independent of machine configuration.

Figure 7 shows the percentage of grouped instructions for two different configurations: *2x MOP* and *8x MOP*. The 2x MOP configuration allows only two instructions to be grouped. The 8x MOP configuration can group as many instructions as possible within the given 8-instruction scope. The y-axis shows the total instructions committed, in which stores are counted simply as non-value-generating candidate operations (store address generations). *Not MOP candidate* shown in the top-most bars are multi-cycle instructions like loads, while the other three stacked bars represent MOP candidate instructions. Grouped instructions are presented in two categories: *MOP-valuegen* (value-generating candidate instructions grouped) and *MOP-nonvaluegen* (other candidate instructions grouped, which can be MOP tails only). Dotted lines present all value-generating candidate instructions (corresponding to the 100% line in Figure 6) so the MOP coverage of such instructions can be derived by comparing dotted lines and MOP-valuegen bars. We do not plot other 3~7x MOP configurations for simplicity of presentation but show the average number of instructions in 8x MOPs instead (*avg # insts grouped in 8x MOP*) to present potentials for different MOP sizes.

Across the benchmarks, 53~73% of total instructions are MOP candidates. The actual grouped instructions are 32.9% and 35.4% on average in 2x and 8x MOPs, respectively, ranging from 18.7% in *eon* to 47.3% in *gzip*. The data indicate that more than two instructions can be grouped in many cases. Although bigger MOP sizes enable the scheduling loop to span over more clock cycles and further reduce queue contention, this study will evaluate the potentials of grouping two instructions since our primary goal is to relax the scheduling atomicity. Evaluating other MOP configurations is left for future work.

5. Macro-op Scheduling Logic

In the previous sections, we presented the principles of macro-op scheduling, discussed the benefits of scheduling instruction at a coarser granularity and determined macro-op scheduling logic configurations (grouping two instructions within an 8-instruction scope) to be used in our study.

This section describes the details of two major components in macro-op scheduling: MOP detection and MOP formation. Although this paper studies instruction scheduling, our technique does not require significant changes in the scheduling logic itself. The issues in pipeline structure and performance will be also discussed later in this section.

5.1. MOP Detection

The purpose of MOP detection logic is to examine the instruction stream to detect MOP candidates considering data dependences, the number of source operands (for the wakeup logic with only two tag comparators) and possible cycle conditions, and to generate *MOP pointers* that represent MOP pairs. Since MOP detection logic is located out-

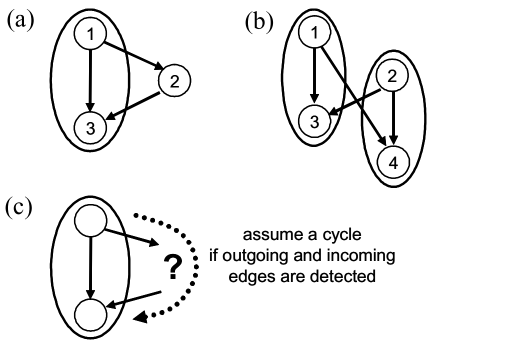


Figure 8. Cycle conditions and a detection heuristic.

side the processor's critical path, it neither increases the pipeline depth nor affects the processor's cycle time.

5.1.1. Cycle detection

Macro-op scheduling abstracts true data dependences by creating false dependence edges when instructions are grouped (explained in Section 3.1). These false dependences may prevent instructions from being issued if they induce cycles in data dependence chains. Figure 8 illustrates possible deadlock conditions created by improper MOP grouping. In Figure 8a, a MOP that contains instructions 1 and 3 has both incoming and outgoing edges to instruction 2. Figure 8b also shows a cycle condition between two MOPs. In both examples, no instruction can be issued earlier than the other since the original source dependences cannot be satisfied, leading to a deadlock.

Although MOP detection logic may filter out deadlock conditions by tracking multiple levels of dependences along with dependence chains, this would significantly increase complexity. Instead, we use a simple heuristic to detect possible cycle conditions conservatively, as shown in Figure 8c; if there is an outgoing dependence edge from the MOP head to other instructions preceding the MOP tail in program order, and the MOP tail also has an incoming edge, the detection logic assumes there may be a potential cycle and foregoes a grouping opportunity. Although some MOPs may be falsely detected to induce cycles by this conservative detection heuristic, our initial experiments (not detailed here) determined that it still achieves over 90% of possible MOP formation opportunities compared to the precise cycle detection. The actual implementation of this heuristic is presented in the following section.

5.1.2. MOP detection logic

Figure 9 shows an example of instruction streams from cycle n to n+2 sent from the rename stage to the MOP detection logic, as well as the detection process that finds candidate pairs and generates MOP pointers using dependence matrices through steps n~n+2.

In the figure, a triangle matrix in each step represents register dependences among instructions currently being examined. Two rectangular matrices on the top and the left represent validity and status of the detection process. In fact, these two rectangular matrices on both sides are identical but showed separately for simplicity of presentation. If any *inval* (invalid or not a candidate instruction), *head* (detected as MOP head) or *tail* (detected as MOP tail) bit is marked, the corresponding row or column is not examined for grouping, which is presented as shadowed boxes.

The basic MOP detection algorithm is to scan column entries vertically and to select an entry that contains a dependence mark that represents a register dependence. If there are multiple entries, the priority decoder selects the first entry if possible. A dependence mark can be "1" or "2", which shows the number of source operands. For example, in step n of the figure, instruction 2 has one "1" (representing the dependence on instruction 1) and instruction 3 has two "2" (representing the dependences on instruction 1 and 2). They are used to detect possible cycles; "1" can be selected without any restriction; "2" can be selected only when it is the first mark in the column. This policy implements the cycle detection heuristic described in Section 5.1.1.

In step n, instructions 1~4 fill the bottom right portion of the triangle matrix. When instruction 1 scans the corresponding column vertically in order to find a matching pair, the entry corresponding to instruction 2 is ignored because it is not a MOP candidate and hence it has an *inval* bit in its rectangular matrices. Although the next entry also contains a dependence mark, it cannot be selected either because the cycle detection heuristic does not allow dependence mark "2" to be chosen across other marks, implying that the MOP head and tail have both incoming and outgoing edges at the same time. In step n+1, instructions 5~8 fill the triangle matrix from the bottom right portion, and instructions 1~4 are moved to the top left portion. The bottom left portion of the matrix represents inter-group dependences and dependence marks are written to corresponding entries. Instructions 3, 4 and 7 find possible matching pairs

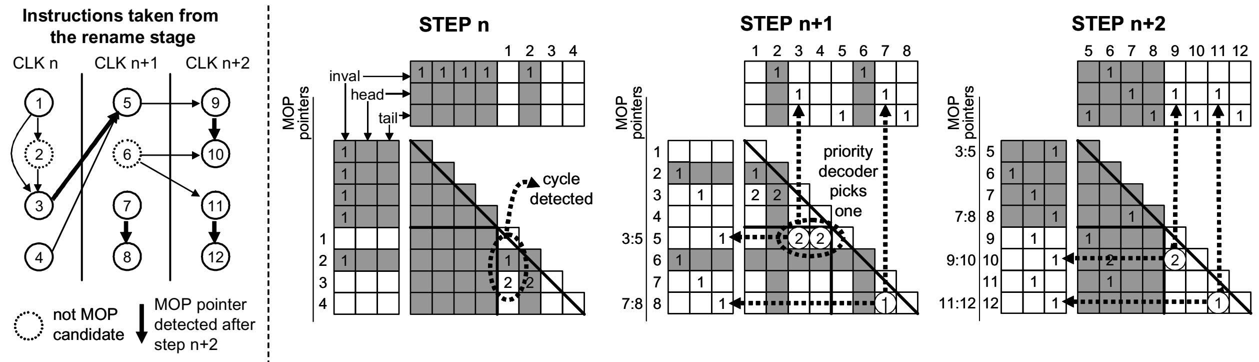


Figure 9. MOP detection process.

after scanning their entries vertically. If an instruction is selected by multiple instructions (e.g. instruction 5 is selected by both instructions 3 and 4), the priority decoder picks only one, resolving the conflict. At the end of step n+1, two MOP pointers are generated. Selected instructions mark corresponding *head* or *tail* fields so that they will not be examined again. Similarly, instructions 9~12 are examined in step n+2 and four MOP pointers are finally generated.

Considering the complexity of operations to be performed in MOP detection process, one can imagine that generating MOP pointers would take many clock cycles. Fortunately, our experimental results indicate that the detection latency does not affect performance critically because MOP pointers are stored in the instruction cache and used repeatedly. We believe that MOP detection logic can be implemented efficiently in a way similar to proposals for instruction preprocessing at trace cache line construction time [21][22][23]. The detailed complexity estimation of this part of the design is left as future work.

5.1.3. MOP pointers

After MOP pointers are generated by MOP detection logic, they are stored into the first-level instruction cache and later fetched along with instructions to direct MOP formation. A MOP pointer is a 4-bit forward pointer from MOP head to tail instructions, in which one bit represents a possible control flow change and three bits represent the offset between two instructions. The control bit captures up to one control flow discontinuity created by a single direct branch or jump. If there is an intervening indirect jump, or there are multiple control instructions and any of them are taken between the MOP head and tail, the MOP detection logic does not generate a MOP pointer. The 3-bit offset field is simply the instruction count from the MOP head to the MOP tail, covering 8 instructions.

Since each instruction has only one pointer, dynamic control flow changes may prevent instructions from being grouped. For example, if a MOP pointer is created across a taken branch, and the branch is later predicted to be not taken, MOP formation logic compares the current control flow to the control bit in the pointer, and does not group with an unexpected instruction in the fall-through path.

5.2. MOP Formation

MOP formation is responsible for checking control flow, locating MOP pairs using the MOP pointers, and converting register dependences into MOP dependences. Two instructions are later inserted into a single issue entry in the queue stage, creating a MOP in the scheduler.

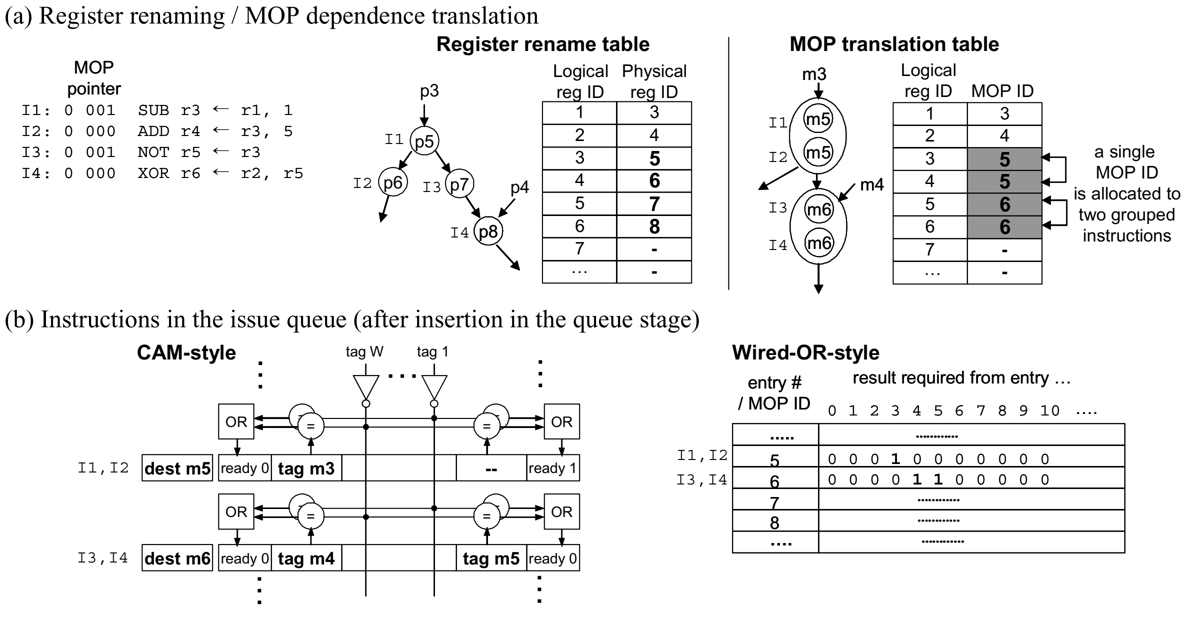
5.2.1. Locating MOP pairs

Locating MOP pairs is the reverse process to MOP pointer creation; it compares the control flow predicted by the branch predictor and control bits in MOP pointers and checks if MOP tail instructions are available using offset bits in MOP pointers. If the MOP pointer is valid, this information is sent to MOP dependence translation logic.

5.2.2. MOP dependence translation

Macro-op scheduling abstracts the original data dependence chains and therefore requires dependence conversion from register to MOP IDs so that scheduling logic keeps track of dependences in a separate name space. We do not believe that translating to this name space will incur much delay. As discussed in Section 2.2, a similar name space conversion is already required for wired-OR-style wakeup logic that specifies register dependences in terms of issue queue entry numbers rather than physical register IDs.

Figure 10a illustrates the register renaming and MOP dependence translation processes. In parallel with register renaming, the *MOP translation table* converts logical registers into the MOP ID name space. In fact, the process and hardware structure required for this translation is identical to what is required for register renaming, except that a single MOP ID can be allocated to two MOP candidate


(a) Register renaming / MOP dependence translation
Register rename table
MOP translation table
a single MOP ID is allocated to two grouped instructions
(b) Instructions in the issue queue (after insertion in the queue stage)
CAM-style
tag W
tag 1
OR
dest m5
ready 0
tag m3
--
ready 1
dest m6
tag m4
tag m5
I1,I2
I3,I4
Wired-OR-style
p3
p5
p6
p7
p4
p8
m3
m4
m5
m6

```
      MOP
    pointer
I1: 0 001   SUB r3 ← r1, 1
I2: 0 000   ADD r4 ← r3, 5
I3: 0 001   NOT r5 ← r3
I4: 0 000   XOR r6 ← r2, r5
```

Logical reg ID	Physical reg ID
1	3
2	4
3	**5**
4	**6**
5	**7**
6	**8**
7	-
…	-

Logical reg ID	MOP ID
1	3
2	4
3	**5**
4	**5**
5	**6**
6	**6**
7	-
…	-

	entry # / MOP ID	result required from entry … 0 1 2 3 4 5 6 7 8 9 10 ….
	.....	....
I1,I2	5	0 0 0 1 0 0 0 0 0 0 0
I3,I4	6	0 0 0 0 1 1 0 0 0 0 0
	7	....
	8	....
	....	....

Figure 10. Dependence translation in MOP formation.

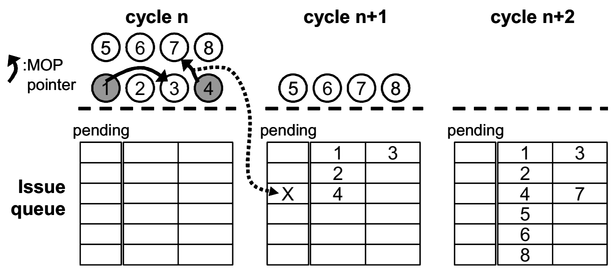


Figure 11. Inserting instructions into issue queue.

instructions specified by a MOP pointer, while in register renaming a physical register is allocated to every destination register. For example, a single identifier `m5` is assigned to both instructions `I1` and `I2` so that any instruction dependent on them will become a child of the MOP `m5`. The MOP ID name space can be issue queue entry numbers, or can even use an arbitrary name space as big as the number of physical registers. Unlike physical register IDs which are associated with actual data storage, MOP IDs are only used to track MOP-grained dependences in the scheduler. Note that register renaming is still performed in parallel and the register values are accessed based on the original data dependences. Figure 10b shows issue queue entries occupied by instructions `I1~I4` grouped in two MOPs. Each entry has source identifiers that are the union of all source operands of two original instructions. The number of source operands is not limited for wired-OR wakeup logic, but is limited to two by MOP detection logic for the wakeup array with two source comparators.

5.2.3. Insertion policy

When instructions are fetched and processed in the pipeline, the MOP head and tail may reside in different pipeline stages because the MOP detection scope is greater than the machine bandwidth (i.e. an 8-instruction scope on the 4-wide machine in our experiments). Moreover, dynamic events such as cache misses or control flow changes may complicate instruction grouping if MOP tails are not delivered to the pipeline in a timely manner. Therefore, our mechanism groups instructions only when they are in the same or two consecutive pipeline stages. Consistent with this policy, the scheduling logic prevents the MOP head from being scheduled before the MOP tail is subsequently inserted into the same entry. Figure 11 illustrates this insertion policy into the issue queue. In cycle n, instructions 1 and 4 have MOP pointers to instructions 3 and 7, respectively. In cycle n+1 when instructions 1~4 are inserted into the issue queue, instructions 1 and 3 occupy a single entry, creating a MOP. Since instruction 7 is in the next insert group when instruction 4 is inserted, instruction 4 sets a *pending* bit, implying that the entry is waiting for the MOP tail and will not request a grant signal from select logic. When instruction 7 is inserted in cycle n+2, the pending bit is cleared and the MOP of instructions 4 and 7 will be issued when all source operands become ready.

5.3. Pipeline Considerations

5.3.1. Sequencing instructions

An issued MOP is converted back to two original instructions that are sequentially executed. This functionality is achieved by dual-entry payload RAM in the dispatch stage. When an issued MOP accesses the payload RAM, the opcodes and register specifiers of two original instructions are acquired, and each instruction is sent down to the appropriate execution pipeline within two consecutive clock cycles. If the base machine does not have the payload RAM structure, sequencing instructions can still be performed by the scheduling logic, similarly to the AMD K7 or the Intel Pentium M [6][18]. Since a MOP is equivalent to non-pipelined 2-cycle operation from the scheduler's perspective, the selection logic does not select and issue another instruction through the same issue slot in which a MOP is being sequenced.

5.3.2. Branch and load mis-speculation handling

If two instructions are grouped across a mispredicted branch, the MOP tail is invalidated and removed from the issue queue and the payload RAM when instructions are squashed. At the same time, the source operand fields associated with the MOP tail instruction are set to ready state so that the MOP head that remains in the issue queue can be scheduled without waiting for incorrect source operands. Even if the MOP tail has already been executed before a branch misprediction or even an exception condition is discovered, it does not affect correctness of the architectural state since the ROB commits ungrouped original instructions separately in program order.

Our scheduling replay mechanism selectively invalidates and reschedules instructions dependent on misscheduled loads. In this load mis-scheduling case, both instructions in a MOP are replayed since the scheduler keeps track of dependences in the MOP name space. Although this policy may replay some instructions unnecessarily, our initial experiments (not detailed here) found that the performance impact is negligible.

5.4. Performance Considerations

5.4.1. Grouping independent instructions

So far, we have discussed MOP grouping of two dependent instructions (*dependent MOP*). There are also cases where two independent instructions can be grouped if both instructions have no source operands or identical source operands. These *independent MOPs* are captured by the MOP detection logic after detecting all possible dependent MOPs, so that it does not lose benefits of grouping dependent instructions. If a pair of instructions is not selected as either MOP head or MOP tail and has the same source dependences, a MOP pointer is created and instructions are later grouped in the same way as dependent MOPs.

Independent MOPs do not shorten dependence edge latencies. Rather, they serialize issue of two independent instructions and may affect performance negatively in some timing-critical cases (e.g. mispredicted branch resolution). However, instructions dependent on independent MOPs can be executed in the same clock cycle as the base 2-cycle scheduling case because the 2-cycle MOP latency hides the wakeup operation. Our experiments determined that their negative impact is not significant, and that they can positively affect performance by reducing issue queue contention in many cases. The number of independent

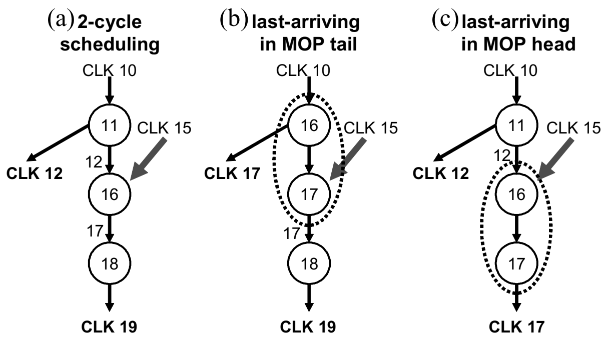


Figure 12. The effects of last-arriving operands.

MOPs will be presented in Section 6.3.

5.4.2. The effects of last-arriving operands

A MOP may negatively affect performance if the source operand associated with the MOP tail is the last-arriving operand that triggers issuing the MOP. Figure 12 illustrates this scenario, in which the second instruction in each case has a source operand that is awakened at clock 15. The numbers denoted in instructions and dependence edges represent issue and wakeup timings, respectively. Since instructions dependent on the MOP tail are scheduled in a consecutive clock cycle, the last-arriving operand in the MOP tail may not degrade performance compared to the base 2-cycle scheduling (CLK 19 in both Figure 12a and b). However, we observed that macro-op scheduling experiences some difficulties in *gap*, losing many opportunities for shortening dependence edges. The worst-case scenario is that instructions dependent on the MOP head are unnecessarily delayed (broadcast at CLK 17 in Figure 12b).

To avoid harmful grouping, we use a filtering mechanism in the MOP detection logic; if a last-arriving operand in a MOP tail is observed during execution, MOP detection logic deletes the MOP pointer in the instruction cache (writing a zero-value pointer) and searches for an alternative pair, as shown in Figure 12c.

6. Experimental Results

6.1. Simulation Methodology

Our execution-driven simulator used in this study is derived from the *SimpleScalar* / *Alpha* 3.0 tool set [13], a suite of functional and timing simulation tools for the Alpha AXP ISA. Specifically, we extended *sim-outorder* to perform speculative scheduling with selective replay. We modeled a 13-stage out-of-order pipeline with 4-instruction machine width. We note that the Alpha binaries contain many no-ops, and they are filtered out by the decoder without executing them [15]. The pipeline structure is illustrated in Figure 2. The detailed configurations are shown in Table 1. The SPEC CINT2000 integer benchmark suite is used for all results presented in this paper. FP benchmarks were not tested since multi-cycle FP instructions do not need 1-cycle schedulers. All benchmarks were compiled with the DEC C/C++ compilers under the OSF/1 V4.0 operating system using -O4 optimization. Table 2 shows the benchmarks, input sets, the number of instructions committed, and the base IPCs. The large reduced input sets from [14] were used for all benchmarks except for *crafty, eon* and *gap*. These three benchmarks were simulated with the reference input sets up to 3 billion instructions since the reduced inputs are not available.

Parameter	Configuration
Out-of-order Execution	4-wide fetch/issue/commit, 128-entry ROB, unrestricted / 32-entry unified issue queue, speculative scheduling with selective replay (2-cycle penalty), fetch stops at first taken branch in a cycle
Functional Units (latency)	4 integer ALUs (1), 2 FP ALUs (2), 2 integer MULT/DIV (3/20), 2 FP MULT/DIV (4/24), 2 general memory ports
Branch Prediction	Combined bimodal (4k entries) / gshare (4k entries) with a selector (4k entries), 16 RAS, 1k-entry 4-way BTB, at least 14 cycles taken for misprediction recovery
Memory System (latency)	16KB 2-way 64B line IL1 (2), 16KB 4-way 64B line DL1 (2), 256KB 4-way 128B line unified L2 (8), main memory (100)

Table 1: Machine configurations.

Benchmark	input sets	inst count	Base IPC (32-issue entry / unrestricted)
bzip	lgred.graphic	2.64B	1.40 / 1.53
crafty	crafty.in	3B	1.45 / 1.55
eon	chari.control.cook	3B	1.86 / 2.13
gap	ref.in	3B	1.73 / 2.10
gcc	lgred.cp-decl.i	5.12B	1.24 / 1.29
gzip	lgred.graphic	1.79B	1.79 / 1.99
mcf	lgred.in	0.79B	0.34 / 0.38
parser	lgred.in	4.52B	1.06 / 1.12
perl	lgred.markerand	2.06B	1.22 / 1.33
twolf	lgred.in	0.97B	1.36 / 1.50
vortex	lgred.raw	1.15B	1.60 / 1.75
vpr	lgred.raw	1.57B	1.48 / 1.64

Table 2: Benchmarks.

6.2. Scheduler Configurations

To measure the effectiveness of macro-op scheduling, we modeled base scheduling, 2-cycle scheduling, macro-op scheduling with CAM-style (2 source operands) and wired-OR-style wakeup logic, and select-free scheduling (1-cycle wakeup and 1-cycle select) [8]. Base scheduling has ideally pipelined scheduling logic, which is conceptually equivalent to conventional atomic scheduling with one extra pipeline stage. All performance data are normalized to this case. 2-cycle scheduling logic has pipelined wakeup and select stages, resulting in a one-cycle bubble between a single-cycle instruction and its dependent instructions. macro-op scheduling logic is built on top of 2-cycle scheduling logic, and groups two single-cycle operations. Although MOP formation is performed in parallel with register renaming, we evaluate macro-op scheduling with 0, 1 or 2 extra stages in order to model possible extra logic complexity. The MOP detection and formation has a 2-cycle scope, which captures up to 8 instructions on our 4-wide machine. For the detection logic delay, we optimistically assumed 3 clock cycles from examining dependences

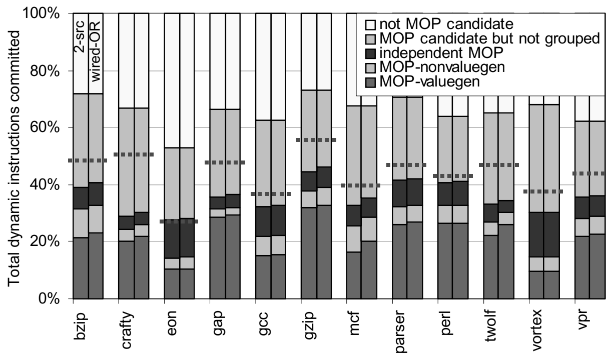


Figure 13. Grouped instructions in macro-op scheduling.

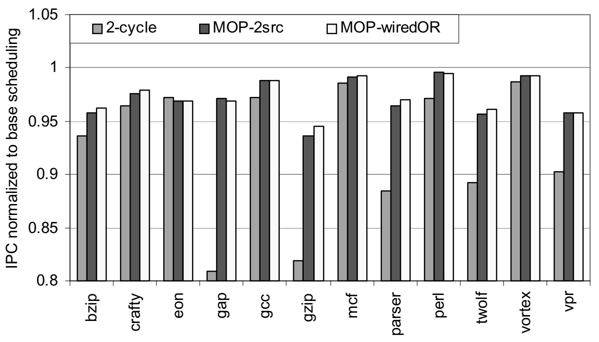


Figure 14. Vanilla macro-op scheduling performance (unrestricted issue queue / 128 ROB, no extra pipeline stage).

to generating MOP pointers. However, we also tested a pessimistic 100-cycle detection delay and found the performance degradation is very slight (average 0.22%, worst 0.76% in *parser*) because MOP pointers stored in instruction cache are used repeatedly. Two types of select-free scheduling logic were modeled: the *select-free-squash-dep* configuration invalidates all dependent instructions selectively when a collision victim is detected in the select stage and hence no pileup victim exists (modeled after *Squash Dep, select-4* configuration in [8]); the *select-free-scoreboard* detects pileup victims using a scoreboard integrated in the register file (modeled after *Scoreboard, select-4* configuration in [8]). All configurations have the same pipeline depth with the exception of 1 or 2 extra stages in macro-op scheduling.

6.3. Grouped Instructions

Figure 13 shows the percentage of grouped instructions in macro-op scheduling. Each benchmark has two bars: CAM-style with two source comparators (*2-src*) and wired-OR-style wakeup logic (*wired-OR*). The data in this graph is plotted in the same way as in Figure 7, except for an added category for independent MOPs. The MOP coverage difference between the data here and the characterization data (2x MOP) in Figure 7 can be interpreted as coming from implementation limitations (e.g. discontinuous instruction fetch, 2-cycle detection scope instead of 8-instruction scope, cycle detection heuristic, MOP pointer restrictions and so on).

MOP-valuegen and *MOP-nonvaluegen* represent dependent MOPs that shorten 2-cycle dependence edges. The number of dependent MOPs is correlated to dependence edge distance shown in Figure 6; *eon* and *vortex* show low MOP potential and hence fewer instructions are grouped. Regarding value-generating candidate instructions, the data indicate that dependent MOPs potentially enable 26~63% of such instructions to be scheduled as if atomic scheduling is performed. We note that the MOP coverage may not be directly proportional to the degree of benefit since the criticality of instructions varies. Across the benchmarks, 28~46% of total instructions are grouped into either dependent or independent MOPs, achieving an average 16.2% reduction in instructions inserted into the scheduler.

6.4. Performance of Macro-op Scheduling

Figure 14 presents macro-op scheduling performance when no issue queue contention exists (unrestricted issue queue / 128 ROB) and no extra stage is added for MOP formation. Here, macro-op scheduling does not benefit from queue contention reduction.

2-cycle scheduling (shown in the left bars) suffers a performance drop of 1.3% (*vortex*) ~ 19.1% (*gap*). These slowdowns correlate to the dependence edge distance measured in Figure 6. In particular, *gap* exhibits relatively short distances between dependent instructions so the instruction window is filled up with chains of dependent instructions, which prevents 2-cycle scheduling logic from finding a sufficient number of ready instructions and leads to a significant performance loss. Conversely, 2-cycle scheduling is likely to issue many independent instructions every clock cycle in a benchmark with long dependence edges (e.g. *vortex)*.

Macro-op scheduling (shown in the middle and right bars) achieves 97.2% of base performance on average. Because macro-op scheduling enables pipelined scheduling logic to issue dependent instructions in consecutive cycles, the degree of its performance gain over 2-cycle scheduling tends to increase as 2-cycle scheduling suffers. Especially, *gap, gzip, parser, twolf* and *vpr* experience 10% or more performance degradation with 2-cycle scheduling but macro-op scheduling makes up a significant portion of this. *Eon* shows a slight slowdown in macro-op scheduling due to the secondary effect of independent MOPs (discussed in Section 5.4.1). In some benchmarks such as *gap* and *perl,* macro-op scheduling with 2-source wakeup logic performs better than with wired-OR wakeup logic although slightly less MOP pairs are captured. This comes from the negative effect of last-arriving operands in the MOP tail (discussed in Section 5.4.2), although the detection mechanism filters out many MOPs exhibiting this behavior.

Relatively low coverage of MOP candidates due to long dependence edges (as shown in Figure 6) does not severely affect overall performance even though only a few instructions can be grouped, since the baseline 2-cycle scheduling is already able to find plenty of independent instructions to issue. In fact, MOP formation complements 2-cycle scheduling by finding instruction-level parallelism in cases where the 2-cycle scheduler is not able to do so. Specifi-

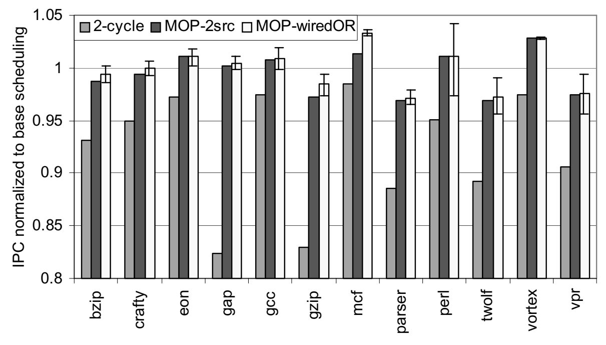


Figure 15. Macro-op scheduling performance under issue queue contention (32 issue queue / 128 ROB).

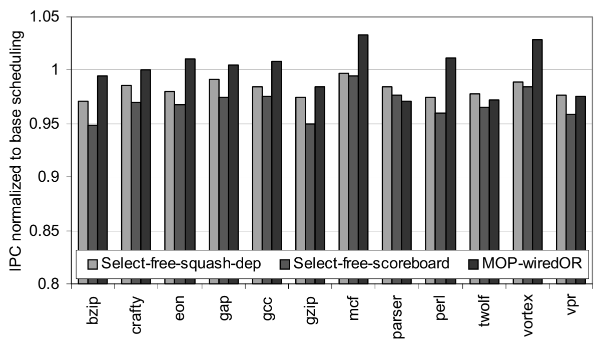


Figure 16. Performance comparison of pipelined scheduling logic.

cally, short-distance dependent pairs are likely to deteriorate the ILP extractable by 2-cycle scheduling and to be more performance-critical. The MOP detection algorithm is set to capture these short-distance dependent pairs, which increases the chances of complementing many performance-degrading instructions.

Figure 15 presents macro-op scheduling performance under issue queue contention (32 issue queue / 128 ROB). In the graph, the solid bars represent performance with 1 extra MOP formation stage, and the error bars indicate performance with 0 or 2 extra stages. The worst case is 3.1% of IPC loss in *parser*. The average slowdown of macro-op scheduling (1 extra stage for MOP formation) with 2-source and wired-OR wakeup logic is measured to be 0.5% and 0.1%, respectively. Overall, macro-op scheduling performance in this machine configuration has noticeably improved over the data shown in Figure 14 because macro-op scheduling reduces issue queue contention by allowing two original instructions to share a single entry. In general, macro-op scheduling performs comparably or shows measurable performance improvement over the baseline. In particular, for *eon, gap, gcc, mcf, perl* and *vortex*, macro-op scheduling (with 1 extra MOP formation stage) outperforms the baseline scheduler.

In conclusion, macro-op scheduling can enable pipelined scheduling logic to perform comparably or even better than atomic scheduling by relaxing the atomicity and scalability constraints in conventional instruction-grained scheduling.

6.5. Performance Comparison of Pipelined Scheduling Logic

Figure 16 compares macro-op scheduling with select-free scheduling logic proposed by Brown et al. [8]. The detailed configurations of select-free scheduling logic were presented in Section 6.2. The base and macro-op scheduling (with 1 extra MOP formation stage) configurations are identical to those of Figure 15. The *Select-free-squash-dep* configuration shown in the left bars performs comparably or slightly worse than macro-op scheduling, with a few exceptions. As noted in the original proposal [8], this configuration requires a hypothetical ideal mechanism for controlling wakeup speculation when it exceeds issue bandwidth. The *Select-free-scoreboard* configuration in middle bars shows more noticeable performance losses compared to macro-op scheduling in most benchmarks. This is because scheduling mis-speculations in this configuration incur pileup victims that consume issue bandwidth and incorrectly wake up subsequent instructions.

Select-free scheduling cannot outperform the baseline scheduler, and experiences performance degradation because of its speculative nature. In contrast, macro-op scheduling is a non-speculative technique and can expose greater opportunity due to relaxed atomicity and scalability constraints.

7. Related Work

Many researchers found that instruction scheduling logic would be a major bottleneck in future microprocessors due to its poor scalability and atomicity. There have been numerous proposals on overcoming these limitations.

Palacharla et al. [9] and Kim and Smith [16] proposed clustered microarchitectures where chains of dependent instructions are issued using a set of FIFO queues that works as a wider and deeper instruction window. Canal and Gonzalez [19], Michaud and Seznec [11] and Raasch et al. [20] proposed data-flow based scheduling that reorders instructions before they enter a small issue window. Lebeck et al. [10] studied the effect of cache misses on the instruction window and explored scheduler designs that re-insert the load and dependent instructions after the cache miss is resolved. Hrishikesh et al. [3] proposed a segmented instruction window in which each segment has a different scheduling priority. To break the atomicity of instruction scheduling loop, Stark et al. [7] described *speculative wakeup* to stretch the wakeup and select operations over two cycles. Brown et al. [8] proposed to move the selection logic out of the scheduling loop in order to pipeline scheduling logic. Most of these studies try to overcome scalability and atomicity constraints in isolation. In contrast, our work explores the scheduler design space at a coarser level with a consistent view to those problems, relaxing both constraints simultaneously.

Interlock collapsing or dependence collapsing techniques [24][25][23][21] merge a series of dependent instructions into one single-cycle operation with more operands, reducing execution latency. In a sense, our approach is a scheduler-side collapsing technique that exploits a similar grouping process to improve scheduling latency rather than execution latency itself. Since macro-op scheduling alters only the dependence mapping in the scheduler and unmodified original instructions are executed, it requires no changes in the datapath (e.g. special ALUs or 3-source register read ports), nor any special handling of the intermediate result that other dependent instructions may consume.

The terminology of macro-op or MOP originally came from AMD; the AMD K7 [6] decodes x86 instructions into MOPs that contain single or multiple RISC primitives. The Intel Pentium M [18] also has adopted a technique of fusing multiple micro-ops to reduce the number of micro-ops to be processed. The instruction schedulers convert K7 MOPs or fused micro-ops and issue original RISC primitives to the execution pipeline. This approach is similar in spirit to our macro-op scheduling, and effective in reducing pressure in the scheduling logic as well as other portions of processor pipeline. The fundamental difference is, K7 MOPs or fused micro-ops are statically constructed from a single x86 instruction, and each micro-op is scheduled individually according to the readiness of corresponding source operands, as if multiple issue entries were allocated to them. The scheduling logic does not exploit fused operations to enable pipelined scheduling. In contrast, our approach dynamically constructs MOPs and issues multiple instructions as a single unit when all source operands are ready, enabling non-speculative pipelined scheduling logic that issues dependent instructions consecutively.

8. Conclusions

We make three main contributions in this work. First, we introduce a concept of scheduling granularity in instruction scheduling logic and describe its design space. Second, we characterize the how often instructions can be grouped and find that many instructions can be scheduled at a coarser level with relaxed atomicity and scalability constraints. Third, we propose macro-op scheduling that converts multiple instructions into a multi-cycle macro-op to achieve both pipelined scheduling logic and a bigger instruction window at the same time. We demonstrate that a machine with pipelined 2-cycle macro-op scheduling achieves comparable or even better performance than a machine of the same pipeline depth using atomic scheduling.

9. Acknowledgements

This work was supported in part by the National Science Foundation with grants CCR-0073440, CCR-0083126, EIA-0103670 and CCR-0133437, and generous financial support and equipment donations from IBM and Intel. We would like to thank the anonymous reviewers for their many valuable comments.

10. References

[1] E. Borch, E. Tune, S. Manne and J. Emer, Loose loops sink chips, in *Proc. of 8th International Symposium on High-performance computer architecture*, 2002.

[2] G. Hinton et al., The microarchitecture of the Pentium 4 processor, *Intel Technology Journal* Q1, 2001.

[3] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler and P. Shivakumar, The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays, in *Proc. of 29th International Symposium on Computer Architecture*, 2002.

[4] D. Ernst and T. Austin, Efficient dynamic scheduling through Tag Elimination, in *Proc. of 29th International Symposium on Computer Architecture*, 2002.

[5] I. Kim and M. H. Lipasti, Half-price architecture, in *Proc. of 30th International Symposium on Computer Architecture*, 2003.

[6] K. Diefendorff, K7 challenges Intel, *Microprocessor Report*, Vol. 12, No. 14, 1998.

[7] J. Stark, M. Brown and Y. Patt, On pipelining dynamic instruction scheduling logic, in *Proc. of 33th International Symposium on Microarchitecture*, 2000,

[8] M. Brown, J. Stark and Y. Patt, Select-free instruction scheduling logic, in *Proc. of 34th International Symposium on Microarchitecture*, 2001.

[9] S. Palacharla, N. P. Jouppi and J. E. Smith, Complexity-effective superscalar processors, in *Proc. of 24th International Symposium on Computer Architecture*, 1997.

[10] A. R. Lebeck et al, A large, fast instruction window for tolerating cache misses, in *Proc. of 29th International Symposium on Computer Architecture*, 2002.

[11] P. Michaud and A. Seznec, Data-flow prescheduling for large instruction windows in out-of-order processors, in *Proc. of 7th International Symposium on High Performance Computer Architecture*, 2001.

[12] M. Goshima et al., A high-speed dynamic instruction scheduling scheme for superscalar processors, in *Proc. of 34th International Symposium on Microarchitecture*, 2001.

[13] D. C. Burger and T. M. Austin, The Simplescalar tool set, version 2.0, Technical Report CS-TR-97-1342, University of Wisconsin, Madison, 1997.

[14] A. Kleinosowski, J. Flynn, N. Meares and D. J. Lilja, Adapting the SPEC2000 benchmarks suite for simulation-based computer architecture research, *Workshop on Workload Characterization in International Conference on Computer Design*, 2000.

[15] Compaq Computer Corporation, *Alpha 21264 microprocessor hardware reference manual*, 1999.

[16] H. Kim and J. E. Smith, An instruction set and microarchitecture for instruction level distributed processing, in *Proc. of International Symposium on Computer Architecture*, 2002.

[17] P. Y.-T. Hsu, J. T. Rahmeh, E. S. Davidson, and J. A. Abraham, TIDBITS: Speedup via time-delay bit-slicing in ALU design for VLSI technology, in *Proc. of 12th International Symposium on Computer Architecture*, 1985.

[18] S. Gochman et al., The Intel Pentium M processor: Microarchitecture and performance, *Intel Technology Journal* vol. 7, issue 2, 2003.

[19] R. Canal and A. Gonzalez, A low-complexity issue logic, in *Proc. of 14th International Conference on Supercomputing*, 2000.

[20] S. E. Raasch, N. L. Binkert and S. K. Reinhardt, A scalable instruction queue design using dependence chains, in *Proc. of 29th International Symposium on Computer Architecture*, 2002.

[21] Q. Jacobson and J. E. Smith, Instruction pre-processing in trace processors, in *Proc. of 5th International Symposium on High Performance Computer Architecture*, 1999

[22] Y. Chou and J. P. Shen, Instruction path coprocessors, in *Proc. of 27th International Symposium on Computer Architecture*, 2000.

[23] D. Friendly, S. Patel and Y. Patt, Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors, in *Proc. of 31st International Symposium on Microarchitecture*, 1998.

[24] N. Malik, R. Eickemeyer and S. Vassiliadis, Interlock collapsing ALU for increased instruction-level parallelism, in *Proc. of 25th International Symposium on Microarchitecture*, 1992.

[25] Y. Sazeides, S. Vassiliadis and J. E. Smith, The performance potential of data dependence speculation and collapsing, in *Proc. of 29th International Symposium on Microarchitecture*, 1996.