MACRO-OP SCHEDULING AND EXECUTION

by

Ilhyun Kim

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN-MADISON

2004

© Copyright by Ilhyun Kim 2004
All Rights Reserved

Abstract

High-performance microprocessors must attain two elusive goals that are often at odds with each other: high operating frequency that demands a minimum amount of logic per pipeline stage, and a high degree of concurrency in the form of instruction-level and memory-level parallelism, which tends to increase the amount of activity required to execute each instruction. The logic required to implement all of this complexity can be broken down into more and more pipeline stages to achieve higher operating frequency, but this comes at the cost of additional power, pipeline latch overhead, and erosion in IPC due to increased branch and scheduling penalties.

One promising approach to overcoming this limitation and simplifying the control logic overhead in an out-of-order processor is to move from the conventional instruction-level processing towards *coarse-grained instruction processing* that amortizes these overheads over a set of two or more instructions. This thesis proposes and evaluates two microarchitectural techniques that perform coarse-grained instruction processing in superscalar, out-of-order processors: *Macro-op Scheduling* and *Macro-op Execution*.

This thesis shows that coarse-grained instruction processing enabled by macro-op scheduling and execution is a good approach to achieving complexity-effective, high-performance superscalar, out-of-order processor designs. Three key points support this conclusion:

- *Many instructions do not require fine-grained, instruction-level controls.*
  Chains of dependent instructions do not require fine-grained, instruction-level controls driven by complex data dependences. Processing dependent instructions in groups does not sacrifice instruction-level parallelism and potentially provides greater opportunities for simplifying control logic overhead and reducing conten-

tion due to fewer units to process.

- *Pipelined macro-op scheduling performs similarly or better than conventional atomic scheduling.*

  Performing coarse-grained instruction scheduling on groups of multiple instructions enables the pipelined instruction scheduler to hide scheduling delays and to issue dependent instructions in back-to-back cycles. Also, it reduces issue queue contention and widens the out-of-order instruction window by handling multiple instructions as a single unit.

- *Macro-op execution increases machine bandwidth using similar or lower complexity hardware.*

  Macro-op execution enables multiple instructions to be managed as a single unit in the entire out-of-order execution pipeline. A single execution lane can handle multiple instructions simultaneously, which increases issue and execution bandwidth.

This thesis makes three main contributions. First, it presents the concept of coarse-grained instruction processing, which reduces the hardware overhead involved in coordinating all of the concurrent actions in a modern out-of-order superscalar processor. Second, it quantifies the degree of abstraction that can be applied to a program for coarse-grained instruction processing, and shows that a significant portion of the instructions can be processed together in groups without requiring fine-grained controls for scheduling and execution. Third, it demonstrates that the proposed techniques for coarse-grained instruction processing enable complexity and performance benefits in an out-of-order superscalar processor.

Acknowledgements

I would like to thank my parents, YongChi Kim and BokSoon Shin for their constant support throughout my whole life. Without them, I would not be what I am now. I am truly grateful for their being encouraging to me all the time. I also thank my family for their support. I am very grateful with my beloved wife, Hwakyu Lee, for encouraging and giving me strength to do what I had to do. She brought much love and warmth to my life.

I want to express my deep gratitude to my advisor, Professor Mikko Lipasti for giving me the opportunity to work with him and for his patience, tolerance, understanding, and encouragement. Not only about academic engineering research and how to find a way to a solution, he also taught me how to be a person who other people need. It has been a great pleasure working with him. I am very thankful to the members of my thesis committee, Prof. Jim Smith, Prof. Michael Shulte, Prof. David Wood, and Prof. Guri Sohi for their feedback on my thesis research and stimulating discussions throughout my graduate tenure.

A special thank goes to all members of the PHARM research group. Especially, I would like to thank Kevin Lepak, Trey Cain and Gordie Bell, who have been Mikko's first students. Kevin has always supported and motivated me in academic as well as non-academic ways. I thank Trey for his substantial efforts on group infrastructure and his persistence that I envy. I thank Gordie for many stimulating discussions and helps. I thank Brian Mestan, Razvan Cheveresan, and Matt Ramsay for their bringing joy to my monotonous life.

Finally, I want to thank to all those who helped me through the research. In particular, I thank the Department of Electrical Engineering staff for their assistance with

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

A compelling argument in favor of RISC design was that simpler instruction sets could be realized more efficiently, giving them an inherent performance advantage over complex instruction sets. Many current-generation microprocessors follow a similar philosophy in their hardware designs. Even in processor implementations for complex instruction sets such as IA-32, complex instructions are broken into micro-ops running on RISC-style cores. The principles of these designs can be summarized as *regularity*, *orthogonality* and *composability* [99], implying that a collection of homogeneous and atomic instruction primitives perform complex operations. Being faithful to this paradigm, instructions are usually designed to perform an indivisible amount of useful work that facilitates efficient hardware implementation with simple controls over instruction and data flow. Accordingly, the pipeline and many hardware structures in a microprocessor are also built to handle each instruction as a minimal processing unit. This traditional *instruction-centric* hardware design has been considered intuitive and reasonable, since execution eventually occurs at instruction boundaries where the architectural state should be preserved.

However, there is increasing demand for reconsidering the premise of instruction-centric hardware design as microprocessors become prohibitively complex. The complexity of modern superscalar, out-of-order processors comes not only from high operating frequency that demands more pipeline stages to facilitate reducing the gate count per stage, but also from the sophisticated control logic that is necessary to coordinate all of the con-

current actions of in-flight instructions in the pipeline. The logic required to implement all of this activity may be further broken down into multiple pieces to lower the complexity in each pipeline stage, but this comes at the cost of additional power, pipeline latch overhead, and erosion in IPC due to increased branch and scheduling penalties.

An instruction is a unit of execution, but may not necessarily be an optimal unit of processing in the pipeline given the technology trends and the difficulties in providing sophisticated controls over instructions. Rather, moving from instruction-level processing towards coarse-grained processing--where multiple instructions are tracked, scheduled and executed as a single entity--is desirable since such an approach potentially decreases the number of processing units in the pipeline and reduces logic overhead by reducing the rate at which the control decisions must be generated.

The purpose of this thesis is to explore the design space of superscalar, out-of-order microprocessors, to evaluate benefits when varying the granularity of the processing unit, and to expose a greater opportunity for high-performance and complexity-effective designs at a coarser processing level by relaxing the microarchitectural design constraints imposed by instruction-centric hardware designs. Specifically, the thesis proposes and evaluates microarchitectural techniques to process multiple operations as a single atomic unit in instruction scheduling logic and execution pipeline. The proposed technique called *macro-op scheduling* performs coarse-grained instruction scheduling to relax the atomicity and scalability constraints of conventional approaches, and enables pipelined instruction scheduling with a wider out-of-order execution window. The other proposed technique called *macro-op execution* performs coarse-grained instruction processing in the execution pipeline to increase the machine bandwidth with similar or lower hardware

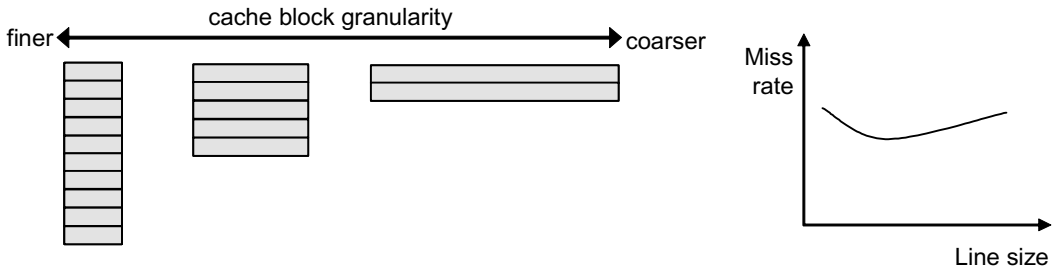


FIGURE 1-1. An analogy for processing granularity.

complexity.

1.1 Processing Granularity

Processing granularity is defined as the amount of work associated with a process. From hardware designer's perspective, it can be interpreted as the amount of *resource* managed by each individual *control*. A simple analogy is cache block granularity given a fixed cache size, in which various cache line sizes affect performance differently. In this case, control can correspond to line transfer from a lower level memory to this cache, and resource can correspond to the data size per cache line transfer.

Figure 1-1 illustrates a trend of cache miss rates for various cache line sizes [85]. At a finer granularity (the left cache configuration in the figure), the cache is configured to have fewer bits per line but the number of lines increases. As the cache block granularity increases toward a coarser level, the cache miss rate will decrease since a coarser granularity incurs fewer data transfers from lower level memory for a certain amount of data. Beyond a certain point as the granularity is further increased, however, the cache miss rate will bounce back because the effective cache size is decreased due to redundant data transferred.

We see that there are trade-offs between resource and control at different process-

ing granularities. As the processing granularity increases, fewer controls are required to perform a certain amount of work, while the resource management may be inefficient. Conversely, a finer processing granularity tends to increase the number of controls, but the resource management may become more efficient. In this spectrum of processing granularity, an optimal processing granularity to maximize benefits may be determined by the goals and constraints of the design. In the previous example of cache block granularity, an optimal point can be chosen to generate the lowest cache miss rate. If latency or power consumption is more critical, a different processing granularity may be chosen to meet such requirements.

1.2 Processing Granularity of Instructions

A conventional superscalar, out-of-order machine processes instructions at an instruction-level granularity, i.e. generating scheduling decisions at every instruction boundary. Also, hardware structures are configured to match an instruction's specifications so that each control enables instructions to access hardware resources, e.g. the register file, without further arbitration. From the perspective of performance, processing instructions at an instruction-level granularity may provide a high degree of instruction-level parallelism, since this enables fine-grained controls over the dataflow defined by a program in the form of register and memory dependences among instructions. Performing necessary tasks in each pipeline stage at the instruction-level granularity is a reasonable design decision because execution eventually occurs at instruction boundaries.

A potential limitation of processing instructions at the same granularity as they are executed is that many hardware parameters are automatically determined by such a granu-

larity, leaving little flexibility in the hardware design space. For example, although the number of ports to the register file may vary depending on the number or functional units or machine width, the number in each issue slot is fixed to the worst-case requirement of an instruction, e.g. two read and one write ports. Similarly, the atomicity constraint of conventional instruction scheduling--a set of wakeup and select operation should occur atomically every clock cycle--comes from the fact that the scheduler should generate scheduling decisions on a per instruction basis. In other words, instruction-granular processing imposes instruction-granular constraints, which often put significant pressures on the hardware structures that are likely to limit the processor's cycle time.

Each pipeline stage has different types of design issues. Some structures like the register file have resource-critical design issues, as the number of entries and ports determines their complexity. Some other structures, such as instruction scheduling logic, have control-critical design issues, i.e. completing certain tasks within a fixed number of cycles is crucial for ensuring a certain level of performance. If instructions are processed at other granularities coarser or finer than an instruction, the different granularity may compensate for the critical design issues incurred by instruction-granular processing. For instance, applying fine-grained, operand-centric processing may resolve resource-critical issues since resources can be more efficiently managed at the expense of added control complexity. For example, creating scheduling bubbles for sequentially accessing a single register ports improves the complexity of the register file, as proposed in [56].

1.3 Coarse-grained Instruction Processing

Between the two opposite ends in the spectrum of processing granularities, this

thesis research explores and evaluates the design space of coarse-grained instruction processing. This is motivated by the design trend of superscalar, out-of-order processors. High-performance microprocessor designs must attain two elusive goals that are often at odds with each other: high operating frequency that demands a minimum amount of logic per pipeline stage, and a high degree of concurrency in the form of instruction-level and memory-level parallelism, which tends to increase the amount of activity required to execute each instruction. Scaling the current generation processor designs to future performance level becomes challenging, since the complexity of control logic to manage all concurrent actions among instructions would not easily fit in the targeted cycle time without being pipelined over multiple stages and hence losing the capability of instantly reacting to dynamic behaviors [6][55]. To overcome this limitation, one promising approach to simplifying the control logic overhead in an out-of-order processor is to move from the conventional instruction-level processing towards coarse-grained instruction processing that amortizes these overheads over a set of two or more instructions. In our approach, this is enabled by grouping multiple instructions into a single unit that is tracked, scheduled and executed as an atomic entity.

This basic idea of coarse-grained instruction processing has been proposed or already realized in several ways. For example, designs like the Alpha 21264 [17] and the IBM POWER4 [71] accommodate basic-block size groups of instructions in each reorder buffer entry to reduce the complexity of the dispatch and commit logic. More recent designs like the AMD Athlon [21] and Intel Pentium M [39] also group some operations into fused operations in order to reduce the number of operations to process. To reduce the overhead involved in scheduling and improve execution bandwidth, many machines with

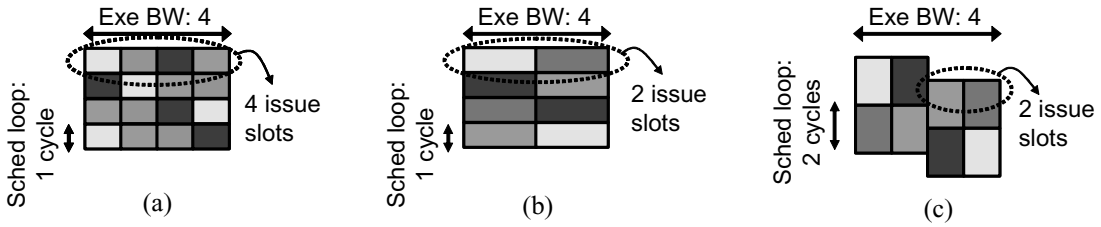


FIGURE 1-2. Approaches to coarse-grained instruction processing.

long instruction words [59][83][79][24] or SIMD vector operations [10][52][22] employ coarse-grained instruction processing by grouping multiple operations into a single schedulable unit--these are more closely related to what this thesis proposes.

One important aspect of coarse-grained instruction processing that distinguishes this thesis research from the previous efforts is the *direction of granularity*. The prior approaches group *independent* operations into each unit, whereas the proposed schemes primarily focus on *dependent* instructions to achieve coarse-grained processing. Figure 1-2 illustrates and highlights the differences among instruction processing models. The vertical direction in the figure implies time window (i.e. clock cycles) to perform a certain operation. The horizontal direction implies the machine bandwidth required. In conventional, instruction-level processing as in superscalar, out-of-order execution (Figure 1-2a), executing instructions requires the same degree of scheduling decisions (e.g. one cycle per scheduling decision) and resources (e.g. four issue slots). Coarse-grained *parallel* processing as in LIW or vector SIMD machines (similar to an out-of-order version of the Itanium processor [26]) (Figure 1-2b), where each unit contains two independent operations, reduces the number of schedulable units and issue slots to achieve the equivalent execution bandwidth, although it still needs the scheduling decisions to be generated at the same rate as the previous case. In contrast, the proposed approach for coarse-grained *serial* processing (Figure 1-2c) achieves the same benefit as coarse-grained parallel processing,

while it additionally enables finding the next issuable dependent operations at a slower rate.

1.4 Macro-op Scheduling and Execution

The thesis proposes and evaluates two microarchitectural techniques that perform coarse-grained instruction processing in superscalar, out-of-order processors: macro-op scheduling and execution. This is enabled by grouping multiple instructions into a single schedulable unit called *macro-op* (*MOP*), which defines the serial execution order of the instructions contained in it. Macro-ops are dynamically created by examining register dependences and grouping matching candidate instructions in a program.

Macro-op scheduling performs coarse-grained instruction scheduling. It performs non-speculative pipelined scheduling of coarser macro-ops with multi-cycle latencies while the processor core still executes the original dependent instructions consecutively. At the same time, the scheduler handles fewer units since the original, multiple instructions are processed and issued together. This reduces the issue queue contention and widens the instruction window. An issued macro-op is converted back to the original instructions, which are sequenced down to the conventional instruction-grained pipeline and executed serially without complicating many structures and data paths.

Macro-op execution extends the range of coarse-grained instruction processing to the entire set of out-of-order pipeline stages. Multiple original instructions in a macro-op are managed and processed together as a single unit throughout the entire execution pipeline. The instruction scheduler for macro-op execution is similar to that of macro-op scheduling, but is able to issue more instructions by eliminating the explicit sequencing

operations required for macro-op scheduling. The original instructions, grouped in macro-ops, perform the required work in each pipeline stage as a group until they reach the execution stage, in which stacked ALUs are configured to naturally sequence the macro-op instructions through the execution stages. This approach enables each execution pipeline to handle multiple instructions simultaneously.

1.5 Thesis Contributions and Outline

The thesis of my research is that coarse-grained instruction processing enabled by macro-op scheduling and execution is a good approach to achieving complexity-effective, high-performance machines that overcome the limitations of conventional superscalar, out-of-order processor designs. I defend this thesis by demonstrating three key points:

- *Many instructions do not require fine-grained, instruction-level controls.*

  Processing instructions at an instruction-level granularity may best optimize the dataflow defined by a program in the form of register and memory dependences among instructions. However, the way programs are written is serial in nature; chains of dependent instructions do not require fine-grained, instruction-level controls driven by complex data dependences. Processing dependent instructions in groups does not sacrifice instruction-level parallelism and potentially provides greater opportunities for reducing contention due to fewer units to process.

- *Pipelined macro-op scheduling performs similarly or better than conventional atomic scheduling.*

  Instruction scheduling logic is a major obstacle to building high-frequency microprocessors since wakeup and select operations are not easily pipelined. When the

conventional scheduler is naively pipelined, it loses the ability of issuing dependent instructions consecutively since extra delays exceeding the instruction's execution latency are induced. Performing pipelined scheduling of multi-cycle latency macro-ops enables the scheduler to hide those extra delays and to issue dependent instructions in back-to-back cycles. Also, macro-ops contain multiple original instructions, and hence reduce the issue queue contention and widen the window.

- *Macro-op execution increases bandwidth with similar or lower complexity hardware.*

  In addition to the benefits of macro-op scheduling, i.e. pipelined instruction scheduling logic and a wider instruction window, macro-op execution increases the machine bandwidth through the entire execution pipeline including issue, dispatch, register file, and bypass logic. The same number of issue slots can handle more instructions by issuing them in groups. Dispatching a macro-op in effect processes multiple instructions. The same number of read ports to the register file supports more instructions since the register dependence that links grouped instructions does not require a register read access. The execution stages need only partial bypass network since not all instructions can be issued consecutively.

To support these points, an elaborate study on the dependence structure of the program that measures the potential for processing instructions at a coarse granularity is first performed. Based on the characteristics of the programs measured by this study, I narrow the design space of the microarchitectural techniques to enable coarse-grained instruction processing and determine the policies for grouping instructions.

Before I apply coarse-grained instruction processing to the instruction scheduling logic, the performance impact of pipelined instruction scheduling and the reasons for different performance sensitivities of programs to pipelined scheduling are studied. Then, the details of macro-op scheduling, its key enablers, and performance and implementation issues are discussed. The proposed microarchitecture for macro-op scheduling, its performance benefits, and trade-offs in its design space are tested and evaluated with extensive simulations.

The studies on macro-op execution follow the steps similar to those of macro-op scheduling; after the basic concept, potential benefit and the detailed microarchitecture is discussed, I test and evaluate the potential of macro-op execution.

The rest of the thesis is organized as follows. Chapter 2 discusses the previous work related to this research. Chapter 3 describes the experimental framework. Chapter 4 defines a macro-op, describes its implications and potential impacts on program and microarchitecture, and measures the groupability of instructions. Chapter 5 analyzes the effects of pipelined instruction scheduling. Chapter 6 details the microarchitecture for macro-op scheduling. Chapter 7 evaluates macro-op scheduling. Chapter 8 presents macro-op execution and its microarchitecture. Chapter 9 evaluates macro-op execution. Finally, Chapter 10 provides conclusions of this thesis and directions for future work.

Chapter 2

Related Work

This chapter outlines previous research that is closely related to the macro-op scheduling and execution described in this thesis. This chapter first discusses related work on processing instructions at a coarse-grained level or across multiple instruction boundaries. It also discusses related work on preprocessing and transforming instructions into other forms to achieve performance and / or complexity benefits. Then, it describes related studies on reducing complexity in many hardware structures that are likely to be a bottleneck in current and future microprocessors, such as instruction scheduling logic, register file, and bypass logic.

2.1 Coarse-grained Instruction Processing

Macro-op scheduling and execution performs coarse-grained instruction processing to reduce the complexity involved in managing instructions in superscalar, out-of-order processors. There are numerous proposals to exploit coarse-grained instruction processing for either reduced hardware complexity or improved performance.

Melvin, Shebanow and Patt [67] discussed the issues in atomic unit sizes and outlined the implementation of a front end to construct large atomic units. Their observation is that larger architectural atomic units potentially decrease the size of the architectural state of the machine and simplify the design of high performance machines due to fewer architectural boundaries to preserve. They advocated larger atomic units for higher performance and simpler controls. Although this work made important observations on the ben-

efit of coarse-grained processing that potentially simplifies hardware design, it discussed neither the direction of granularity (i.e. serial or parallel operations in an atomic execution unit) nor the benefits of coarse-grained processing units consisting of serial operations. Extending this basic paradigm, Melvin and Patt [68] proposed combining software or compiler techniques with dynamically scheduled processors to extract more instruction-level parallelism from the atomic units coarser than individual instructions. The proposed ISA, called *block-structured ISA,* packs multiple instructions into a coarse atomic block and enables the machine to track fewer architectural states across blocks since only results that are live upon exit of an atomic block update the architectural states. It also enables placing instructions within an atomic block in an arbitrary order (as opposed to a sequential order) to improve dynamic instruction scheduling by widening the instruction supply bottleneck. The fundamental difference from this thesis research is that the instructions within an atomic block are scheduled and processed dynamically and individually, which still require fine-grained controls over instructions. In contrast, macro-op scheduling and execution performs a coarse-grained control over a set of multiple instructions. Their approach focuses on reducing the number of *architectural* states to improve performance, whereas our approach focuses on reducing *non-architected* states (e.g. reducing the number of schedulable units) to lower control logic complexity.

There have been numerous efforts to process a set of computations in a group in order to minimize the communication cost across multiple groups and to maximize the processor utilization. In the area of dataflow architectures, the Monsoon processor by Papadopoulos and Culler [75] exploits both fine-grained parallism within an *activation frame* that localizes a set of computations within a processing element, and coarse-grained paral-

lelism across different processing elements with low network traffic. The TRIPS architecture by Sankaralingam et al. [72][81] aggregates a group of instructions into an atomic hyperblock that is mapped to each dataflow-based processor core while coarse-grained parallelism is exploited by executing multiple hyperblocks. Work in the area of Multiscalar processors by Franklin and Sohi [35][36][37], and Sohi, Breach, and Vijaykumar [7][86] adopts the expandable split window paradigm that considers a window of instructions as a single unit, and exploits both fine-grained parallelism within a window and coarse-grained parallelism across multiple windows by overlapping the execution of them. The proposed processor is composed of a collection of multiple sequential processors, which execute multiple blocks of instructions in parallel starting from multiple different points in a sequentially written program. A *activation frame* in Monsoon processors and a *task* in Multiscalar processors can be analogous to the coarse-grained schedulable unit called *macro-op* in this thesis, although a macro-op is significantly finer and focuses on register-level communication within a processor core.

There is a long and rich history of coarse-grained instruction set designs for improved processing bandwidth with reduced complexity, ranging from classical vector architectures, to long instruction word architectures, to SIMD instruction sets that have been adopted as multimedia extensions to many current-generation ISAs [80][90][15][33][34][10][52][22][59][83][63][79][24]. Since there are numerous proposals and working examples for these approaches, I do not discuss in detail the pros and cons of such parallel instruction set designs. However, an important aspect of the approach presented here, which sets it apart from most of previous work, is that macro-op execution exploits natural dependences of programs to extract parallelism whereas other approaches

primarily search for independent operations to pack into a coarser processing unit[1].

As related work on exploiting serial operations, there was an unimplemented design for a Cray Research processor [18] that packs multiple dependent operations with an implicit accumulator to chain them together. H. Kim and Smith [53] rearchitected the implementation set to more explicitly identify dependent instructions sequences with a shared accumulator register. This enables hardware that can efficiently dispatch dependent instructions to logically independent datapaths, reducing the overhead of coordination and synchronization across these distributed execution pipelines and leading to complexity-effective, high-performance designs. There are also numerous proposals for dependence-based clustered microarchitectures and techniques for complexity-effective instruction scheduling, register file and bypass logic. These will be discussed later in their corresponding sections.

From the viewpoint of packing multiple primitive operations into a coarser schedulable unit and performing schedule and execution of macro-ops, the proposed microarchitecture employs a counter approach to recent x86 processor implementations that crack a CISC instruction and convert it into multiple RISC semantics running on RISC-style cores [21][48][42]. Although I do not propose a new instruction set architecture nor advocate one design strategy over the other, the proposed approach may be closely related to the endless and recurring debates about whether CISC or RISC is better, due to the similarities between CISC instructions and macro-ops. A recent and relevant debate about the design philosophy between the two approaches took place in mid 90's, initiated by the Cyrix M1 processor [66] that performs "native" execution of complex instructions on a

1. Some vector or SIMD operations such as the dot product operation require a serise of dependent computations among the elements contained in a single coarse-grained unit.

superscalar out-of-order core specialized for the x86 instruction set, as opposed to the "RISC-like" approach employed in other x86 implementations by its competitors. Bluhm and Garibay [5], advocating their native approach in the M1, argued that a RISC-like approach may require more communication in the data path by creating more boundaries to preserve, whereas the native approach enables only the architecturally-visible results to be communicated across different functional units, achieving lower complexity and higher performance--these are also the advantages of the coarse-grained instruction processing.

Despite their similarities, a fundamental difference between the proposed approach and native execution of CISC instructions lies in the attributes of coarse-grained schedulable units (either macro-ops or CISC instructions) designed for different purposes. Historically, one of the purposes of CISC was to reduce the semantic gap between high-level language operations and machine language primitives [23]. Each instruction is designed with respect to high code density and ease of programming, resulting in powerful but complex operations that are often unsuitable for efficient hardware implementations. In contrast, decisions on which instructions are grouped into macro-ops are based on the microarchitectural benefits in terms of complexity and performance of the underlying hardware. Specifically, for complexity, our approach restricts the number of operations, the number of source operands and the types of operations in macro-ops so that maintaining regularity among macro-ops enable hardware resources to be efficiently utilized. For performance, a single-cycle operation is placed as a macro-op head so that pipelined instruction scheduling can achieve compelling performance of conventional atomic scheduling. The details of macro-op grouping will be discussed in Chapter 6 and Chapter 8.

Two hardware implementations that process multiple instructions at a coarser

granularity--the AMD K7 [21] and the Intel Pentium M [39]--fuse multiple RISC ops or micro-ops to reduce the number of units to be processed. Instead of breaking a complex x86 instruction into multiple micro-ops, the decoder generates a single, coarse-grained unit that contains multiple RISC operation semantics. This approach reduces the queue occupancy in many pipeline stages including instruction scheduler due to fewer micro-ops to process. In the instruction scheduler, this fused micro-op is broken into multiple RISC operations and sent down to the execution pipelines according to the readiness of the corresponding source operands. The fundamental difference from our work is that the original operations are scheduled and processed individually and hence execution of a coarser unit is interruptible (i.e. partially executing one operation and delaying others in a coarse-grained unit is allowed). In contrast, our approach forces uninterruptible and deterministic execution within a macro-op and therefore individual instructions do not require fine-grained controls for scheduling and execution, which reduces the rate of controls that must be generated. To reduce complexity involved in tracking architectural states at multiple instructions boundaries, designs like the Alpha 21264 [17] and the IBM POWER4 [71] accommodate basic-block size groups of instructions in each reorder buffer entry to reduce the complexity of the dispatch and commit logic.

Interlock collapsing or *dependence collapsing* techniques [64][82][38][50][98] merge a series of dependent instructions into one single-cycle operation with more operands, reducing execution latency. In a sense, the proposed macro-op scheduling is a scheduler-side collapsing technique that exploits a similar grouping process to improve scheduling latency rather than execution latency itself. Macro-op scheduling alters only the dependence mapping in the scheduler and unmodified original instructions are exe-

cuted; hence, it requires no changes in the datapath (e.g. special ALUs or 3-source register read ports), nor any special handling of the intermediate results that other dependent instructions may consume. Macro-op execution may require changes in the register file to support three or more source operands if a macro-op allows more than two source operands. However, the total number of register read ports is lower than what is required to service the same number of instructions individually. These issues are studied in detail in Chapter 8.

2.2 Instruction Preprocessing and Transformation

Macro-op scheduling and execution examines register dependences among instructions and generates macro-op pointers, which later direct macro-op formation process to transform multiple instructions into coarse-grained macro-ops. There are numerous proposals for preprocessing instructions and transforming them into other forms to improve performance and / or reduce hardware complexity.

Jacobson and Smith [50], Friendly, Patel and Patt [38] proposed transforming instructions based on peephole optimizations during trace cache line construction to implement better instruction scheduling, constant propagation, instruction or dependence collapsing and many other optimizations. Chou and Shen [16] proposed the *instruction path co-processor*, which is a programmable internal processor that operates on instructions of the core processor to transform them into a more efficient stream, and showed the performance gain from similar optimizations. I. Kim and Lipasti [55] proposed *speculative decode* that also transforms instructions into a different instruction stream in order to overcome the limitations of value-based dynamic optimizations under scheduling latency

A translation layer often exists to bridge the gap between the user-visible architected instructions set and a realizable implementation instruction set. A number of working examples that perform translation between them, ranging from trap-based translators where unimplemented instructions are emulated in the operating system's invalid instruction exception handler (e.g. Micro VAX or PowerPC 604 [47]); to microcoded emulation routines stored in an on-chip lookup table (e.g. the Intel Pentium Pro [48] and its derivatives, and the IBM POWER4 [71]); to binary translation approaches [25] like the Transmeta Crusoe processor [59] and the IBM DAISY [24], which perform a dynamic binary translation with supports of virtual machines that translate instructions written in their user ISAs to an internal VLIW-style instruction set.

Regarding the work on transforming instructions into other sequence considering data dependences, work on *instruction-level distributed processing* by H. Kim and Smith [53][54] proposed an instruction set and its co-designed microarchitecture based on dynamic binary translation. Vajapeyam, Joseph and Mitra proposed a scheme for dynamically vectorizing ordinary programs based on trace processors [97]. Pajuelo, Gonzalez and Valero [73] proposed an approach to exploiting SIMD parallelism by dynamically generating SIMD instructions from regular binaries. An instruction set and its translation approach that directly supports macro-op scheduling and execution in this thesis was proposed by Hu and Smith [46]. It *fuses* dependent instructions using a software-based translator running on a co-designed virtual machine. This approach removes considerable complexity from the hardware and enables more sophisticated heuristics for forming macro-ops.

2.3 Complexity-effective Instruction Scheduling Logic

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

Palacharla, Jouppi and Smith [74] proposed the use of clusters to distribute the scheduling window and data paths to alleviate the impact of high operating frequency by introducing delay for inter-cluster communications. Specifically for instruction scheduling logic, their approach enables a collection of small instruction windows to work as a wider and deeper instruction window. Instruction steering logic examines data dependences among instructions and forces chains of dependent instructions to be assigned to the same cluster. A FIFO-style instruction queue in each cluster only checks the readiness of instructions located at the head of FIFOs, eliminating expensive hardware mechanisms for wakeup and broadcast operations and enabling simpler and faster instruction scheduling logic.

Canal and Gonzalez [13], Michaud and Seznec [69], and Raasch, Binkert and Reinhardt [77] proposed several schemes for data-flow based scheduling that reorders instructions before they enter a small issue window. These approaches statically or quasi-statically estimate dynamic issue timing of instructions considering data dependences, and dispatch instructions into a small dynamic instruction window only when instructions become soon to be issued. These approaches focus a demonstrably expensive hardware for dynamic instruction scheduling to only instructions that are likely to be executed, achieving an effectively larger instruction queue structures.

Brekelbaum, Rupley, Wilkerson and Black [8] proposed the use of hierarchical

instruction schedulers to achieve a large and deeper instruction window. This technique exploits the observation that a significant portion of instructions is latency tolerant and needs not be executed as fast as possible. Initially, all instructions are inserted into a large and slow instruction window. As instructions are dequeued from the large instruction window, they are dispatched into either a small and fast window that supports fast consecutive execution of dependent instructions, or a slow cluster that executes latency tolerant instruction at a slower rate. By explicitly managing latency-tolerant instructions in a separate cluster, the proposed microarchitecture can focus expensive hardware for schedule and execution on the small number of instructions, leading to lower power consumption and faster clock cycles at the expense of marginal performance degradation.

Lebeck et al. [61] studied the effect of cache misses on the instruction window and explored scheduler designs that re-insert chains of dependent instructions after cache misses are resolved. Draining the instruction queue by removing instructions dependent on cache misses can increase the effective size of the instruction window by reallocating issue entries to other newer instructions, overlapping long-latency dynamic events such as cache misses to the lowest level of the memory system.

Hrishikesh et al. [43] proposed a segmented instruction window in which each segment has a different scheduling priority. They assumed a shifting or collapsing instruction queue structure as underlying hardware, which fills the empty queue entries created by issued instructions with other unissued instructions, and maintains the original program order among instructions in the instruction queue. Since the older instructions in program order that are likely to be issued can be found in a physically small section of the window in this configuration, broadcasting instruction tags can be pipelined to reach different seg-

ments with different priorities, hence potentially improving the cycle time of the scheduling logic.

To break the atomicity of conventional instruction scheduling, Stark, Brown and Patt [88] described *speculative wakeup* to stretch the wakeup and select operations over two cycles. In this technique, each instruction keeps track of readiness of indirectly antecedent instructions by two levels (*grandparent* instructions), as well as directly antecedent instructions (*parent* instructions) as in conventional instruction scheduling logic. When an instruction detects grandparent instruction become ready by monitoring the wakeup bus activities, the instruction is speculatively scheduled in the following clock cycle so that wakeup operations are overlapped with select phases and dependent instructions can be consecutively issued by pipelined instruction scheduling logic with separate wakeup and select stages.

Brown, Stark and Patt [9] extended and generalized the prior approach for pipelining instruction scheduling logic over multiple pipeline stages. This *select-free* instruction scheduling moves the select phase of scheduling out of the single-cycle instruction scheduling loop. The key observation behind this proposal is that the number of ready instructions hardly exceeds the issue bandwidth. When all source operands of an instruction become ready, scheduling logic speculatively broadcasts its own tag through the wakeup bus without checking structural hazards among the instructions issued in each clock cycle. Then the select operation is performed on the speculatively issued instructions outside the instruction scheduling loop and a recovery mechanism later cancels mis-scheduled instructions when structural hazards are detected. As in [88], this approach allows instruction scheduling logic to be pipelined without eroding the instruction-level parallelism as

long as there are not many structural hazards among instructions issued in each cycle.

Ernst and Austin [27] proposed *tag elimination*, a combined scheme that uses specialized windows and last-tag speculation to achieve wakeup logic cycle time improvement by reducing load capacitance on the wakeup bus. Their observation is that not many instructions have two source operands that the instruction set architecture defines. For the instructions with zero or one source operand, specialized instruction queue entries with zero or only one tag matching logic are allocated. For the instructions with two source operands that require two tag comparators, a last-tag predictor speculates which operand is the one that triggers the actual issue, eliminating half of the tag comparators from the wakeup bus. The combined benefits are reduced load capacitance on the wakeup bus achieved by 1) reducing the number of tag comparators wired to the wakeup bus and 2) shortening the length of the wakeup bus, since not all instructions queue entries need to monitor the wakeup bus activities. In the cases where the last tag that triggers instruction issue is mis-speculated and instructions are incorrectly scheduled even before all source operands become ready, a scoreboard checks the correctness of the scheduling and recovers from the mis-speculation, as similarly in load latency-related replay. In a similar vein, I. Kim and Lipasti proposed *sequential wakeup* [56], which predicts last-arriving operands of instructions with two source operands, and sequentially wakes up the first and the other half of source operands in two consecutive clock cycles, reducing the load capacitance on the wakeup bus. A major difference between this and tag elimination scheme is that sequential wakeup is nonspeculative, and it incurs lower scheduling penalties for the cases in which last-tag speculation is incorrect or both source operands should be awakened simultaneously.

Ernst, Hamel and Austin [28] proposed *Cyclone*, which is composed of a static instruction scheduler and timed queue structures that circulate instructions through execution pipelines. When new instructions enter the out-of-order window, they are inserted into the timed queue entries according to issue timings estimated by the static instruction scheduling logic that manages time delays among instructions. Instructions in the timed queue are shifted to the next entries every clock cycle. When they reach the end of the structure, instructions are dequeued and issued. After instructions are executed, correctness of execution is checked by a scoreboard integrated in the register file and bypass network, which triggers replaying mis-issued instructions by re-inserting them again back to the timed queue, recirculating instructions in the pipeline until they hit in the scoreboard and correctly complete. In a similar context, Hu, Vijaykrishnan and Irwin [45] proposed wakeup-free instruction scheduling.

Most of these studies try to overcome scalability and atomicity constraints in isolation. In contrast, the work in this thesis explores the scheduler design space at a coarser level with a consistent view to those problems, relaxing both constraints simultaneously.

2.4 Reducing Complexity in the Register File

Macro-op execution enables the same number of register read ports as the conventional case to service more instructions, potentially reducing the total number of register read ports.

Many researchers including Black and Shen [4], Tremblay, Joy and Shin [92], and Farkas, Jouppi and Chow [29] studied the complexity of register file and found that the area of a register file increases quadratically and the latency increases approximately lin-

early as the number of ports grows. There have been numerous researches on reducing complexity in the register file and improving its cycle time. This section will divide them into two categories and briefly discuss them. One category is to reduce the total number of entries in the register file by either efficiently managing them, or caching a few entries into a small and separate structure, which can be directly accessed by the execution core with low latency. The other category is to reduce the number of ports to the register file by exploiting the fact that the register port requirements are relatively low since a significant portion of value communications occurs through the bypass network.

2.4.1 Reducing the number of entries

The Cray-I has two sets of two-level register files to improve the cycle time of the register file structure [80]. This machine needs compiler support to explicitly move values between different levels of the register file.

Cruz, Gonzalez, Valero and Topham [19] and Balasubramonian et al. [3] studied two-level hierarchical organizations with various caching policies that enable the smaller register file to be accessed with lower latency than required for a unified structure. The basic concept of hierarchical register file is similar to cache hierarchy that exploits locality in memory accesses. The upper-level portion of the register file is small and highly multi-ported to support the full execution bandwidth. On the other hand, the lower-level register file design is slow but large enough to support all in-flight instructions in the machine. Depending on the register cache management policy, the performance impact due to upper-level register misses can be minimized while the processor can operate at a higher clock frequency.

From a different perspective, there have been efforts for reducing the number of

register entries below what is required for supporting all in-flight instructions in the pipeline. Monreal, Gonzales, Valero, Gonzalez and Vinals [40][70] proposed *virtual-physical registers*, which reduces the size of the register file by delaying the allocation of actual data storage to the physical registers until instructions become ready to execute. This technique is based on the fact that the lifetime of a register value generated by executing an instruction is usually shorter than the period from allocation to deallocation of a physical register. Lipasti, Mestan and Gunadi proposed *physical register inlining* [62], which exploits the portion of the register lifetime from execution to deallocation. Since many register values can commonly be expressed using fewer bits than the physical register identifiers, the register identifiers can be directly used to carry such values and physical register entries can be conserved.

Jourdan, Ronen, Bekerman, Shomar and Yoaz [51] extended the basic concept of virtual-physical registers and exploited value locality in register values. They introduced physical register reuse and multiple-to-one mappings in the register rename table in order to reduce the size of the data storage in the physical register file. In a similar context, Balakrishnan and Sohi [2] refined the previous approaches and optimized value-locality-based register file design to focus on commonly used values, enabling a simpler implementation.

Borch, Tune, Manne and Emer [6] studied the effects of the load scheduling resolution loop and found that its pipeline depth impacts on performance greater than other portion of the pipeline. They proposed the *distributed register algorithm* as a way of reducing the load resolution loop length, which manages a small register cache to reduce the schedule-to-execution latency.

2.4.2 Reducing the number of ports

Balasubramonian et al. [3] and Park, Powell and Vijaykumar [76] proposed techniques to reduce the number of register ports. The fundamental observation behind their approaches is that the actual register read or write port requirements are usually lower than the machine execution bandwidth since not all instructions have the maximum number of source and destination operands that the ISA defines, and also many value communications are performed through the bypass paths in the out-of-order execution window. For read ports, their approaches place a fully- or partially-connected crossbar between execution slots and register ports. In order to multiplex a limited number of register ports, they require an arbitration mechanism that counts register read port utilizations either at schedule time or in a separate arbitration stage and directs instructions to access the assigned ports. If the requests exceed the read port bandwidth, the current schedule is canceled and instructions are replayed when read port conflicts are resolved. Park [76] also proposed a scheme to reduce the number of write ports to the register file. The proposed macro-op execution in this thesis in spirit shares the same observation as the previous proposals, and forces source operands to read values off the bypass network, reducing the number of register read ports. However, macro-op execution does not require any crossbar nor port arbitration; the macro-op detection logic can be configured to generate macro-ops with fewer source operands than those of the original individual instructions.

A real processor implementation, the Alpha 21264 [17] uses duplicated register files to reduce the number of read ports per each copy of the register file. A similar approch can be found in VLIW processors such as the TI C62x processor [89] that uses clustered register files. Our approach is orthogonal to such a replicated register file and can be applied in conjunction with other complexity-effective register file techniques.

2.5 Bypass Logic Complexity and Clustered Microarchitecture

Many proposals for reducing bypass logic complexity are closely related to clustering the machine based on data dependences, since this approach minimizes global communication across independent instructions and also maximizes local communication within a cluster for a series of dependent instructions. Bypass logic complexity is reduced by macro-op execution since chains of dependent instructions are grouped into macro-ops and value communication within a macro-op occurs locally through a specialized bypass path.

The PEWs architecture by Ranganathan and Franklin [78], and the dependence-based instruction steering by Palacharla, Jouppi and Smith [74], and the Multicluster architecture by Farkas, Chow, Jouppi and Vranesic [30] have dependence-based clusters to reduce hardware complexity in instruction scheduling and bypass logic. In a similar context, the ILDP by H. Kim and Smith [53] is an effort of improving the physical communication locality. The Alpha 21264 [17] has a clustered microarchitecture with a simpler bypass network.

A distinctive aspect of the bypass logic used for macro-op execution is that, in a sense, macro-op execution can be interpreted as vertically clustering (or horizontally slicing across dependence chains) the datapaths and bypass paths, whereas other clustered microarchitectures horizontally cluster the machine along with the chains of dependent instructions. This comes from the fact that globally visible values communicated through bypass paths can be generated only by the tail instruction grouped in a macro-op. Other values generated by head instructions of macro-ops are communicated through either local bypass paths or the register file. This approach is orthogonal to the conventional horizon-

tal clustering along with dependence chains, and they can be used in combination.

Chapter 3

Experimental Framework

This chapter describes the experimental framework used in this thesis. First, the base processor microarchitecture assumed in the thesis research is described, which is a current generation, conventional superscalar out-of-order processor. Then, this chapter discusses the simulator models and the types of experiments for which they are used. Finally, the suite of benchmarks, input sets, and their execution characteristics measured on the baseline timing simulator are presented.

3.1 Base Processor Microarchitecture

The pipeline and the microarchitecture of the processor assumed in this thesis research is a conventional, current-generation superscalar out-of-order processor.

3.1.1 Pipeline overview

Figure 3-1 presents an overview of the processor pipeline. The pipeline consists of the following stages: Fetch, Decode, Rename (2), Queue, Schedule, Dispatch (2), Register File Read (2), Execute, Writeback and Commit. The in-order portion (from fetch to queue stages) of the processor pipeline is similar to the Alpha 21264 [17], with an additional stage in register renaming. The out-of-order portion (from schedule to writeback stages) of the processor pipeline was modeled after the integer pipeline of the Pentium 4 [42]. In the instruction fetch stage, the branch prediction mechanism generates the next target PC address and accesses the instruction cache to bring instructions into the pipeline. Decode logic decodes instruction words and generates necessary control signals depending on

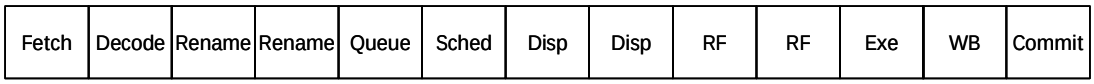


FIGURE 3-1. Processor pipeline.

instruction types. In the rename stage, the source and target register identifiers are renamed and physical registers are assigned to remove false dependences. Since the register map table is accessed a few clock cycles before instructions enter the out-of-order window, they need to check the most recent ready status of source operands in the queue stage, and are inserted into free issue queue entries. At the same time, this stage allocates the reorder buffer (*ROB*) entries to the instructions so that the correct architectural state is maintained in the event of pipeline flushing for branch misprediction or precise exception handling. In the scheduling stage, a set of wakeup and select operations links data dependences among instructions and speculatively (in terms of assuming that loads will hit in the cache) issues instructions down to the pipeline. In order to reduce the scheduling logic complexity, the payload RAM [9] is located next to the scheduling stage (in the dispatch stage) and the actual register identifiers and other necessary information are accessed from this separate structure. At the same time, issued instructions are dispatched to the execution pipeline. In the register file read stage, the register operand values are obtained from the physical register file. If the source operand values have not been written back and are unavailable in the physical register file, they will be read off the bypass network. Functional units in the execution stage realize the semantics of instructions. For memory operations, a load instruction that finishes its address generation proceeds to the memory stage (not shown in the figure), which is located next to the execute stage. After an instruction completes execution, its result value is written back to the physical register file. Finally,

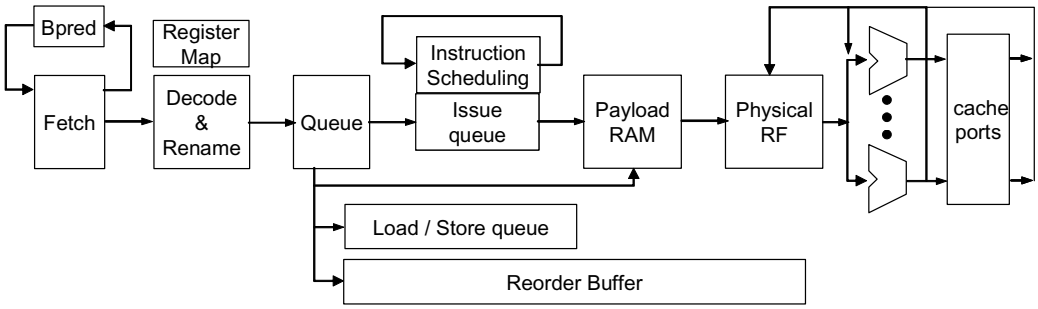


FIGURE 3-2. An overview of the microarchitecture modeled.

instructions update architectural state (register and memory) with their result values in the commit stage in program order when they become the head of the ROB. ROB entries are also released at this point and become available to other newly fetched instructions. An overview of the microarchitecture modeled in the timing simulator is illustrated in Figure 3-2.

3.1.2 Instruction scheduling logic

The function of instruction 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.

This thesis studies macro-op scheduling and execution built on two different styles of wakeup logic arrays: conventional CAM-style and wired-OR-style [74][9]. 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 against 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, which selects ready instructions to issue considering the available resources (i.e. functional units and memory ports) 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 operation of wired-OR-style wakeup logic is similar to that 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.

3.1.3 Speculative scheduling and replay

Out-of-order processors are built based on Tomasulo's algorithm [91], in which instructions that finish execution wake up their dependent instructions and scheduling logic selects issue candidates from the pool of ready instructions. In recent physical-register-based processor designs, the number of pipeline stages between instruction scheduling and execution has increased to accommodate the latency needed for reading the register

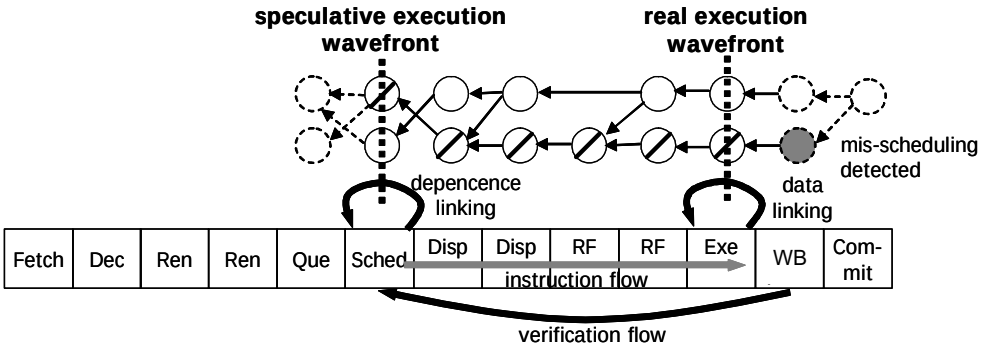


FIGURE 3-3. Speculative scheduling and replay.

file and performing other bookkeeping duties. As instruction scheduling and execution stages are separated, a naive implementation (e.g. based on the original Tomasulo's algorithm) fails to achieve maximum ILP because back-to-back execution of dependent instructions is no longer possible. To address this problem, current-generation processor implementations [17][42][96] use speculative scheduling in which the instruction scheduler speculatively wakes up and selects dependent instructions several clock cycles before the actual execution.

This process is illustrated in Figure 3-3. In the schedule stage, the wakeup and select operations link data dependences among instructions, and initiate the *speculative execution wavefront* [84][57]. This speculative image of execution is projected to the execution stage and drives the actual instruction execution, creating the *real execution wavefront*. Since load latency is not deterministic, instructions dependent on loads are scheduled assuming the common case cache hit latency. If load instructions incur dynamic events (e.g. cache misses or memory dependences) unexpected at schedule time, the actual instruction execution may diverge from the scheduled execution. In this scheduling miss case, the current execution of load-dependent instructions are canceled and will be replayed with correct inputs after the mis-scheduling condition is resolved. If the schedul-

ing is correct and instructions are successfully executed, they become ready for commit so that the processor's critical resources, i.e. the issue queue entries, can be reclaimed or a branch misprediction can be resolved earlier without waiting to reach commit.

3.1.3.1 Source of scheduling misses

Instruction scheduling resolves both data dependences and structural dependences. Since scheduling is ultimately converted into timing decisions (i.e. time slot allocated for a certain resource), a scheduling miss occurs when the actual execution diverges from the speculated order of execution due to dynamic changes in execution latency. This is primarily caused by load instructions with variable execution latency. If the actual load latency is shorter than what was assumed, this does not create any problem although the real execution may lose some opportunities to reduce the execution time or to save resource uses. However, if the actual latency is longer than expected, this should trigger replay operations because dependent instructions cannot get correct load values in time when they reach the execution stage. Some processor implementations [42] issue load instructions assuming that they hit in the first level cache because this is the common case. Alternatively, load latency predictions as in Alpha 21264 can be used to avoid frequent scheduling misses by conservatively issuing instructions with respect to longer load latency (e.g. L2 access latency) [17][101].

These are the conditions in which load instructions cause scheduling misses:

- Cache miss: this case also includes resource conflicts (e.g. cache bank conflict [101]) in the cache / memory system
- Data TLB miss
- Store-to-load memory dependence: Since memory dependence is dynamically

detected during execution, a load may not complete with a predetermined fixed latency if a memory aliasing is detected but the store data is not available, or if a memory aliasing cannot be determined since preceding store addresses have not been computed.

Scheduling misses may also be caused by many speculative techniques for performance or complexity optimizations but such techniques are beyond the scope of this thesis research and will not be discussed.

3.1.3.2 Scheduling replay

As mentioned in the previous sections, the purpose scheduling replay is to cancel the execution of incorrectly issued instructions and to reschedule them when mis-scheduling condition is resolved and correct source values become available. Although there are many possible implementations of scheduling replay mechanisms, the base machine model assumed in this thesis uses an issue-queue-based, position-based selective replay scheme [57].

The issue-queue-based replay is similar to the one used in the Alpha 21264 [17], where an instruction can leave the issue queue only after it is correctly executed. The shortcoming of this approach is that many issue queue entries can be unnecessarily held by correctly scheduled instructions, which reduces the effective size of the window. However, issued instructions can monitor all wakeup and rescheduling activities, which enables instructions to be dynamically scheduled after replay events are detected. In order to track data dependences among instructions and terminate incorrect speculative execution wavefront propagation, each issue queue entry has a dependence matrix in which each bit represents the two-dimensional position of a direct and indirect parent load instruction

in the pipeline. When a load mis-scheduling is detected, all instructions dependent on the load are selectively invalidated and replayed based on the information stored in dependence matrices. All other instructions independent on the load are unaffected by the rescheduling event. The details of issue-queue-based, position-based selective replay mechanism are presented in [57].

3.1.4 Memory disambiguation

In order to resolve memory dependences among instructions in the pipeline, the base microarchitecture uses a load / store queue (LSQ), which tracks readiness and effective addresses of load and store instructions. When a load or store instruction enters the out-of-order window, a LSQ entry is allocated. A load can be issued only after its source operand dependence is satisfied and all earlier store instructions in program order has been issued or executed. This policy guarantees that a load can check if it is aliased with the any of previous stores by the time it reaches the writeback stage. A store instruction is decoded into two separate operations (e.g. an effective address generation and an 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 [42].

When a store-to-load aliasing is detected during execution, and the store value is available, the value is forwarded to the load instruction with the cache hit latency so that this operation does not incur any scheduling replay event. If an aliasing is detected but the store value is not available, the load and its dependent instructions are replayed. If a load cannot determine if it is aliased with earlier stores in program order since the store address operation was issued before the load but has been invalidated due to scheduling replay, this case is also handled as a scheduling miss; the load and its dependent instructions are

replayed. 39

3.2 Simulators

The experiments for this thesis research are performed using two types of simulators built based on *SimpleScalar-Alpha* 3.0a tool set [11], a suite of functional and timing simulation tools for the Alpha ISA. In order to characterize the groupability of instructions (Chapter 4), a functional simulator has been developed to collect program characteristics that are affected by neither machine parameters nor dynamic runtime environments. For detailed performance analysis of pipelined instruction scheduling and evaluation of macro-op scheduling and execution (Chapter 7 and Chapter 9), an execution-driven timing simulator has been developed, which models the details of macro-op scheduling and execution, as well as the key aspects of the microarchitecture described in Section 3.1.

3.2.1 Functional simulator

The functional simulator models the architectural behavior of the processor at instruction level. It is derived from *sim-profile* in the *SimpleScalar* tool set [11], and has been extended to collect program characteristics related to macro-op scheduling and execution. As the original tool set does, this functional simulator executes only user-level instructions and system-level instructions (e.g. system calls) and operating system codes are handled by separate proxy routines and hence it does not collect any information on operating system behavior.

3.2.2 Timing simulator

The execution-driven, timing simulator used in this thesis research is derived from

sim-outorder in the *SimpleScalar* tool set [11], and has been extended to model the detailed base machine microarchitecture and to incorporate the features required for macro-op scheduling and execution. Just as the functional simulator, this timing simulator only executes user-level instructions and does not capture system level behaviors. Here are several key aspects of modifications adopted to the original sim-outorder code:

- *Deeper and configurable pipeline*: in order to simulate the impacts of branch and load scheduling resolution loops on performance, modeling more realistic pipeline structures similar to current-generation processor implementations is essential. The timing simulator used in this thesis research can simulate extra pipeline stages added to the in-order portion (from fetch to queue) and the out-of-order portion (from schedule to execute) of the processor pipeline. In addition, the in-order portion of the pipeline correctly models the back pressure that may be created by e.g. queue contention, and prevents instructions from advancing to the next pipeline stages when queue contention is partially resolved (i.e. it does not compress instructions in the pipeline). All extra stages (in-order as well as out-of-order portion) added to the pipeline increase the branch misprediction penalty. In addition, the extra pipeline stages added between schedule and execute stages increase the load mis-scheduling penalty.
- *Speculative scheduling*: unlike the RUU-based machine model that the original sim-outorder assumes, the base processor microarchitecture is a physical-register-based machine in which an issued instruction acquires source values from a separate physical register file before execution. Due to the additional pipeline stages between schedule (*issue* in the original sim-outorder) and writeback, speculative

scheduling is essential in order to ensure consecutive execution of dependent instructions in such a pipeline configuration. The timing simulator used in this research faithfully models speculative scheduling and execution. Instructions are speculatively awakened and issued several clock cycles before the actual execution occurs, assuming load instructions have the fixed execution latency (DL1 hit latency). The speculative scheduling assumed in the base processor microarchitecture is detailed in Section 3.1.3.

- *Scheduling replay*: when a scheduling miss occurs due to load latency misprediction, the load and its dependent instructions are invalidated and replayed. The timing simulator used in this thesis research models three different types of issue-queue-based scheduling replay schemes: squashing replay, delayed selective replay, and position-based selective replay [57]. Among those replay schemes, the base machine model uses the position-based selective replay, which is an ideal scheme that instantly terminates incorrect speculative execution wavefront and does not affect independent instructions [57].
- *Separate issue queue from ROB*: unlike the RUU-based original sim-outorder, the timing simulator has issue queue structures managed separately from ROB, modeling the microarchitectural impact of issue queue contention on performance. The policy for issue queue management used in the timing model is similar to the one used in the Alpha 21264 [17]; an issue queue entry is allocated to each instruction when inserted into the out-of-order window, and deallocated when the instruction is successfully executed.
- *Improved memory disambiguation*: the timing simulator has an improved memory

disambiguation model that correctly handles fully overlapped (i.e. all data bits are available from a single earlier store that has not updated the memory state) and partially overlapped (i.e. some data bits are available from prior uncommitted stores but other bits should be accessed from the memory system) store-to-load aliasing cases. The instruction scheduler and the datapath were also appropriately modified so that it supports the memory disambiguation policy described in Section 3.1.4. Note that the base machine model assumes a full store-to-load bypass network that supports partially overlapped aliasing cases within the LSQ by accessing the data cache and merging cache and uncommitted store values. In order to validate the heuristics for detecting cycle conditions induced through memory dependences by grouping instructions in macro-op scheduling and execution, there is also an alternative configuration in which partially overlapped aliases are handled only through the memory system. Executing aliased loads is stalled until after antecedent stores are committed and write store values to the memory system in this case.

- *NOP handling*: Alpha binaries contain many no-ops (NOPs) to satisfy certain instruction alignment requirements. They are filtered out by the decoder without being executed [17]. We note that NOPs that perform cache prefetching operations are still executed but do not incur scheduling misses, as described in [17].

3.2.3 Verification and debugging

For the timing simulator that models detailed microarchitectural features such as speculative scheduling and replay as well as macro-op scheduling and execution, several difficulties arise in identifying the cause of malfunctions (i.e. deadlocks or livelocks) and

verifying execution behaviors for the following reasons. First, due to the attribute of speculative scheduling, a single dynamic instruction may incur multiple executions within the out-of-order window before being committed. Second, the readiness of instructions (output registers) can transition from ‘ready’ to ‘not ready’ state when scheduling replay occurs, which potentially causes other dependent instructions to complete their execution even before the parent instruction completes, if handled inappropriately. Third, identifying the cause of deadlock or livelock conditions should track multiple levels of dependence chains across instructions in reverse program order, since such conditions generally emerge many clock cycles after an incorrect simulator behavior occurs. Fourth, livelock conditions may occur due not only to simulator bugs but also to complicated and combined side-effects of scheduling priority functions, changes in register dependence mappings (for macro-op scheduling and execution) and memory access patterns of benchmark programs.

To verify the execution behavior and debug the timing model that ensures correct architectural state of the proposed microarchitecture, three standard approaches were used and implemented in the timing simulator as follows:

- *Back-end scoreboard*: to verify the correct scheduling and execution behavior of the timing simulator, a scoreboard is located in the back-end of the pipeline (i.e. writeback stage) and checks if each instruction is executed with correct source register inputs. This approach ensures that the out-of-order execution core runs dynamic instructions in the correct dependence order. Incorrect behavior observed during execution automatically terminates simulation, reporting the cause of the error.

- *Execution trace generation*: to track long-period execution behaviors and instruction movement in the processor pipeline, the timing simulator generates execution traces and log messages. This execution trace differs from the conventional dynamic instruction trace in that the execution trace contains unarchitected machine state, instruction movement and dynamic events in each pipeline stage. When deadlock or livelock conditions are detected by the watchdog timer that monitors if no instruction is committed for the predetermined period, the current simulation is terminated and the simulator is configured to generate the execution trace when the problematic simulation is performed again, so that long-period execution behaviors leading up to the emergence point can be tracked and analyzed.
- *Microbenchmark test*: to ensure the correct behavior of macro-op scheduling and execution implemented on the timing simulator, several microbenchmark programs were written and tested. The proposed techniques primarily improve instruction scheduling driven by register dependences, and are also dependent on the instructions placement. In these circumstances, a standard approach to generating microbenchmarks using, e.g. standard C codes compiled with a -O0 (no optimization) option is not suitable for our purpose since we lose full control over both instruction placement as well as register allocation. Therefore, all microbenchmarks were manually written using an inline assembler in C, and compiled by gcc using a -O2 option, which does not alter the manually-written assembly code but minimizes the overhead of other miscellaneous instructions added to the prologue and epilogue portion of the compiled binary.

Table 3-1: Base machine configuration.

Parameters	Configuration
Instruction fetch	4 instructions per cycle. Fetch stops at the first taken branch in a cycle. Cannot fetch across cache line boundaries in the same cycle. 32-entry fetch queue.
L1 Instruction Cache	16 Kbytes, 2-way set associative, 64-byte line, 2-cycle hit latency, LRU replacement policy. 4K-page, 64-entry ITLB.
Branch Predictor	Combined branch prediction of bimodal and gShare with a selector. The bimodal predictor has 4K-entry, 2-bit saturating counters. The gShare predictor has 4K-entry, 2-bit saturating counters with a 12-bit global history register. The selector has 4K-entry, 2-bit saturating counters. 16-entry return address stack (RAS). 4K-entry 4-way branch target buffer (BTB). The global history register is speculatively updated at prediction time and later fixed with correct history when a misprediction is detected. The whole branch predictor is permanently updated at commit time.
Out-of-order execution	4-wide issue and commit, 128-entry ROB, 128-entry unified issue queue, 128-entry LSQ, speculative scheduling, issue-queue-based and position-based selective replay.
Functional Units (latency)	4 integer ALUs (1), 2 floating ALUs (2), 2 integer MULT/DIV (3/20), 2 floating MULT/DIV (4/24), 2 general memory ports (2). A load takes 3 cycles (uncracked AGEN + port access).
L1 Data Cache	16 Kbytes, 4-way set associative, 64-byte line, 2-cycle hit latency, LRU replacement policy. 4K-page, 128-entry DTLB.
Memory System (latency)	256 Kbytes, 4-way, 128-byte line unified L2 (8), main memory (100).

The base machine configuration is summarized in Table 3-1. For the performance sensitivity study, I also test various machine configurations in later chapters.

Table 3-2: Benchmarks and their program characteristics.

Bench-marks	Input sets	Inst count	% 1-cycle ALUs	% controls	% loads	% stores	% long-lat / FPs	% NOP	% Misc
bzip	lgred.graphic	2.64B	48.4	11.0	24.5	12.3	0.0	3.7	0.0
crafty	crafty.in	3B	49.8	11.2	28.3	5.6	0.4	4.7	0.0
eon	chari.con-trol.cook	3B	25.8	11.4	24.6	17.3	14.2	6.7	0.1
gap	ref.in	3B	48.1	6.8	21.2	11.6	7.3	4.9	0.1
gcc	lgred.cp-decl.i	5.12B	36.6	14.3	20.1	11.5	0.1	16.9	0.5
gzip	lgred.graphic	1.79B	55.7	11.2	19.1	6.5	0.0	7.6	0.1
mcf	lgred.in	0.79B	37.3	21.7	23.8	8.7	0.0	8.1	0.4
parser	lgred.in	4.52B	45.5	15.9	21.8	9.1	0.1	7.6	0.1
perl	lgred.mark-erand	2.06B	40.1	13.4	24.8	10.2	1.8	9.1	0.0
twolf	lgred.in	0.97B	46.7	11.7	21.4	6.9	5.8	7.3	0.0
vortex	lgred.raw	1.15B	35.5	16.4	24.2	16.3	0.3	7.1	0.2
vpr	lgred.raw	1.57B	42.8	10.8	24.6	7.9	6.5	6.4	0.0

The benchmark programs that this thesis research uses are the SPEC CINT2000 benchmark suite [87]. Since one of primary goals of this thesis research is to relax the scheduling atomicity constraints, and macro-op scheduling and execution primarily focuses on optimizing integer instructions with single-cycle execution latencies (these reasons will be further discussed in Chapter 6 and Chapter 8), floating-point benchmarks were not tested. All benchmarks were compiled with the DEC C/C++ compilers under the OSF/1 V4.0 operating system using -O4 optimization. Table 3-2 shows the benchmarks, input sets, the number of instructions committed, and other program characteristics collected on the functional simulator.

The large reduced inputs sets from [60] were used for all benchmarks except for *crafty, eon* and *gap*. These three benchmarks were simulated with the reference input sets from beginning (without fast-forward) up to three billion instructions since the reduced

Table 3-3: Runtime characteristics.

Bench-marks	% IL1 misses / fetched insts	% DL1 misses / refs	% load replays / loads	% bpred misses / branches	Original IPC (w/ 0 extra stage)	Base IPC (w/ 1 extra stage)
bzip	0.01	1.73	5.01	5.40	1.55	1.53
crafty	1.87	4.21	7.46	5.63	1.57	1.55
eon	0.89	0.68	5.02	4.32	2.16	2.13
gap	0.51	0.66	2.09	5.17	2.11	2.10
gcc	1.15	2.41	4.64	6.32	1.31	1.29
gzip	0.01	3.02	7.90	5.37	2.04	1.99
mcf	0.01	12.61	46.74	3.64	0.38	0.38
parser	0.12	2.96	9.69	4.40	1.14	1.12
perl	0.13	0.19	3.69	18.16	1.37	1.31
twolf	0.26	4.65	19.45	10.62	1.54	1.50
vortex	1.71	0.17	6.89	0.90	1.76	1.75
vpr	0.01	3.10	13.08	12.65	1.70	1.64

inputs are not available. All other benchmarks were tested using end-to-end runs. Note that the number of committed instructions contains NOPs since they still affect the instruction fetch and occupy decode slots in the in-order portion of the pipeline, although they are filtered out by the decoder logic without execution.

Table 3-3 shows the runtime characteristics of the benchmarks collected on the base machine. The details of machine configurations were presented in Table 3-1. Note that instruction and data cache miss rates (second and third columns) are presented on a per instruction basis and a per reference (i.e. read or write access) basis, respectively. In addition, they include speculative accesses of wrong-path instructions. Compared with the DL1 miss rates (third column), on the other hand, the fourth column (*% load replays / loads*) in the table presents the rate of committed loads that incur scheduling replay due to load misses or store-to-load aliases. These numbers are greater than DL1 miss rates (third column) because many scheduling misses come from store-to-load aliases, and multiple accesses to a newly fetched cache line may all be counted as scheduling misses, whereas

only the first access to the line is counted as a cache miss. Therefore, this load replay rate is more meaningful since they account for the load misses observed by the processor core. The fifth column presents the rate of control (i.e. branches, direct / indirect jumps) mispredictions that incur either machine squashing (e.g. due to wrong target PC or direction) or fetch redirection (e.g. direction hit but BTB miss). In the case of machine squashing, the branch misprediction penalty is at least 14 clock cycles. The branch predictor configuration is detailed in Table 3-1.

The last two columns require an explanation. The *Original IPC* column shows the IPCs (committed instructions / execution cycles) measured on the baseline machine configuration where instruction scheduling logic performs a set of wakeup and select operation atomically every clock cycle. When scheduling logic is pipelined over two pipeline stages, the machine will suffer performance degradation primarily from not being able to issue dependent instructions consecutively. A secondary effect of pipelining instruction scheduling logic is that an extra stage (i.e. separate select stage) is added to the pipeline, which increases speculation-related penalties for load scheduling replay and branch misprediction recovery [6]. Since the secondary effect is dependent on the performance of branch prediction or load hit / miss prediction [17][101], it is necessary to decouple the secondary effect from experimental results so that we can measure how effectively the techniques proposed in this thesis improve pipelined instruction scheduling logic. To do so, the last column (*Base IPC*) presents the IPCs measured on the baseline machine with one extra pipeline stage. Therefore, the total number of pipeline stages in the base (with conventional instruction scheduling logic) and other comparison cases (with pipelined scheduling logic) are the same. For the rest of this thesis, all performance data will be

compared to this machine configuration, which will be referred to as the *base machine* unless specified otherwise. Note that this approach was also used in other studies on pipelined instruction scheduling logic [9][58]. Since the performance impact of the extra pipeline stage is 1.6% on average across the benchmarks, this should not affect the fundamental conclusions of this research.

In order to simplify our discussion and highlight the key results of the experiments performed for this thesis study, I frequently present relative performance data compared to the base case. IPCs of the base machine with various scheduler and issue width configurations used in this thesis can be found in Table 3-4 and Table 3-5. The detailed discussions on the results here will be presented in the corresponding chapters. In these tables, a machine with N issue bandwidth and M-cycle scheduling logic is referred to as *N-wide-M-cycle*. N-wide-1-cycle is a machine with conventional atomic instruction scheduling that issues dependent instructions consecutively. N-wide-2-cycle machines have scheduling logic pipelined over two stages. The details of 2-cycle scheduling will be discussed in Chapter 5. All machines have the same 4-wide fetch, decode and commit bandwidth as the base case (*4-wide-1-cycle*). Other narrow issue bandwidth configurations (*3-wide-M-cycle* and *2-wide-M-cycle*) will be used to evaluate macro-op execution in Chapter 9.

Before we start evaluating the effectiveness of macro-op scheduling and execution, the 4-wide-M-cycle machine with the 32-entry issue queue should be discussed, since its performance is not significantly better than that of 3-wide-M-cycle machines with the same size issue queue. This is because the machine is unbalanced in that the execution bandwidth is not fully utilized with such a small issue queue size. In other words, 4-wide execution bandwidth requires more than 32 issue queue entries (at least 48 entries

are preferable) given the pipeline and resource configuration. However, the results measured on this machine are still meaningful since our techniques reduce issue queue contention and overcome this limitation in many cases.

Table 3-4: Base IPCs with various scheduler and issue width configurations (32- and 48-entry issue queue).

Bench-marks	4-wide -1-cycle	4-wide -2-cycle	3-wide -1-cycle	3-wide -2-cycle	2-wide -1-cycle	2-wide -2-cycle
32-entry issue queue						
bzip	1.40	1.30	1.37	1.28	1.22	1.16
crafty	1.45	1.38	1.43	1.36	1.29	1.25
eon	1.86	1.81	1.83	1.78	1.63	1.59
gap	1.73	1.42	1.71	1.41	1.53	1.35
gcc	1.24	1.20	1.22	1.19	1.12	1.10
gzip	1.79	1.48	1.74	1.46	1.49	1.32
mcf	0.34	0.33	0.34	0.33	0.33	0.33
parser	1.06	0.94	1.05	0.93	0.98	0.88
perl	1.22	1.16	1.21	1.15	1.12	1.08
twolf	1.36	1.21	1.34	1.20	1.21	1.12
vortex	1.60	1.56	1.58	1.54	1.43	1.40
vpr	1.48	1.34	1.44	1.31	1.28	1.20
48-entry issue queue						
bzip	1.51	1.41	1.44	1.35	1.23	1.18
crafty	1.54	1.48	1.49	1.44	1.30	1.27
eon	2.13	2.05	2.05	1.97	1.67	1.63
gap	1.98	1.56	1.91	1.55	1.56	1.42
gcc	1.29	1.25	1.26	1.23	1.12	1.10
gzip	1.96	1.61	1.82	1.54	1.48	1.31
mcf	0.37	0.36	0.37	0.36	0.36	0.35
parser	1.12	0.99	1.09	0.97	0.98	0.89
perl	1.30	1.26	1.27	1.23	1.15	1.13
twolf	1.47	1.31	1.43	1.28	1.24	1.15
vortex	1.74	1.71	1.70	1.67	1.47	1.46
vpr	1.61	1.44	1.54	1.39	1.30	1.22

Table 3-5: Base IPCs with various scheduler and issue width configurations (64- and 128-entry issue queue).

Bench-marks	4-wide -1-cycle	4-wide -2-cycle	3-wide -1-cycle	3-wide -2-cycle	2-wide -1-cycle	2-wide -2-cycle
64-entry issue queue						
bzip	1.53	1.43	1.45	1.36	1.23	1.18
crafty	1.55	1.50	1.50	1.45	1.30	1.27
eon	2.13	2.07	2.05	1.99	1.67	1.63
gap	2.07	1.66	1.96	1.64	1.57	1.48
gcc	1.29	1.26	1.26	1.23	1.12	1.10
gzip	1.99	1.63	1.82	1.53	1.47	1.30
mcf	0.37	0.37	0.37	0.37	0.36	0.36
parser	1.12	0.99	1.09	0.97	0.98	0.89
perl	1.33	1.29	1.29	1.26	1.15	1.14
twolf	1.49	1.33	1.43	1.29	1.25	1.15
vortex	1.75	1.73	1.72	1.70	1.48	1.47
vpr	1.64	1.47	1.55	1.41	1.28	1.21
128-entry issue queue						
bzip	1.53	1.43	1.46	1.37	1.23	1.18
crafty	1.55	1.50	1.50	1.45	1.30	1.27
eon	2.13	2.07	2.05	1.99	1.68	1.64
gap	2.10	1.70	1.98	1.65	1.57	1.49
gcc	1.29	1.26	1.26	1.23	1.11	1.09
gzip	1.99	1.63	1.82	1.55	1.47	1.30
mcf	0.38	0.37	0.37	0.37	0.36	0.36
parser	1.12	0.99	1.09	0.97	0.97	0.89
perl	1.31	1.28	1.29	1.26	1.15	1.14
twolf	1.50	1.33	1.43	1.29	1.24	1.15
vortex	1.75	1.73	1.73	1.70	1.49	1.47
vpr	1.64	1.48	1.55	1.41	1.28	1.20

3.4 Summary

This chapter describes the experimental framework used in the thesis. The base microarchitecture is a current-generation superscalar, out-of-order processor with 4-wide fetch, decode, issue, dispatch, execute and commit bandwidth. The details of the pipeline structure and the operations of each stage are also presented.

Two types of the simulators are used in this thesis. The functional simulator models the architectural behavior of the processor, and is used to measure the program characteristics. The timing simulator models the detailed base machine microarchitecture and is extended to incorporate the features required for macro-op scheduling and execution. The key aspects of modifications adopted to the original simulator code are deeper pipeline, speculative scheduling, scheduling replay, issue queue contention, improved memory disambiguation and NOP handling. To ensure the correct behavior of the timing simulator that models the proposed microarchitecture, several verification and debugging approaches are used: back-end scoreboard, execution trace, and microbenchmark test.

The benchmark programs that this thesis research uses are the SPEC CINT2000 benchmark suite. All benchmarks (except for three benchmarks that use reference input sets) use the large reduced input sets and are tested using end-to-end runs. The program characteristics measured on the functional simulator as well as the runtime characteristics measured on the timing simulator are presented in this chapter.

Chapter 4

Groupability of Instructions

In Chapter 1, the basic concept of coarse-grained instruction processing was discussed and its potential benefits were described. Before discussing methods to realize the benefits of coarse-grained instruction processing, we want to measure the potential for constructing coarse-grained schedulable units from the original program binaries.

The *groupability of instructions* is the degree of abstraction that can be applied to a program by grouping instructions into coarse-grained schedulable units. To quantify this, this chapter begins by defining a *macro-op* and discussing its implications on program structure and microarchitecture. We then discuss issues in grouping instructions and measure the groupability of instructions in terms of the coverage of candidate instructions, the size of macro-ops and its impact on instruction-level parallelism.

There are several issues in determining which instructions are grouped and processed together. For performance, capturing many groupable candidates is needed to maximize its potential benefits. At the same time, improper grouping may degrade performance by serializing instruction execution. The complexity of creating macro-ops may be significantly affected by macro-op grouping policies such as macro-op size or grouping scope, i.e. the number of instructions that should be searched and examined to find groupable instructions. Based on the groupability of instructions measured in this chapter, macro-op grouping policies will be determined for the proposed microarchitecture.

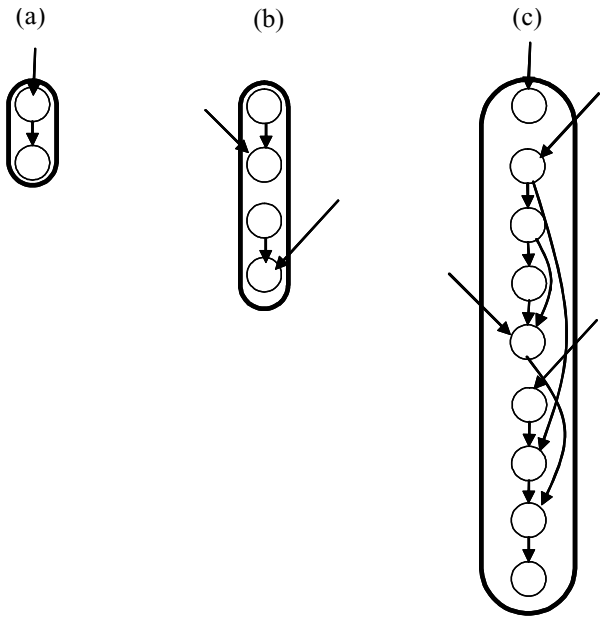


FIGURE 4-1. Examples of macro-op.

A macro-op (MOP) is defined as *an atomic schedulable unit that contains multiple instructions with a sequential execution order.* A MOP must be scheduled either completely or not at all. This means that 1) a MOP can be issued only after all source dependences are either satisfied or guaranteed to be satisfied so that every instruction in a MOP can complete without violating the data dependences defined in a program, and 2) issuing a MOP guarantees uninterrupted and deterministic execution of the instructions contained in it[1].

Figure 4-1 shows examples of MOPs that contain multiple instructions with data dependences represented as arrows. Figure 4-1a is a simple form of a MOP, which contains two dependent instructions. In a conventional instruction scheduler that makes scheduling decisions for each instruction, the dependent instruction may not be issued

1. The issues in precise exception handling or branch misprecition recovery will be discussed in Section 6.6.

consecutively following the parent instruction because of e.g. resource conflict. In contrast, the MOP confines scheduling to a predetermined order (i.e. sequential order) and forces consecutive issue of those two instructions. Figure 4-1b and Figure 4-1c show other forms of MOPs with different sizes; Figure 4-1b presents a MOP with a series of four instructions, which have two source dependences coming from the outside of the MOP; Figure 4-1c is a MOP of nine instructions with four source dependences. The first and the last instructions in a MOP are defined as *MOP head* and *MOP tail* instructions, respectively. Note that a MOP can contain either dependent or independent instructions. Regardless of the actual data dependences, instructions in a MOP are executed in sequential order, which was predetermined when the MOP was constructed. For example, the MOP head and the third instructions in Figure 4-1b do not have source dependences and could be executed sooner than any other instructions if they were not grouped. However, the four instructions in the MOP are sequentially executed (i.e. in the order of the MOP head, second, third, and MOP tail) when two source dependences to the second and the MOP tail instructions are satisfied, or guaranteed to be satisfied by the time each corresponding instruction is executed.

An important aspect is that MOPs only define an unarchitected state of machine behavior, i.e. instruction scheduling. The underlying assumption is that source and target registers of the original instructions have been renamed so that arbitrary instruction placement in a MOP does not reorder instructions nor affect data dependences. However, placing instructions in a MOP should ensure a legitimate scheduling behavior so that execution does not violate true data dependences, i.e., a parent instruction should be placed earlier than its dependent instructions.

Many x86 processor implementations crack a CISC instruction and convert it into multiple RISC semantics to run on RISC-style cores [21][48][42]. Despite their similarities, a fundamental difference between a MOP and a CISC instruction is the architectural visibility of intermediate results. This means that no instruction outside a CISC instruction can depend on an intermediate result generated during execution unless it is architecturally defined. However, all result values of the instructions in a MOP are architecturally visible since MOPs define scheduling behaviors but do not alter nor restrict the original dataflow. Another fundamental difference is that MOPs force uninterrupted execution of instruction sequences, while CISC instructions do not explicitly define the scheduling behaviors of multiple operations. For instance, if an x86 instruction is converted into two micro-ops, their execution timings are not restricted by the original CISC instruction semantic as long as data dependences between the two instructions are correctly preserved.

A MOP is different from a *strand* [53] in that executing instructions in a strand is interruptible whereas those grouped in a MOP should follow the predetermined execution timings. Also, MOPs can contain independent instructions, although we primarily focus on chains of dependent instructions.

4.2 Implications of Grouping Instructions into MOPs

Grouping multiple instructions into a MOP and processing them as a single schedulable unit has several implications. Figure 4-2 illustrates examples of the original data dependence graph as well as equivalent data dependence graphs composed of MOPs. To simplify our discussion, we assume that all instructions are single-cycle operations and

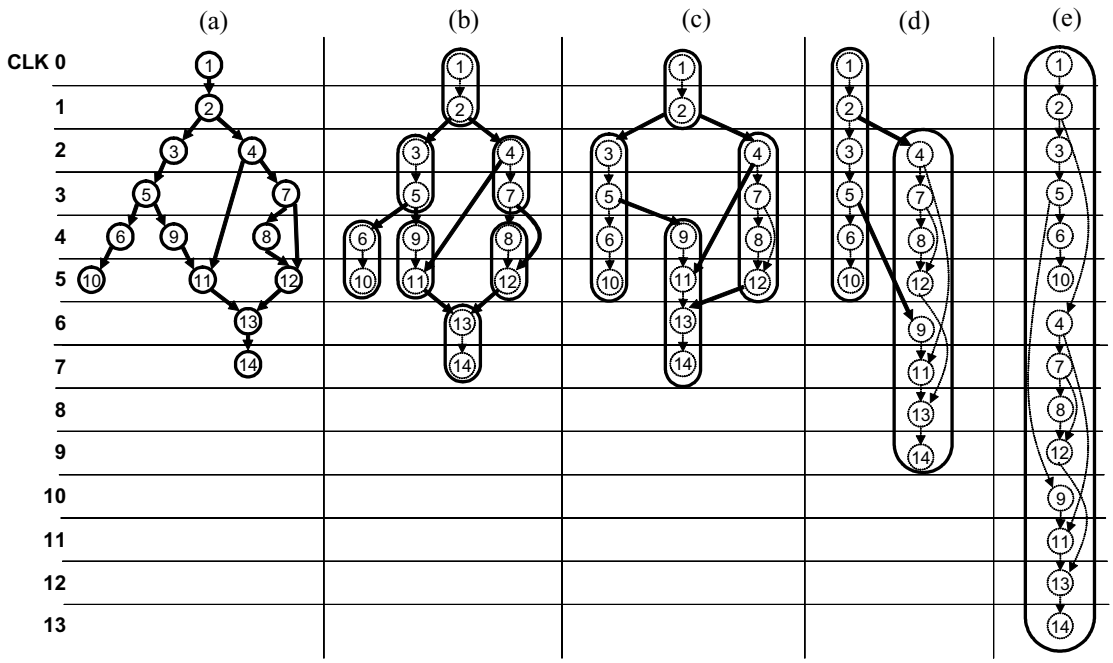


FIGURE 4-2. Grouping instructions into MOPs with different sizes.

only register dependences exist.

One implication of grouping instructions into MOPs is *dependence abstraction*. Figure 4-2a shows the original dependence graph of 14 instructions. In a conventional out-of-order processor, instruction scheduling is performed on individual instructions based on this data dependence graph. Figure 4-2b shows a case when MOPs contain a pair of two dependent instructions each. Since each MOP defines the execution order of the two instructions grouped in it and the dependence edge between the two need not be explicitly observed by others as far as scheduling is concerned, the original data dependence graph can be abstracted into one with fewer schedulable units (seven MOPs) which preserve the true dependences among instructions. Figure 4-2c~e show the examples of the same dependence graph abstracted into larger MOPs that contain more than two instructions. As the dependence graph is further abstracted, the number of schedulable units and dependence edges involved in scheduling decreases because more instructions and dependence

edges are included within MOPs. An extreme case is represented in Figure 4-2e; all instructions are grouped into a single MOP. Since the MOP is constructed in a way that the sequential order preserves the true dependences among instructions, this MOP still generates a legitimate execution although no parallelism can be achieved.

Another implication of grouping instructions into MOPs is *increased scheduling determinism*. Since MOPs have predetermined scheduling decisions for their instructions, the execution timing of each instruction can be inferred several clock cycles before the actual execution occurs. In Figure 4-2a, for example, instruction 1 is issued at CLK 0. Although instruction 2 becomes ready in the next clock cycle, the instruction scheduler does not guarantee it to be issued at CLK 1, which makes other dependent instructions (instructions 3 and 4) unable to determine their scheduling decisions until after their parent instruction 2 is actually issued. In contrast, MOPs enable other dependent instructions to infer when their source operands become ready several clock cycles before the actual issue occurs. In Figure 4-2c, issuing the MOP with instruction 1 guarantees instruction 2 to be issued at CLK 1. This in turn enables instructions 3 and 4 to infer when they become ready to issue, i.e. at CLK 2. Similarly in Figure 4-2e, issuing instruction 1 at CLK 0 determines that other following instructions 2 to 14 will be issued at corresponding clock cycles from 1 to 13, providing a lookahead of up to 13 clock cycles to instructions potentially dependent on instruction 14.

An important observation to make in grouping instructions is that out-of-order instruction scheduling does not have to be driven by data dependences defined in a program, as long as the dependences within and across MOPs guarantee legitimate execution. The purpose of out-of-order instruction scheduling is to generate an order of execution

that expedites the value communication defined in a program. Unlike conventional instruction scheduling that requires fine-grained, instruction-level controls over instructions, grouping instructions allows scheduling and execution to be performed in two different dependence domains: coarser MOP-level dependences for scheduling, and finer instruction-level dependences for execution. This enables simpler dependence tracking in scheduling logic without affecting the actual dataflow, although dependence translations between two different domains may be required. The basics of constructing and tracking MOP-level dependences will be discussed in the following sections, while a detailed implementation is proposed in Section 6.4.

In summary, grouping instructions into MOPs has the following implications:

- *Dependence abstraction*: grouping instructions reduces the number of schedulable units. This in turn enables to reduce the queue pressure involved in instruction scheduling.
- *Increased scheduling determinism*: grouping instructions restricts scheduling decisions to a subset of all possible scheduling outcomes in a foreseeable manner. Hardware resources to schedule and execute instructions can look ahead of future events and therefore have more time to determine the next actions to take.

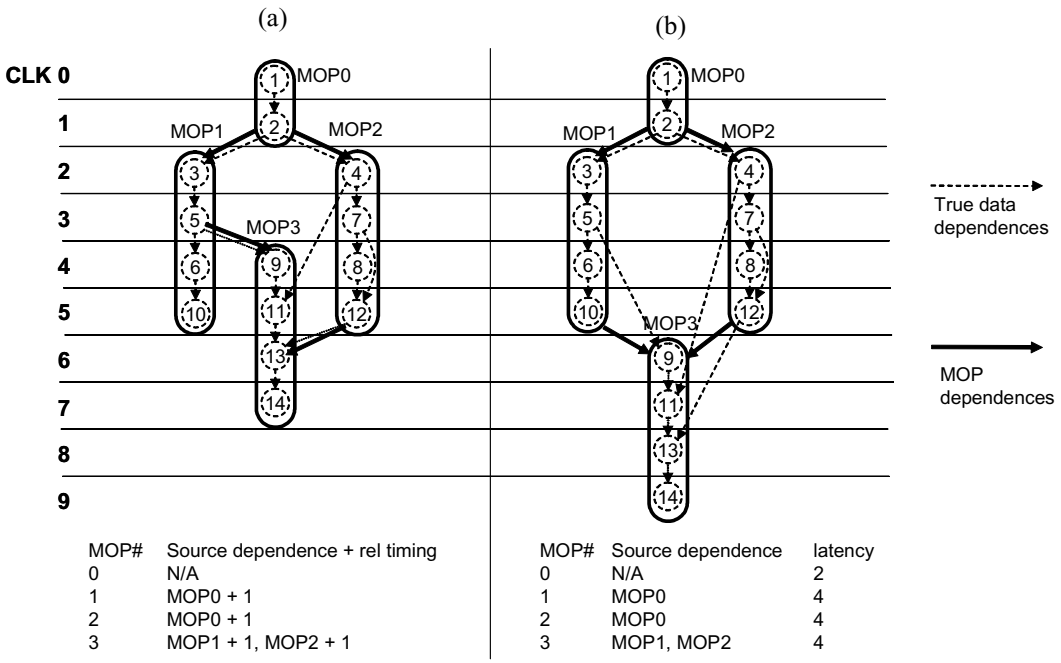
4.3 MOP Dependences

Scheduling of MOPs must preserve the original data dependences among instructions so that it does not violate the dataflow defined in a program. The original data dependences can be categorized into two groups as far as MOP formation is concerned: dependences within a MOP or *intra-MOP dependences*, and *inter-MOP dependences* that

span across MOP boundaries. Intra-MOP dependences are statically preserved at MOP formation time by placing instructions in a certain execution order. Although there must be several policies to construct an execution order to satisfy the original data dependences, statically placing instructions in program order is sufficient since no older instruction in program order depends on newer instructions. In contrast, inter-MOP dependences need to be dynamically tracked across MOP boundaries in order to synchronize executions of instructions grouped in different MOPs in a way to satisfy the original data dependences. A MOP can be issued only after all source dependences are either satisfied or guaranteed to be satisfied, as discussed in Section 4.1. Instead of tracking readiness of inter-MOP dependences of each instruction individually, they are combined and handled as source operands of a MOP. These combined dependences of MOPs will be referred to as *MOP dependences*. Only MOP dependences are visible to the scheduler and dynamically drive scheduling operations; intra-MOP dependences do not participate in dynamic instruction scheduling.

The basic concept of a MOP does not confine detailed scheduler implementations, e.g. how to implement the instructions scheduler nor how MOP dependences are tracked. To help understand the process of tracking MOP dependences, Figure 4-3 replicates the data dependence graph in Figure 4-2c and presents two different approaches to constructing MOP dependences and checking their readiness.

Figure 4-3a illustrates an example of the *MOP offset tracking* approach, in which execution of multiple dependent MOPs can be overlapped by tracking instruction offsets within MOPs. When MOPs are initially constructed, each instruction reads the offsets of parent instructions within their own MOPs, and then calculates timing difference between



MOP#	Source dependence + rel timing
0	N/A
1	MOP0 + 1
2	MOP0 + 1
3	MOP1 + 1, MOP2 + 1

MOP#	Source dependence	latency
0	N/A	2
1	MOP0	4
2	MOP0	4
3	MOP1, MOP2	4

FIGURE 4-3. Offset tracking (a) vs. latency tracking (b) of MOP dependences.

parent and dependent MOPs. For example, *MOP3* has three inter-MOP source dependences from *MOP1* and *MOP2*; instructions 9, 11 and 13 depend on instructions 5, 4 and 12, respectively. Instruction 9 is placed at offset 0 in *MOP3*. Instruction 5 is placed at offset 1 in *MOP1*. The relative offset between the two instructions is 1 (instruction 5) - 0 (instruction 9), which becomes one clock cycle. This means issuing *MOP3* should be delayed by at least one clock cycle after *MOP3* observes the tag broadcast by *MOP1* (or *MOP3* should be issued at least two clock cycles after *MOP1*, assuming a 1-cycle delay for broadcast). Relative timing difference can be a negative value, e.g. -1 between instructions 4 and 11, which implies that the dependent instruction can be unnecessary delayed even after the source dependence has already been satisfied. Although multiple dependence edges exist between *MOP2* and *MOP3*, *MOP3* tracks only the maximum relative offset (the dependence between instructions 12 and 13) since this is a sufficient condition to satisfy other dependences (between instructions 4 and 11). MOP dependences of each

MOP are summarized at the bottom of Figure 4-3a.

From the perspective of scheduler implementation, tracking MOP dependences is fundamentally no different than the conventional case, except that each issue queue entry has delay timers (initially set to the relative offset value) for their source operands to check readiness of parent MOPs, as it already must for multi-cycle instructions in conventional scheduling logic. However, a single parent MOP may be interpreted to have different execution latencies by different MOPs dependent on it, and this approach requires each instruction to calculate relative offsets when MOPs are constructed.

Figure 4-3b shows the *MOP latency tracking* approach, in which each instruction in a MOP does not calculate relative offset differences but only checks which MOP is antecedent. A MOP is simply handled as a multi-cycle operation. For example, instruction 9 in *MOP3* is dependent on instruction 5 in *MOP1*. Instead of identifying the offsets of instructions 5 and 9, *MOP1* and *MOP3* simply assume the worst-case latency, as if the scheduler handles two dependent multi-cycle instructions. All instructions in the parent MOP (*MOP1*) are issued before any instruction in the dependent MOP (*MOP3*). Although dependence tracking in the scheduling logic is simpler than offset tracking, it does not allow two dependent MOPs to be partially overlapped and may lose performance.

In later chapters on MOP scheduling and execution, appropriate tracking approaches will be used depending on the configurations of scheduling logic and MOPs.

4.4 Issues in Grouping Instructions

MOP dependences that drive scheduling of MOPs (shown as solid arrows in Figure 4-3) are not always consistent with the original data dependences (shown as dashed

arrows) in both approaches. This is because MOP dependences provide *sufficient* conditions to satisfy multiple original dependences of grouped instructions simultaneously, while conventional instruction scheduling is driven by individual data dependences that provide *necessary* conditions for correct execution. The degree of discrepancy between MOP dependences and the original dependences can be interpreted as the amount of sufficiency injected into the data dependence graph to enforce correct execution.

These sufficient conditions of MOP dependences potentially flatten the data dependence graph and may not be beneficial for performance. As instructions are grouped and the dependence graph is abstracted, the scheduler loses fine-grained, instruction-level controls over instructions and hence the parallelism extractable among instructions may decrease; MOPs may force instructions to execute sequentially when they might have executed in parallel in the base case. Examples of performance loss due to grouping instructions are shown in Figure 4-2d and Figure 4-2e, where independent instructions grouped in MOPs are unnecessarily delayed. To minimize the negative performance impact of grouping instructions, MOPs must be constructed in a way that MOP dependences resemble the true data dependences as much as possible.

There are some other issues in determining which instructions are grouped and processed together. The complexity of creating and handling MOPs is dependent on grouping policies. Also, correctness of scheduling should not be compromised by improper MOP grouping. The following sections discuss these issues in grouping instructions into MOPs.

4.4.1 Dependences among grouped instructions

A chain of dependent instructions is more suitable for a MOP than independent

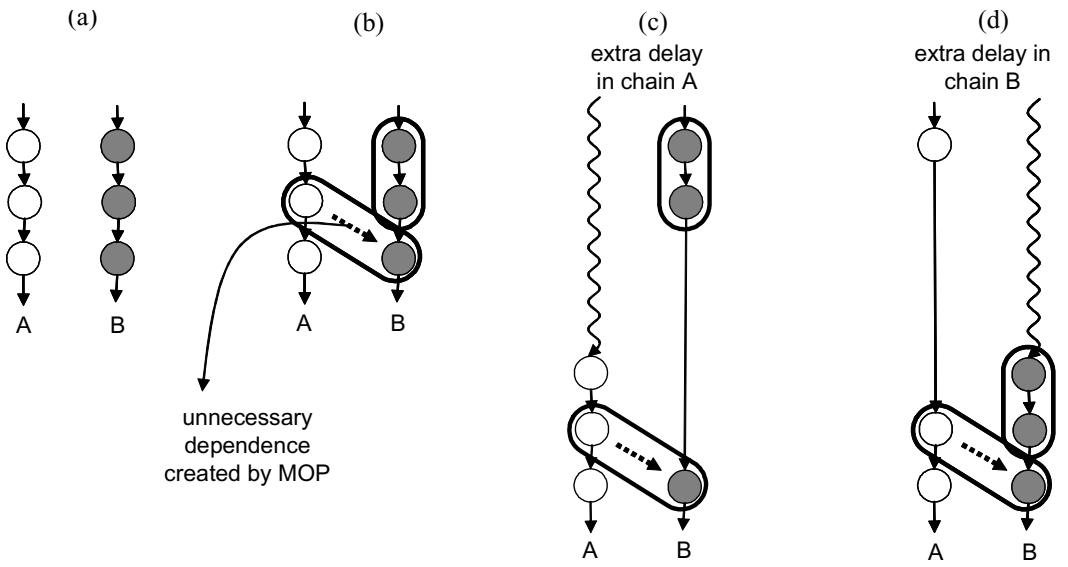


FIGURE 4-4. Impact of grouping independent instructions.

instructions because, by definition, a MOP forces consecutive and serial execution of its grouped instructions. Grouping independent instructions creates unnecessary dependences that not only serialize two independent instructions, but also causes independent chains to be tied up together. Consider an example in Figure 4-4a with two separate chains of dependent instructions. A MOP can be created across two chains, grouping independent instructions as shown in Figure 4-4b. If the scheduler tracks instruction offsets within MOPs as discussed in Section 4.3, grouping two independent instructions may not necessarily degrade performance and instructions can be issued and executed with the same timings as the base case. If an extra delay is incurred in one chain, they will affect the other chain through the unnecessary dependence edge created by the MOP, delaying instructions in both chains (Figure 4-4c and Figure 4-4d). Balancing the critical paths that lead up to a MOP with independent instructions may eliminate this problem but requires global analysis of register dependences as well as the perfect knowledge of future dynamic events that skew the dependence tree structure. Therefore, serial portions of the data dependence tree are the primary candidates for MOP grouping.

4.4.2 MOP size

MOP size is the number of instructions grouped in a MOP. It determines the granularity of processing in the scheduler as well as the degree of abstraction applied to the original data dependences. The more instructions are grouped, the fewer scheduling units participate in dynamic instruction scheduling, reducing the number of dependences for the scheduler to track. At the same time, bigger MOPs increase the likelihood of incurring unnecessary false dependences since MOP dependences should be conservative enough to satisfy all of the original dependences of the instructions grouped. This problem gets worse in MOP latency tracking (explained in Section 4.3) where the scheduler is only able to track MOP execution latencies, because scheduling assumes the worst case delay, which increases with MOP size.

4.4.3 MOP scope

MOP scope is the number of instructions to be searched and examined to find groupable instructions. A wider MOP scope is essential for capturing more grouping opportunities because this increases the probability of observing chains of dependent instructions within the range. However, there are several difficulties in widening MOP scope over a certain level. First, a wider MOP scope requires more instructions to be buffered and examined, which increases hardware complexity involved in detecting groupable candidates. Second, from the perspective of implementation, instructions are processed sequentially in the front end of the pipeline, where MOP grouping and its miscellaneous book-keeping duties take place. If the original instructions to be grouped into a MOP are placed far away from each other in program order, benefiting from the MOP is difficult because the MOP tail may not have been fetched when the MOP head is ready to execute.

Therefore, MOP scope should be determined by considering the coverage of groupable candidate instructions, the complexity of hardware implementation, and its potential microarchitectural impacts.

4.4.4 Merging and branching of dependence chains

Merging and branching of data dependence chains affects performance as well as the way MOPs are constructed. Merging of dependence chains occurs in instructions with multiple source operands. Branching of dependence chains occurs in instructions with multiple consumers (dependent instructions). Since a MOP contains a single chain of dependent instructions, whether MOP formation starts a new MOP or terminates an existing one should be determined at those merging and branching points.

The most restrictive and conservative way to avoid any performance penalty from grouping instructions is 1) starting a new MOP at merging points, and 2) terminating an existing MOP at branching points (for the latency tracking case discussed in Section 4.3). This policy guarantees MOP dependences to exactly match the data dependence edges without injecting any sufficient conditions nor extra delays, regardless of actual issue and execution timings. However, it may lose many grouping opportunities even when grouping across these points does not negatively affect performance.

When instructions are grouped across merging and branching points, the following factors should be considered to avoid performance penalty.

4.4.4.1 Dependence merging point

Figure 4-5 illustrates several scenarios for grouping instructions across a dependence merging point located at instruction 3. Without grouping, the execution timing of instruction 3 is determined by the last arriving operands originating from instructions 1 or

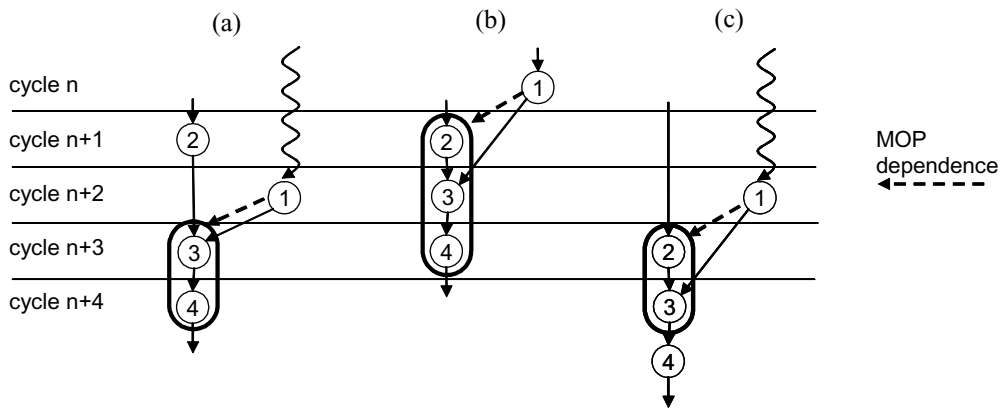


FIGURE 4-5. Impact of grouping a dependence merging point.

2. As mentioned, a MOP of instructions 3 and 4 (Figure 4-5a) does not delay execution regardless of the actual last arriving operand since MOP dependences are consistent with the true data dependences.

Consider a case shown in Figure 4-5b where instructions 2 and 4 are grouped across a dependence merging point at instruction 3. As long as the dependence edge from instruction 1 is not the last arriving operand of the MOP, the execution timings of individual instructions are the same as the base case (without grouping). This is also true for the case in Figure 4-5c. However, if the edge from instruction 1 is the last arriving operand of instruction 3, its grouped and antecedent instructions (instruction 2 in the figure) are negatively affected since they cannot be executed until after all source operands of the MOP are satisfied. Note that execution of instructions 1 and 2 cannot be overlapped in Figure 4-5b and Figure 4-5c since the MOP in each figure should observe the tag broadcast by instruction 1 before determining its issue timing.

Avoiding performance penalty requires knowledge of last arriving operands. Since it is not always possible to statically detect last arriving operands, creating MOPs may rely on a history-based prediction mechanism similar to the one used in [56].

4.4.4.2 Dependence branching point

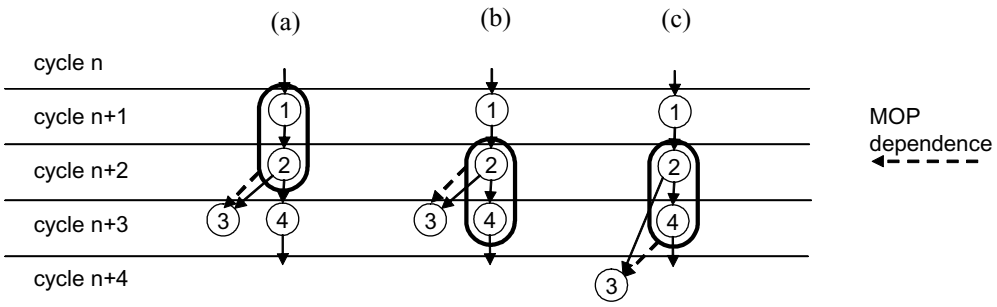


FIGURE 4-6. Impact of grouping a dependence branching point.

Figure 4-6 illustrates several scenarios for including a dependence branching point (instruction 2) in a MOP. A dependence branching point placed at the end of a MOP as a tail instruction (Figure 4-6a) does not affect the execution timings of dependent instructions since MOP and data dependences are consistent. In the case of MOP offset tracking (Figure 4-6b), executing instructions dependent on non-tail instructions (instruction 3 in the figure) are allowed to overlap with other grouped instructions and hence do not incur extra delays. However, they may be unnecessarily delayed in the MOP latency tracking approach (Figure 4-6c) because of the MOP dependence between the MOP tail (instruction 4) and the dependent instructions (instruction 3) in order to guarantee that all instructions in a MOP are executed ahead of any dependent instructions outside the MOP.

4.4.5 Cycle conditions

Grouping instructions into MOPs may prevent instructions from being executed if it induces cycles in data dependence chains. Cycle conditions may be induced through either register dependences or memory dependences when an outgoing dependence edge from a MOP comes back to itself as a source dependence through other instructions outside the MOP. The details of cycle conditions through register and memory dependences, their detection and avoidance techniques will be further discussed later in Section 6.3.1 and Section 6.3.2.

4.5 Quantifying the Groupability of Instructions

To quantify the groupability of instructions, this section identifies candidate instruction types and determines policies for grouping. The following sections measure the number of instructions grouped in MOPs and MOP sizes considering performance impact, and study sensitivity to various macro-op configurations.

4.5.1 Candidate instruction types

Since one of the primary goals of this thesis research is to relax the atomicity of instruction scheduling, I focus on grouping single-cycle operations: single-cycle ALU, control (e.g. branch), and load / store address operations. Load instructions are not explicitly cracked into two separate operations and have multi-cycle latencies (i.e. 1-cycle address generation and 2-cycle cache port access) in our base machine model described in Section 3.1. Since they are not primary candidates for these reasons, we will separately measure groupability with and without them.

Grouping instructions with multi-cycle execution latencies (e.g. integer multiply or floating-point operations) is possible but was not considered here due to the limited benefits of doing so. Multi-cycle latency instructions do not require single-cycle atomic instruction scheduling. Further, such instructions (particularly floating-point operations) are usually scheduled and executed in separate pipelines with dedicated functional units and datapaths. Finally, handling instructions with different scheduling and resource requirements is not feasible due to restrictions in the datapath designed for macro-op execution (this will be discussed in Chapter 8). Since non-candidate instructions account for relatively a small number of instructions in the majority of integer programs, this policy for determining candidate instruction types should not affect the fundamental conclusion

Table 4-1: MOP candidate instruction types.

Instruction types	MOP candidate	MOP head only	MOP tail only
1-cycle integer ALU	All types	SEXTBI, SEX-TWI	
1-cycle control operations	All types	BR, BSR	BLBC, BEQ, BLT, BLE, BLBS, BNE, BGE, BGT
stores	Store AGEN		Store AGEN
loads	Load AGEN		Load AGEN
Others (complex integer, floating-point, NOP, syscall, and etc.)	none		

of this thesis.

Among the MOP candidate instructions, not all instructions can be MOP head or MOP tail instructions since some instruction types do not have source or target registers. Integer ALU candidates can be either MOP heads or tails. On the other hand, most branch instructions can be only MOP tails since they do not have dependent instructions. In the Alpha ISA [17] studied in this thesis, a few indirect jump instructions can have dependent instructions to save return addresses in registers and therefore can be MOP heads. Store address operations can be only MOP tails because they have no dependent instructions except for the actual memory access operations, which are separately managed by the load / store queue in our base microarchitecture. For a similar reason, loads are grouped only as MOP tails. Table 4-1 summarizes MOP candidate instruction types used in this study.

4.5.2 MOP scope and size

To limit the range of study in determining MOP scope, i.e. the number of instructions to be searched and examined in program order for groupable candidates, we characterize the dependence edge distance measured in terms of instruction count between two candidate instructions in Figure 4-7 (without candidate loads) and Figure 4-8 (with candi-

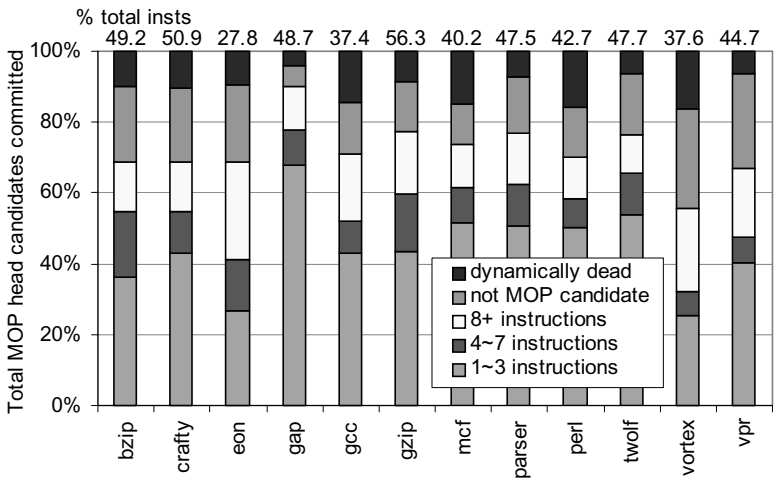


FIGURE 4-7. Dependence edge distance from a MOP candidate to the nearest groupable instruction (measured without load MOP tail).

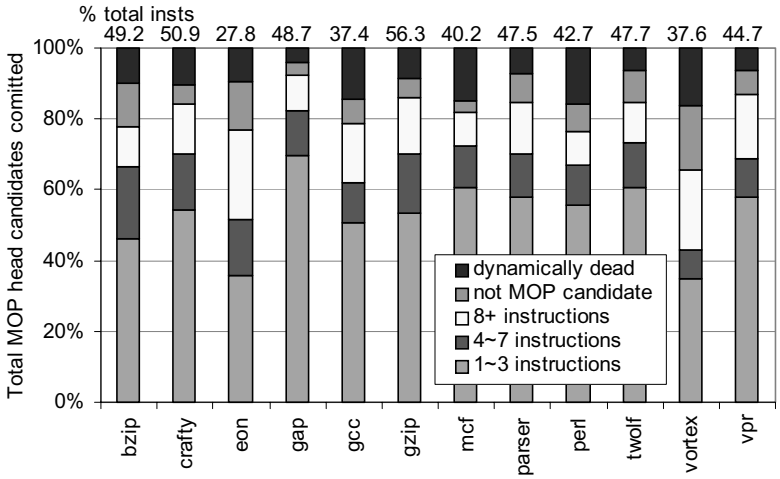


FIGURE 4-8. Dependence edge distance from a MOP candidate to the nearest groupable instruction (measured with load MOP tail).

date loads as MOP tails). In the graph, the y-axis represents all potential MOP head candidates. Their 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. 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). Note that the data shown here show program characteristics, and are not dependent on machine configuration.

The data indicate that many groupable dependent instructions are placed near their parent instructions. Especially, an 8-instruction scope (shown as *1~3* and *4~7 instructions*) captures a significant portion of groupable instructions. Compared with Figure 4-7 (without load), including loads as MOP tail candidates strengthens this trend because of many ALU instructions used for stack pointer manipulation. Some benchmarks like *vortex* and *eon* have many dependence edges that span over eight instructions and require a wider scope to capture groupable instructions. In measuring the groupability of instructions in the following sections, we limit the MOP scope up to 32 instructions because the number of grouped instructions saturates at this MOP scope in many cases.

We examine three different configurations for MOP sizes: 2x, 4x and 8x configurations. The 2x configuration allows only two instructions to be grouped. The 4x and 8x configurations can group up to four and eight instructions, respectively. Hence, they may group fewer instructions than their limits if insufficient groupable candidates are found given a MOP scope. Although a MOP can capture as many instructions as the MOP scope (up to 32 instructions), MOP sizes bigger than eight instructions were not examined because many dependence chains tend to be much shorter than eight instructions [53]. Note that the MOP scope limits the total length of a dependence chain grouped in a MOP, i.e. the distance between MOP head and tail instructions should not exceed the MOP scope.

4.5.3 MOP grouping policies

An important issue in measuring the groupability of instructions is that there is no optimum solution to generate maximum benefit, since it is a function of not only the number of grouped instructions but also the impact on performance, which is dependent on

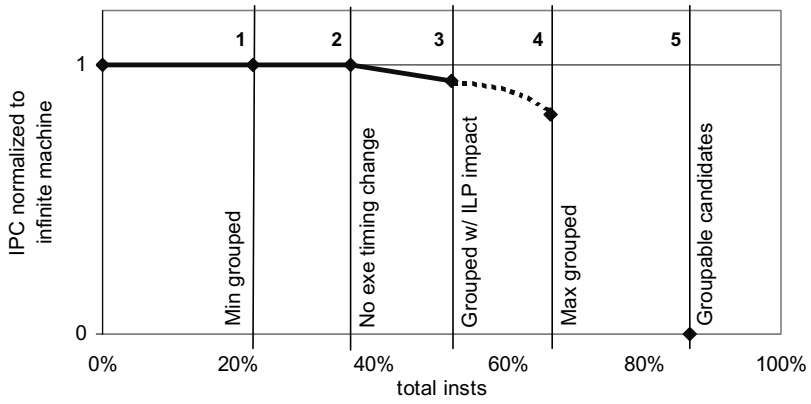


FIGURE 4-9. An example of groupability-performance curve.

machine parameters with different hardware constraints. For example, a solution to maximize the number of grouped instructions may not be beneficial due to the dependence serialization discussed in Section 4.4. On the other hand, a solution that avoids any potential serialization may not be optimum either because the serialized instructions could be located off the critical path (from the perspective of both data dependence and hardware constraint [31]) and hence may not affect performance.

To reflect this issue, our measurement uses four different grouping policies so that both groupability and performance can be considered. Figure 4-9 illustrates an example of a groupability-performance curve. In the graph, the y-axis represents relative performance normalized to an *infinite* machine (normalized performance of 1 in the figure). This infinite machine assumes infinite fetch and out-of-order execution bandwidth with perfect cache and branch prediction. Hence, performance is bounded only by true data dependences through registers and memory locations. In fact, this is similar to measuring the instruction-level parallelism in a program but different in that our infinite machine assumes different types of instruction latencies, which are set to those used in our finite base machine. An overview of the infinite machine is presented in Table 4-2.

Table 4-2: Infinite machine configuration.

Parameters	Configuration
Instruction fetch	Infinite, perfect knowledge of control flow
Memory System	2-cycle perfect cache, perfect memory disambiguation, infinite store buffer. 4-byte granularity for memory dependence detection (simulator limitation)
Execution	Infinite instruction window, infinite execution bandwidth, 0 pipeline stage
Instruction latencies	Same as those of the base machine configuration in Table 3-1

The x-axis in Figure 4-9 represents the number of instructions grouped. 0% implies no grouping and hence no performance degradation due to serialization conditions. The first data point denoted as *min grouped* represents a policy in which instructions with a dependence merging point (e.g. 2-source instructions or loads with memory dependence) can be grouped only as MOP heads. Stores are handled as 1-source AGEN operations and hence do not create dependence merging points within a MOP. This policy avoids any potential performance degradation due to last-arriving operands (discussed in Section 4.4.4.1). To decouple implementation issues in dependence tracking from characterizing the groupability, the MOP offset tracking approach is assumed so no performance degradation is incurred by dependence branching points (discussed in Section 4.4.4.2).

The second data point denoted as *No EXE timing change* is a grouping policy that allows dependence merging points within a MOP as long as they do not alter the execution timings of individual instructions observed in the no-grouping case. Note that the solution acquired from this grouping policy does not guarantee no performance degradation in a finite machine due to different execution timings. This policy captures more instructions than the *min grouped* case, but is still conservative because the execution timing changes may not necessarily affect performance if they are not in the critical path of the data dependence graph.

To consider this limitation, the third data point denoted as *Grouped with ILP impact* is plotted, where changes in execution timing are allowed as long as the delay incurred by dependence merging points does not exceed a certain threshold (20 clock cycles in this experiment).

The forth data point denoted as *Max grouped* represents a greedy grouping policy that captures MOPs without considering performance degradation. The algorithm used in this policy is that 1) an instruction with one source operand is grouped with its parent instruction as long as the MOP size does not exceed its maximum limit, and 2) an instruction with two source operands first examines the nearest parent instruction in program order then tries the other parent instruction if the first try was not successful. Note that this policy still does not guarantee a solution that maximizes the number of grouped instructions. However, the results measured with other algorithms (e.g. testing farthest instructions first) are not fundamentally different from the current data. More importantly, achieving an optimal solution requires global knowledge of the whole data dependence graph, which is not feasible in a real hardware implementation. Therefore, this data point should be interpreted as a maximal groupability attainable using an algorithm feasibly implementable in hardware.

Finally, the fifth data point simply represents the total number of MOP candidates in a program. The candidate instruction types are presented in Table 4-1. The first set of results (Figure 4-10 and Figure 4-11) does not include loads as MOP candidates. The second set of results (Figure 4-12 and Figure 4-13) includes loads, and hence show a significant increase in this data point compared with the first set of results.

From 0% to data point 2, no performance degradation should be observed (normal-

Table 4-3: Performance on the infinite machine compared to the base machine.

Benchmarks	Inst count	Execution cycles	IPC on infinite machine	IPC on finite base machine (identical to Table 3-3)
bzip	2.64B	63.96M	41.34	1.55
crafty	3B	39.44M	76.06	1.57
eon	3B	83.50M	35.93	2.16
gap	3B	34.05M	88.09	2.11
gcc	5.12B	75.65M	67.64	1.31
gzip	1.79B	74.49M	23.98	2.04
mcf	0.79B	3.04M	261.45	0.38
parser	4.52B	113.03M	40.05	1.14
perl	2.06B	376.91M	5.33	1.37
twolf	0.97B	34.16M	28.47	1.54
vortex	1.15B	43.79M	26.34	1.76
vpr	1.57B	47.20M	33.20	1.70

ized IPC of 1). Depending on benchmarks, data points 2 and 3 may not be distinguishable since they are plotted too closely. A dashed curve between data points 3 and 4 is interpolated based on points 2, 3 and 4.

4.5.4 Caveat

Table 4-3 presents IPCs on the infinite machine and compares them to the actual performance measured on the finite base machine. Note that there is no direct correlation between the two IPCs because no structural limitation is considered in the infinite machine. Also note that the performance results in this section do not directly indicate the degree of actual performance impact on finite machines, because many hardware constraints hide negative effects of MOP grouping. Rather, they should be interpreted as impacts on the instruction-level parallelism of a program, which may or may not be related to the actual performance on finite machines.

4.6 Results

4.6.1 Coverage of candidate instructions

Figure 4-10 and Figure 4-11 present the percentage of total instructions grouped in MOPs in each benchmark when loads are not categorized as groupable candidate instructions. For MOP size and scope, a MOP captures up to eight instructions within a 32-instruction scope for this measurement. Across the benchmarks, 53 ~ 73% of total instructions are MOP candidates. When instructions are maximally grouped without considering performance impact (*Max grouped*), the percentage of grouped instructions ranges from 27.0% (*eon*) to 56.2% (*gzip*) of total instructions (42.6% on average). The variability in the number grouped instructions is related to the dependence edge distance presented in Figure 4-7. For example, benchmarks like *eon* and *vortex* have relatively long dependence edges and therefore show low potential since fewer candidate instructions are found in the same MOP scope than other benchmarks.

Considering the performance impact of MOP groupings, we find that maximally grouping instructions severely reduces the parallelism in many cases by serializing unnecessary instructions. Especially in *eon* and *gap*, this inconsiderate and greedy grouping policy quadruples the execution time on the infinite machine. This impact will not affect the actual performance in the same degree as shown here because the actual execution is bounded by not only data dependences but also many other hardware constraints (e.g. fetch serialization) or dynamic events (e.g. cache misses). However, the results indicate that avoiding harmful MOPs would be crucial in many cases. In fact, sacrificing a few grouping opportunities (~10% of maximally grouped instructions) can eliminate the negative impact of last-arriving operands at dependence merging points, as the data points at

An interesting result is observed at *min grouped* data points: allowing only 1-source instructions still captures many groupable candidates in most benchmarks. This is because 1-source instructions account for a significant portion of instructions in the benchmarks we tested [56]. However, this policy tends to capture only short dependence chains (two instructions in most cases) because of many intervening 2-source instructions in long dependence chains. The results on MOP sizes will be presented in the following chapter.

Figure 4-12 and Figure 4-13 show the percentage of grouped instructions including loads when the same experiment is performed. This configuration groups noticeably more instructions in many cases by capturing ALU-to-load pairs frequently. When maximally grouped, the percentage of grouped instructions ranges from 32.3% (*eon*) to 64.9% (*gzip*) of total instructions (50.9% on average). Since the groupability-performance curve show similar trends with or without groupable load instructions, we will not further discuss the detailed results.

In summary, the results in this chapter indicate that 1) there are a significant number of instructions that can be grouped into MOPs, and 2) a judicious grouping policy is required to avoid performance penalty from unnecessarily serializing instructions.

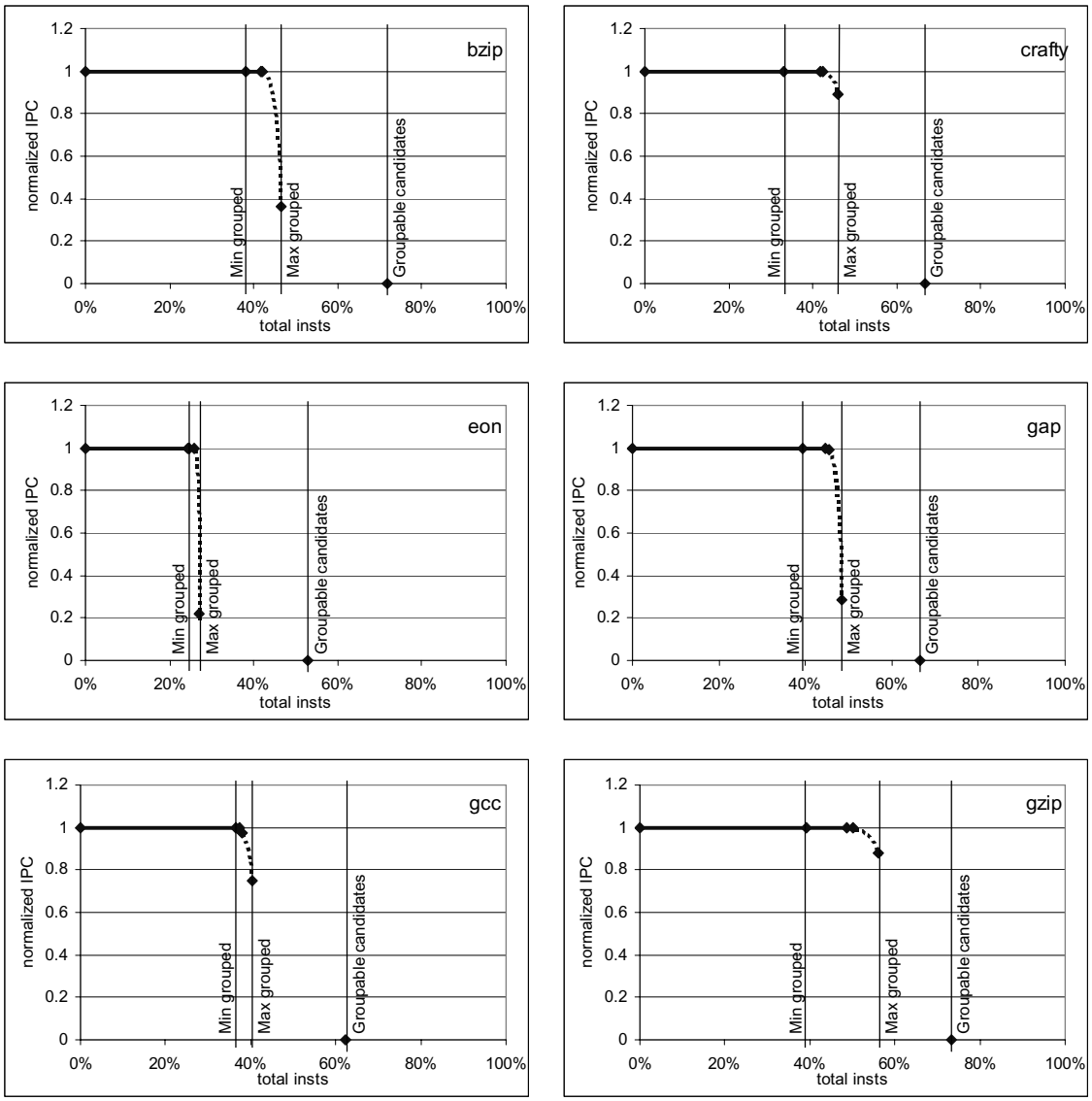


FIGURE 4-10. Coverage of candidate instructions (bzip ~ gzip, measured without load MOP tail).

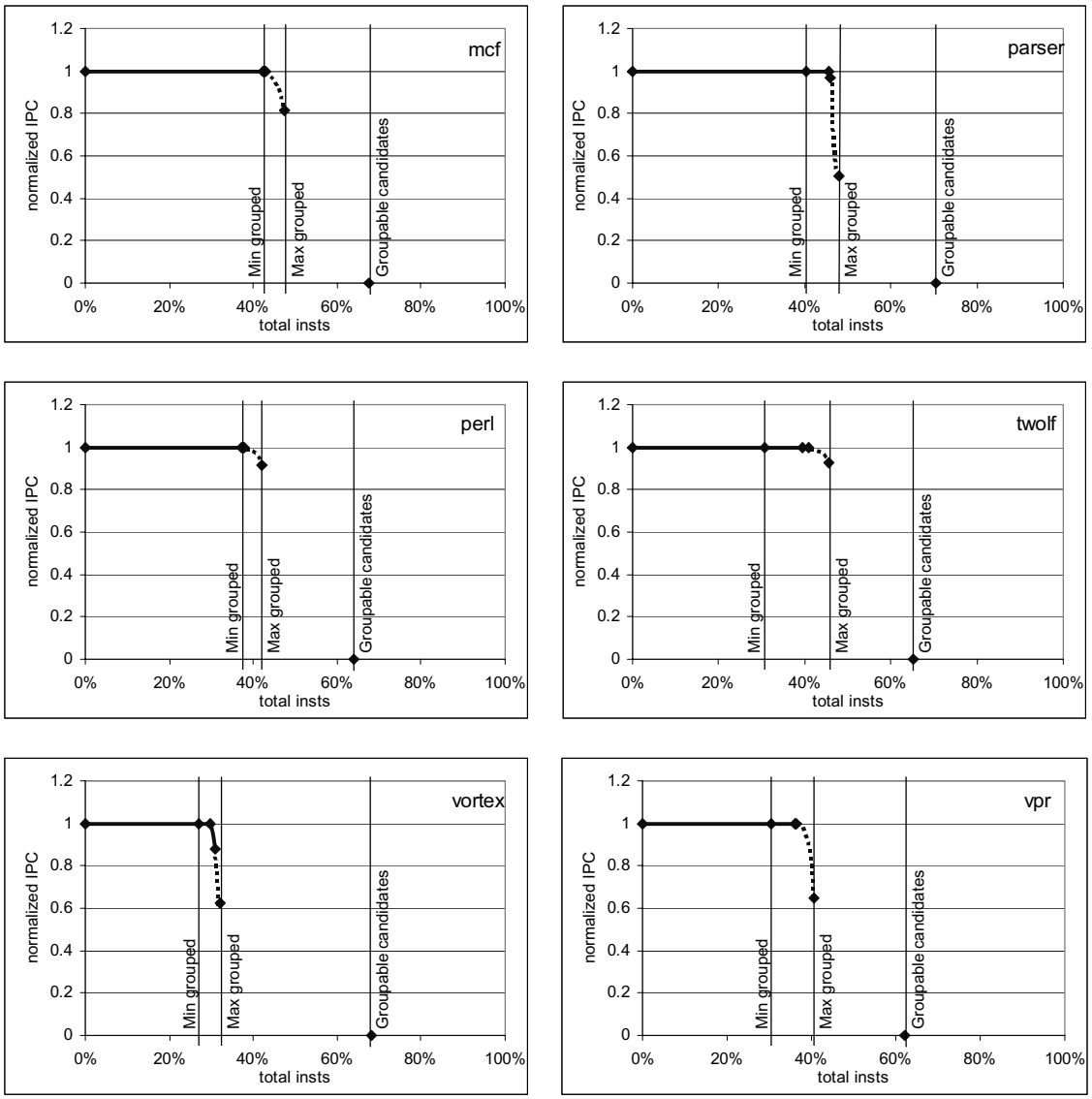


FIGURE 4-11. Coverage of candidate instructions (mcf ~ vpr, measured without load MOP tail).

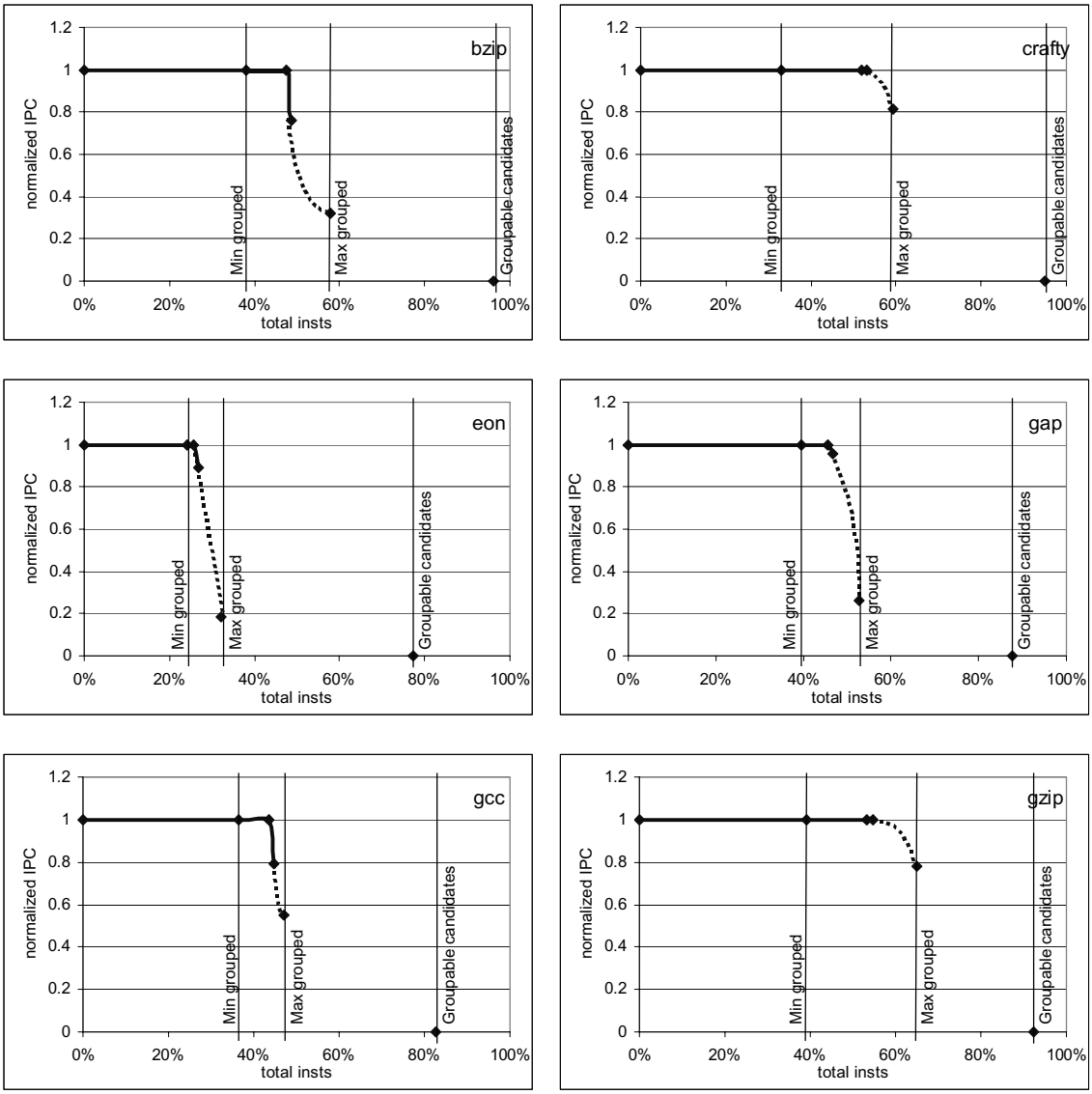


FIGURE 4-12. Coverage of candidate instructions (bzip ~ gzip, measured with load MOP tail).

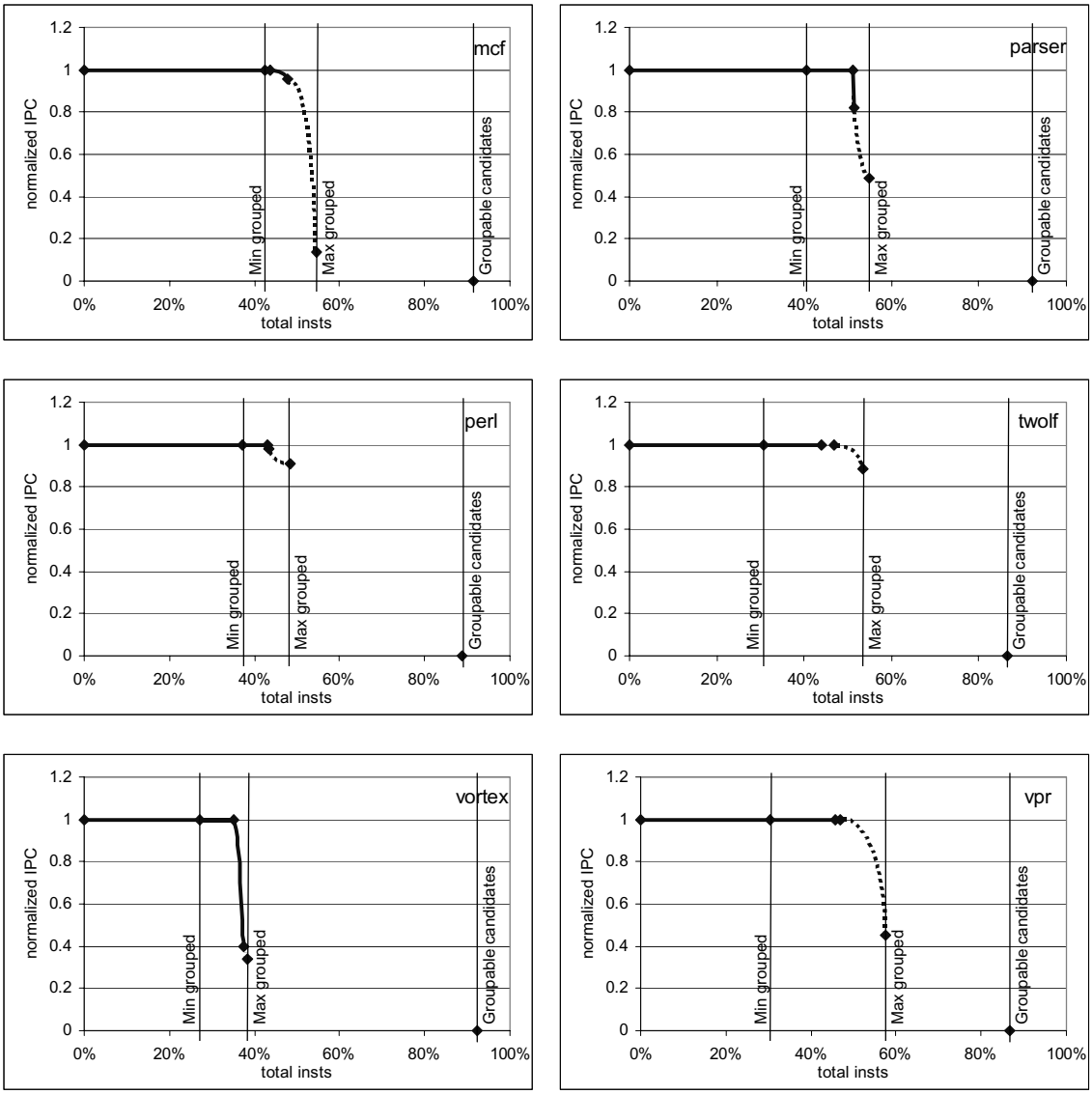


FIGURE 4-13. Coverage of candidate instructions (mcf ~ vpr, measured with load MOP tail).

4.6.2 MOP size distribution

Figure 4-14 and Figure 4-15 present percentages of instructions grouped in different MOP sizes, with and without groupable loads. The number of instructions in a MOP is limited to eight instructions collected within 32 instructions. Each benchmark has three stacked bars to show MOP size distributions measured by *min grouped*, *no EXE timing change*, and *max grouped* policies discussed in Section 4.5.3. The 100% point in the y-axis represents the total instructions grouped using the *max grouped* policy. The number of grouped instructions in the other two grouping policies is normalized to the *max grouped* policy so that the differences in instruction counts (presented in Figure 4-10 through Figure 4-19) can be observed. Each stacked bar has 7 categories from 2x (two instructions grouped in a MOP) to 8x (eight instructions grouped in a MOP). Note that the y-axis does not represent the number of MOPs but instructions in each category, so the actual number of MOPs in e.g. the 8x category is four times lower than that of the 2x category.

As briefly discussed in Section 4.6.1, the *min grouped* policy captures relatively short chains (two instructions in most cases) due to the restrictions for dependence merging points. The *no EXE timing change* and *max grouped* policies can capture more instructions, which results in increases in 3x ~ 8x categories. Still, the 2x category is dominant in most benchmarks except for *gap*, which contains chains of more than three instructions in many cases. This result is also correlated to the dependence edge distance (Figure 4-7 and Figure 4-8); short distances between dependent instructions increase the likelihood of capturing more instructions in a finite detection scope. Similarly, *vortex* has long dependence edges and relatively small MOP sizes dominate the distribution, compared to other benchmarks.

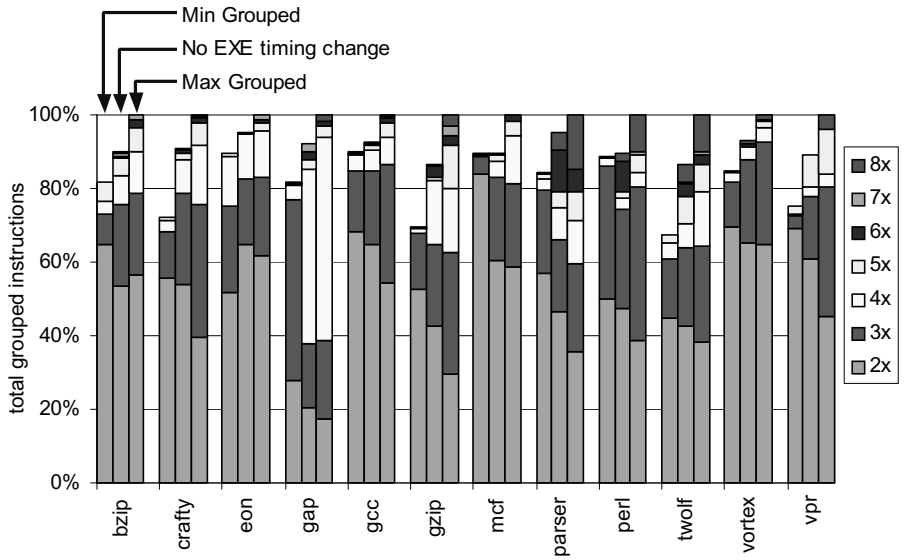


FIGURE 4-14. MOP size distribution (measured without load MOP tail).

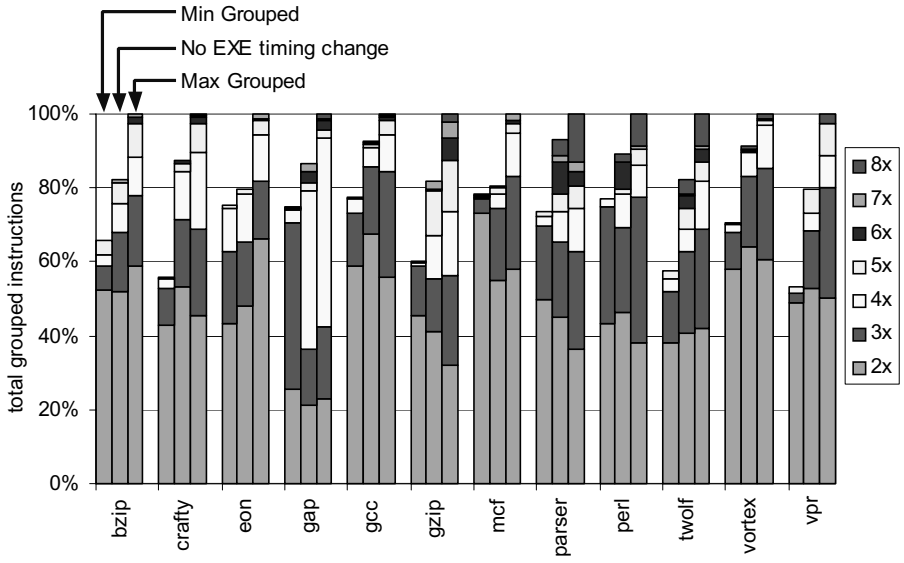


FIGURE 4-15. MOP size distribution (measured with load MOP tail).

Table 4-4: Average number of instructions grouped in a MOP (8x, 32-inst scope).

Benchmarks	with load MOP tail		without load MOP tails	
	no EXE timing change	max grouped	no EXE timing change	max grouped
bzip	2.41	2.48	2.39	2.47
crafty	2.39	2.68	2.40	2.66
eon	2.30	2.39	2.39	2.35
gap	3.18	3.27	3.14	3.15
gcc	2.27	2.45	2.24	2.44
gzip	2.56	2.97	2.63	3.02
mcf	2.28	2.43	2.27	2.43
parser	2.70	3.01	2.70	2.96
perl	2.53	2.74	2.56	2.74
twolf	2.64	2.88	2.62	2.78
vortex	2.25	2.32	2.27	2.38
vpr	2.32	2.62	2.33	2.55

Table 4-4 shows the average number of instructions in a MOP captured with a configuration of 8x, 32-instruction scope. Note that the average numbers with loads tend to be lower than those without loads. This is because a load terminates grouping and does not allow any more instructions to be grouped together.

The average numbers do not exceed three instructions in most cases except for *gap* and *gzip*. This result is also consistent with dependent strand length measured in [53], although the chain length in a MOP tends to be shorter than the strand length because MOPs cannot group across loads. This implies that building a microarchitecture that focuses on groups of two instructions can yield most of benefits. Potentially, 3x ~ 8x MOPs presented in the graphs can be broken into multiple 2x MOPs and hence 2x MOP grouping should not severely affect its coverage, although it may lose some opportunities for “leftovers” if a dependence chain contains an odd number (e.g. three, five or seven) of instructions.

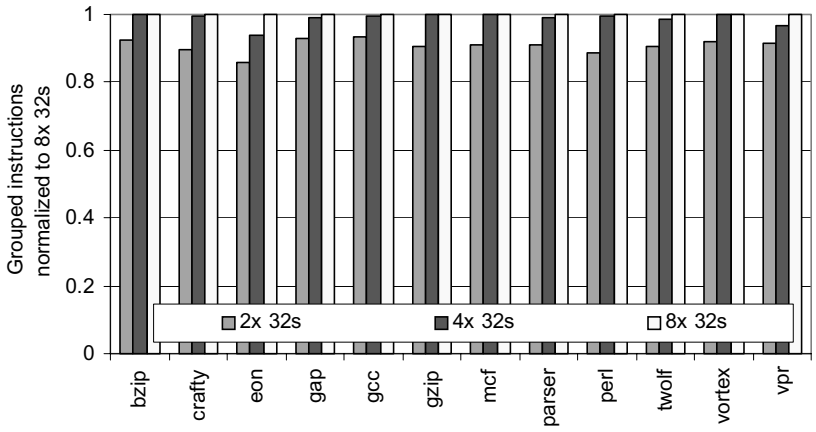


FIGURE 4-16. Impact of MOP sizes on MOP coverage (measured without load MOP tail).

Figure 4-16 and Figure 4-17 show the impact of MOP sizes on the number of grouped instructions. Each benchmark has three bars that represent 2x (two instructions only), 4x (up to four instructions) and 8x (up to eight instructions) configurations given a 32-instruction scope. The *no EXE timing change* policy is used for all cases. The y-axis represent the number of grouped instructions normalized to the *8x 32s* configuration (grouping up to eight instructions in a 32-instruction scope). In general, capturing only 2x MOPs loses only ~10% of instructions grouped, compared with the 8x configuration.

From the perspective of exploiting the benefits of grouping instructions, bigger MOPs will enable greater benefits from fewer schedulable units and increased scheduling determinism (discussed in Section 4.2). From the perspective of implementation, creating and handling equally sized MOPs is important due to their regularity. Unfortunately, creating a sufficient number of equally sized, large MOPs is hard due to short dependence chains in most benchmarks. For example, ~30% fewer instructions are captured in *gap* if grouping allows only 3x MOPs, although the average chain length is around three instruc-

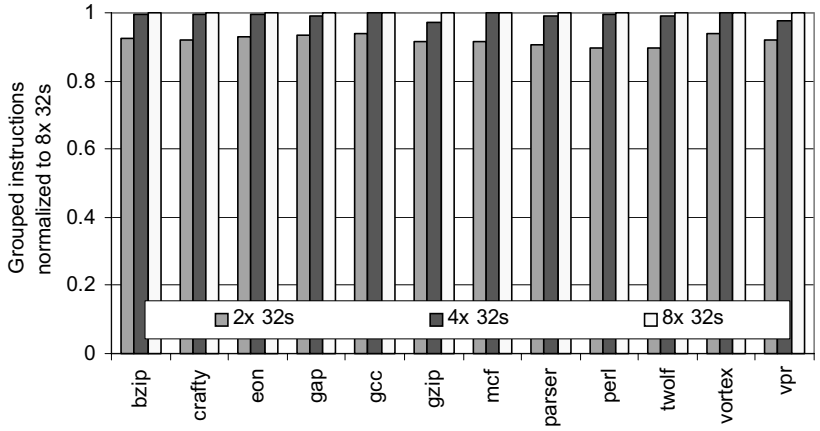


FIGURE 4-17. Impact of MOP sizes on MOP coverage (measured with load MOP tail).

tions. Therefore, MOP size should be carefully determined to maximize benefit by considering both dependence chain length and regularity in hardware implementation.

In summary, the results in this chapter indicate that 1) chains of dependent instructions groupable into MOPs tend to be short and the vast majorities are two instructions, and 2) allowing only two instructions in each MOP does not severely degrade coverage.

4.6.4 Impact of MOP scope on MOP coverage

Figure 4-18 and Figure 4-19 present the impact of MOP scope on the number of grouped instructions, with and without load MOP tail. Each benchmark has three bars that represent MOP scope of 8, 16 and 32 instructions, respectively. The *no EXE timing change* policy is used for all cases. The y-axis represent the number of grouped instructions normalized to the *8x 32s* configuration.

A MOP scope of eight instructions (*8x 8s*) is measured to lose some grouping opportunities, ranging from 35.7% (*eon*) to 10.7% (*gap*) without load MOP tail. With load MOP tail, the degree of reduction is alleviated but substantial in some benchmarks. However, an 8-instruction scope still captures a significant portion of candidate instructions

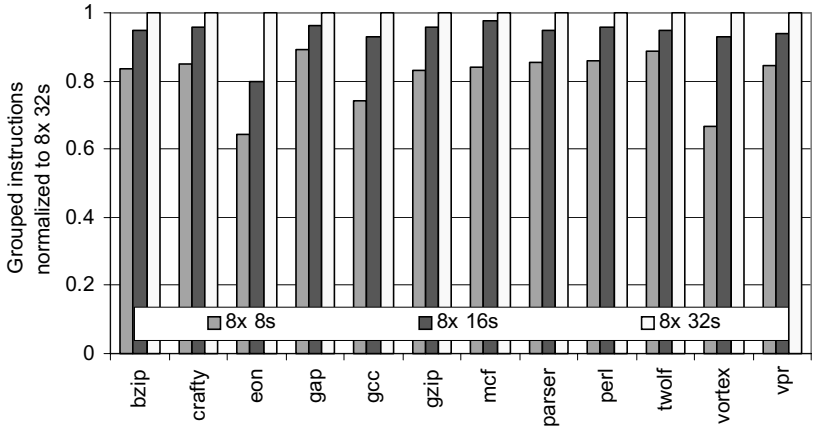


FIGURE 4-18. Impact of MOP scope on MOP coverage (measured without load MOP tail).

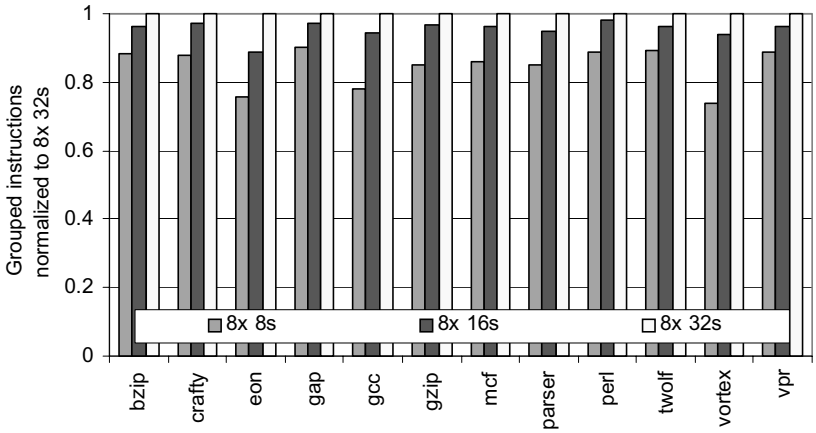


FIGURE 4-19. Impact of MOP scope on MOP coverage (measured with load MOP tail).

because dependent instructions tend to be placed near each other within a short range. Again, this result correlates with the dependence edge distance (Figure 4-7 and Figure 4-8); short distances between dependent instructions increase the likelihood of capturing more instructions in a finite detection scope. *Eon* and *vortex* have relatively long dependence edges and are most sensitive to MOP scope.

As one might expect, a MOP scope of 16 instructions (*8x 16s*) captures more candidate instructions than the *8x 8s* configuration. Although wider MOP scopes will be more

beneficial in that they increase the coverage of candidate instructions, the complexity of detecting groupable candidate instructions should be also considered, since wider scopes require more instructions to be buffered and examined.

4.7 Summary and Conclusions

This chapter defines macro-op and describes the implications of grouping instructions and processing them as single units, which can be summarized as dependence abstraction and increased scheduling determinism.

Several issues in grouping instructions are discussed. Macro-ops may potentially reduce instruction-level parallelism by serializing unnecessary instructions. To avoid these harmful macro-ops, grouping dependent instructions into macro-ops is beneficial, as opposed to grouping independent instructions. Special considerations should be made at dependence branching and merging points for negative effects of last-arriving operands. Also, cycles in data dependence chain may be induced by improper grouping, and hence should be avoided.

To estimate the potential for macro-op grouping, candidate instruction types are first determined. Then, dependence edge distances between groupable candidate instructions are characterized. The results from this characterization indicate that many dependent macro-op candidates are placed near each other within a few instructions in program order. Based on an infinite machine model that measures the parallelism of programs, the degree of groupability and its performance impact are measured. The results indicate that a significant number of instructions can be processed in groups. Also, they show that carelessly grouping instructions may significantly degrade parallelism.

To determine the size of macro-ops, the average number of candidate instructions in a dependence chain is measured. Our data indicate that dependence chains tend to be short and that macro-ops with two instructions account for the vast majority of grouping opportunities. The impact of macro-op scope on groupability is also measured. As more instructions are searched in a wider scope, more macro-ops are captured and the coverage increases. However, an 8-instruction macro-op scope captures a significant portion of candidate instructions in many cases, compared to wider macro-op scopes.

The groupability of instructions is related to the dependence edge distance among dependent candidate instructions. Benchmarks with long dependence edges (e.g. *eon* and *vortex*) are measured to have low groupability. In contrast, some other benchmarks with short dependence edges (e.g. *gap*) exhibit relatively high potential for macro-op grouping.

Based on these results, the base grouping policy for MOP size and scope that our experiments with macro-op scheduling and execution focus on is determined to be grouping two instructions captured within an 8-instruction scope. However, other policies for larger MOPs or wider scope will be also discussed and evaluated.

These attributes of macro-ops can be exploited to improve instruction scheduling logic and increase machine bandwidth. The details of such techniques will be discussed in Chapter 6 and Chapter 8.

Chapter 5

Understanding the Effects of Pipelined Scheduling

Conventional instruction scheduling logic performs a set of wakeup and select operations atomically every clock cycle, to ensure back-to-back execution of dependent instructions. If pipelined over multiple stages, it loses the capability for issuing dependent instructions consecutively, resulting in performance degradation. One major benefit of coarse-grained serial instruction processing, which is realized by macro-op scheduling and execution in this thesis, is to reduce the rate at which scheduling decisions must be generated. This enables pipelined instruction scheduling logic to reap most of the performance of conventional atomic scheduling.

To understand how our techniques compensate for the negative effects of pipelined instruction scheduling, this chapter examines several aspects of pipelined instruction scheduling and provides an insight into the reasons for the variability in performance sensitivity to 2-cycle scheduling observed across different programs.

This chapter is laid out as follows: Section 5.1 describes the problems with conventional instruction scheduling and discusses the trade-offs between scheduling atomicity and scalability. Section 5.2 discusses the variability in the performance impact of pipelined instruction scheduling. Section 5.3 and Section 5.4 study the hardware constraints and program characteristics that cause the machine to be tolerant of the negative effects of pipelined instruction scheduling. Section 5.5 presents the correlation between the distance distribution of dependence edges and the performance impact of pipelined instruction scheduling.

5.1 Atomicity vs. Scalability

Ensuring back-to-back execution of dependent instructions in a conventional out-of-order processor requires scheduling logic that performs a set of wakeup and select operations every clock cycle. This *scheduling atomicity* is a major obstacle to building high-frequency out-of-order microprocessors because it prevents the conventional instruction scheduling logic from being pipelined over multiple stages. Its complexity is primarily determined by the size of instruction window (i.e. number of entries in the issue queue). To achieve the target frequency without losing scheduling atomicity, its scalability should be constrained; the issue queue should be limited to a certain size, which may result in low architectural performance (i.e. IPC) because of the low capability for extracting parallelism from the program.

If scheduling logic is pipelined, i.e. performs wakeup and select operations in two different stages, it loses the ability to issue dependent instructions consecutively, and hence may degrade performance since it creates scheduling bubbles in the dependence chain (through register dependences) and the critical path of the program is potentially lengthened. On the other hand, a positive effect is anticipated; the logic complexity is distributed over multiple pipeline stages and therefore a wider instruction window can be built to search for more instructions to issue, which may recover the performance degradation, or enable even better performance.

To evaluate these trade-offs, Figure 5-1 shows the average IPCs of SPEC2K integer benchmarks measured on the base machine with atomic and pipelined scheduling logic. The details of the machine configuration (4-wide, 128-entry issue queue and ROB) are presented in Figure 3-1. The atomic scheduling logic performs a set of wakeup and

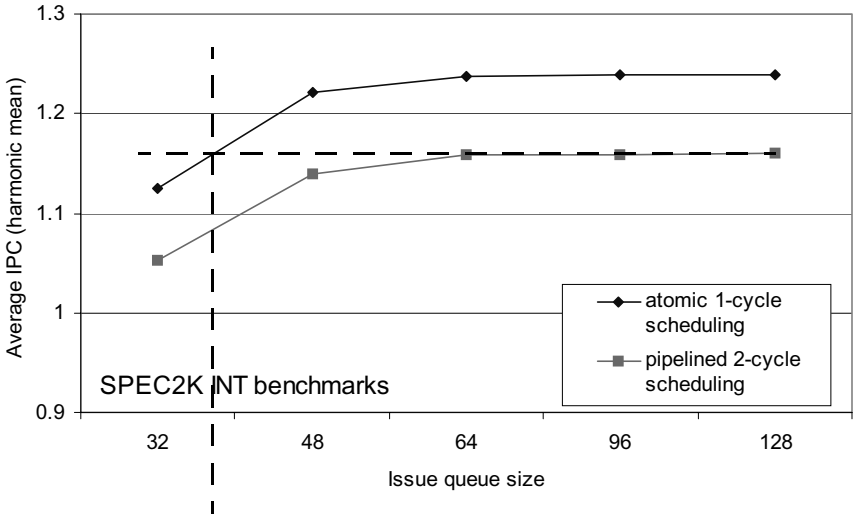


FIGURE 5-1. Performance of pipelined scheduling logic with various issue queue sizes.

select operations every clock cycle to enable back-to-back execution of dependent instructions (*1-cycle scheduling*). The pipeline scheduling logic performs a set of wakeup and select operations every two cycles (*2-cycle scheduling*) and hence is unable to issue instructions dependent on parent instructions with single-cycle execution latency. Note that it does not introduce scheduling bubbles for other multi-cycle instructions because they can be hidden behind the execution latency.

With a 32-entry issue queue, the IPC loss due to 2-cycle scheduling is measured to be 6.6% on average. If we assume that pipelining allows the issue queue size to grow to 48 entries, it outperforms the 32-entry, 1-cycle scheduling case. However, its equivalent window size is located at a point slightly greater than 32 entries, and even a 128-entry window is unable to achieve the performance of 1-cycle scheduling with 48 entries. Recovering the performance loss due to 2-cycle scheduling requires a significantly larger instruction window to find more instructions to issue. However, increasing the window size needs not only a larger issue queue structure, but also the pipeline to bring in more instructions

Table 5-1: Performance of 2-cycle scheduling with larger issue queue and ROB.

Benchmarks	IPC with 1-cycle scheduling, 48 IQ, 128 ROB	IPC with 2-cycle scheduling, 256 IQ, 256 ROB	IPC loss
bzip	1.51	1.48	2.3%
crafty	1.54	1.57	-1.9% (speed up)
eon	2.13	2.11	0.7%
gap	1.98	1.91	3.3%
gcc	1.29	1.28	0.9%
gzip	1.96	1.64	16.5%
mcf	0.37	0.42	-13.2% (speed up)
parser	1.12	1.01	10.0%
perl	1.30	1.33	-2.2% (speed up)
twolf	1.47	1.35	8.3%
vortex	1.74	1.83	-5.3% (speed up)
vpr	1.61	1.48	8.0%
Average (Harmonic mean)	1.2213	1.2232	-0.16% (speed up)

because insufficient instruction supply prevents the large window from being fully utilized. The IPC curves in Figure 5-1 become saturated at 64 entries in both cases because of this effect. Improving the instruction supply potentially requires higher fetch bandwidth and a larger reorder buffer and register file.

Table 5-1 presents the 2-cycle scheduling performance when both reorder buffer and issue queue are scaled to 256 entries to achieve a wider instruction window, while the 1-cycle scheduling case still has the 128-entry reorder buffer and the 48-entry issue queue. Although the average IPC indicates that 2-cycle scheduling now outperforms 1-cycle scheduling due to a larger instruction window powered by the increased reorder buffer size, still, many benchmarks (e.g. 16.5% performance drop in *gzip*) cannot recover the performance gap between the two cases.

In summary, pipelining instruction scheduling logic incurs negative performance impact, which may not be easily recovered by increasing the instruction window size.

Table 5-2: IPC loss due to 2-cycle scheduling at each issue queue size.

Benchmarks	32-entry	48-entry	64-entry	128-entry
bzip	6.9%	6.7%	6.5%	6.4%
crafty	5.0%	3.9%	3.6%	3.5%
eon	2.7%	3.5%	2.9%	2.8%
gap	17.7%	21.0%	19.9%	19.1%
gcc	2.6%	2.7%	2.7%	2.7%
gzip	17.0%	18.2%	18.2%	18.1%
mcf	1.5%	1.5%	1.4%	1.4%
parser	11.4%	11.7%	11.5%	11.6%
perl	5.0%	3.2%	2.8%	1.5%
twolf	10.7%	10.8%	10.8%	10.8%
vortex	2.5%	1.6%	1.3%	1.3%
vpr	9.4%	10.7%	10.1%	9.8%
Average IPC loss	6.55%	6.75%	6.47%	6.33%

Therefore, it requires some other way to overcome this limitation.

5.2 Variability in Performance Sensitivity to 2-cycle Scheduling

Before I discuss the ways to overcome the limitation of pipelining the instruction scheduling logic, it is necessary to analyze the causes for its performance loss. To simplify our discussion, I first want to define *n-cycle dependence edge*, which is an output register dependence edge from a parent instruction with n-cycle execution latency, to its dependent child instruction. An n-cycle instruction has as many n-cycle dependence edges as the number of its consumers. As discussed in the previous section, 2-cycle or greater dependence edges are not affected by 2-cycle scheduling. In contrast, 1-cycle dependence edges are vulnerable to 2-cycle scheduling in which they behave as if they are 2-cycle dependence edges and increase the dependence tree height due to scheduling bubbles. Note that 1-cycle instructions that create 1-cycle dependence edges are MOP head candidates defined in Section 4.5.1.

Table 5-2 presents IPC loss due to 2-cycle scheduling measured for each benchmark when the issue queue size varies from 32 to 128 entries. Note that the performance data here are identical to those of Figure 5-1. An interesting trend exists in the table: the average numbers in the last row are not representative for the performance losses across all benchmarks. Some benchmarks (highlighted rows) such as *gap, gzip, parser, twolf* and *vpr* exhibit significant performance degradations over 10% (up to 21% in *gap*), while the others are nearly insensitive or less sensitive to 2-cycle scheduling.

The performance insensitivity of some benchmarks is, of course, because many 1-cycle dependence edges are not in the critical path of the program [31], which is formed through non-1-cycle dependence edges or other structural dependences created by hardware constraints. In the following sections, likely causes for the performance insensitivity will be discussed.

5.3 Performance Insensitivity Caused by Hardware Constraints

Hardware constraints that may cause the performance insensitivity can be listed as follows:

- *Branch prediction performance*: frequent branch mispredictions increase the total execution time. Therefore, the performance degradation due to 2-cycle scheduling become less significant compared with the total execution time.
- *Memory performance*: instruction and data cache misses cause the same effect as branch mispredictions. In addition, data cache misses may hide other computations sensitive to 2-cycle scheduling.
- *Execution bandwidth*: if a program has many independent ready instructions that

exceed the execution bandwidth, the structural hazards in the execution bandwidth and resources may hide the performance degradation due to 2-cycle scheduling.

- *Fetch bandwidth*: if a child instruction is not delivered into the pipeline in time before its parent instruction is issued due to a limited fetch bandwidth, 2-cycle scheduling may not affect performance because the child instruction is already ready to issue when entering the instruction window.

In order to evaluate how much performance degradation is hidden by such constraints, each benchmark was tested on the base machine (4-wide, 128 IQ, 128 ROB) with 1- and 2-cycle scheduling as each hardware constraint is relaxed cumulatively in the order of branch prediction, memory performance, execution bandwidth and fetch bandwidth. In Figure 5-2 through Figure 5-4, the left graphs present the total execution time differences between 1- and 2-cycle scheduling, to show the extra cycles induced by 2-cycle scheduling. The right graphs present the same data as the left ones, but show IPCs normalized to 1-cycle scheduling. Note that the unnormalized IPCs are shown in Table 9-2. The reason for showing two different graphs is to identify the cause of performance insensitivity. For example, relaxing a certain hardware constraint may increase the performance degradation for 2-cycle scheduling. If the execution time gap between 1- and 2-cycle scheduling is stable (observed from the left graph), the relative performance drop (right graph) is caused by the reduction in the total execution time. If the execution time gap has increased, the result implies that the extra delays incurred by 2-cycle scheduling have been hidden behind other delays incurred by the hardware constraint.

The *base* category in each graph shows the base performance with no hardware constraint relaxation (same as the *128-entry* column in Table 5-2). The *bpred* category

shows the case where perfect branch prediction is added. The next *mem* category further relaxes the constraint of memory performance. Perfect instruction and data caches (with the same access latency as the base case) are used on top of the perfect branch predictor. Note that the processor core still experiences scheduling replays due to store-to-load aliases although no data cache miss causes loads to replay. From the graphs, we find that relaxing branch prediction and memory constraints exposes performance degradation due to 2-cycle scheduling in *gap, parser* and *perl*. In particular, *perl* was relatively insensitive to 2-cycle scheduling on the base machine but it loses over 10% of IPC when the two constraints are relaxed. Note that the execution time gap (left graph) in *perl* has increased after relaxing the two constraints, whereas those in *gap* and *parser* are stable. This implies that the exposed performance degradation in *perl* did not simply come from the reduced total execution time. Also note that *mcf* is insensitive to 2-cycle scheduling not simply because of low cache performance. Although perfect memory improves its performance by more than seven times, 2-cycle scheduling loses only a small fraction of the IPC. A similar trend is also observed in other insensitive benchmarks such as *eon, gcc* and *vortex*.

Execution bandwidth limits are relaxed in the *exe* category. In addition to the previous relaxations, the base machine is configured to have infinite execution resources (functional units and memory ports) as well as infinite issue and execution bandwidth. Therefore, the machine can issue and execute instructions any time as long as they are ready. Except for *gap*, the execution time difference between 1- and 2-cycle scheduling in most benchmarks are stable, implying that execution bandwidth and resource conflicts are not major causes of the insensitivity. One reason for this result is that the machine cannot fully utilize infinite execution bandwidth because of the finite fetch bandwidth.

The *fetch / commit* category assumes infinite fetch and commit bandwidth so that the ROB (128 entries) is always full of instructions. Now, performance is constrained only by the data dependences (determined by the code structure) and the window size (determined by the ROB size), ignoring other miscellaneous factors. Instructions are fetched and committed still in the original program order as in the base machine. The reason why the commit bandwidth limit is also relaxed as well is because the machine cannot fetch more instructions than the finite commit bandwidth (four instructions per cycle) which limits the maximum number of ROB entries that can be reassigned to newly fetched instructions. It is important to note that finite execution bandwidth has the same effect on fetch as the commit bandwidth. For this reason, even if the *exe* and the *fetch / commit* categories are measured in reverse order, the performance curves in all benchmarks show the same trends (with minor variations) as the current ones. In other words, relaxing only one or two constraints among fetch, execution and commit does not fully improve the machine bandwidth and therefore the current category should be interpreted as relaxing the constraint of the finite machine bandwidth, not just fetch or commit constraints.

An interesting trend is observed after the finite machine bandwidth constraint is relaxed. The two curves in the left graphs diverge and the execution time gaps increase in many benchmarks. In particular, *crafty, gcc, mcf* and *vortex* become sensitive to 2-cycle scheduling, although the execution time gaps have been relatively constant until before this relaxation. This implies that the performance insensitivity to 2-cycle scheduling comes from a factor related to the machine bandwidth, which determines how fast instructions are delivered to the instruction window.

Finally, the last two categories labeled as *2x window* and *4x window* show the

cases where the size of the ROB and the issue queue increases to 256 and 512 entries, respectively. Although some benchmarks show more performance drops (seen from the right graphs) while the others exhibit an opposite trend as the window size increases, the execution time gaps (which may not seem obvious due to the scale of the left graphs) in all benchmarks are reduced because extra delays from 2-cycle scheduling are overlapped with each other as more parallelism is extracted by a deeper instruction window.

In summary, the degree of the performance insensitivity to 2-cycle scheduling differs significantly from benchmark to benchmark. Some hardware constraints cause programs to be less sensitive to the negative performance impact of 2-cycle scheduling. In particular, machine bandwidth that determines how fast instructions are delivered to the instruction window greatly affects the performance sensitivity.

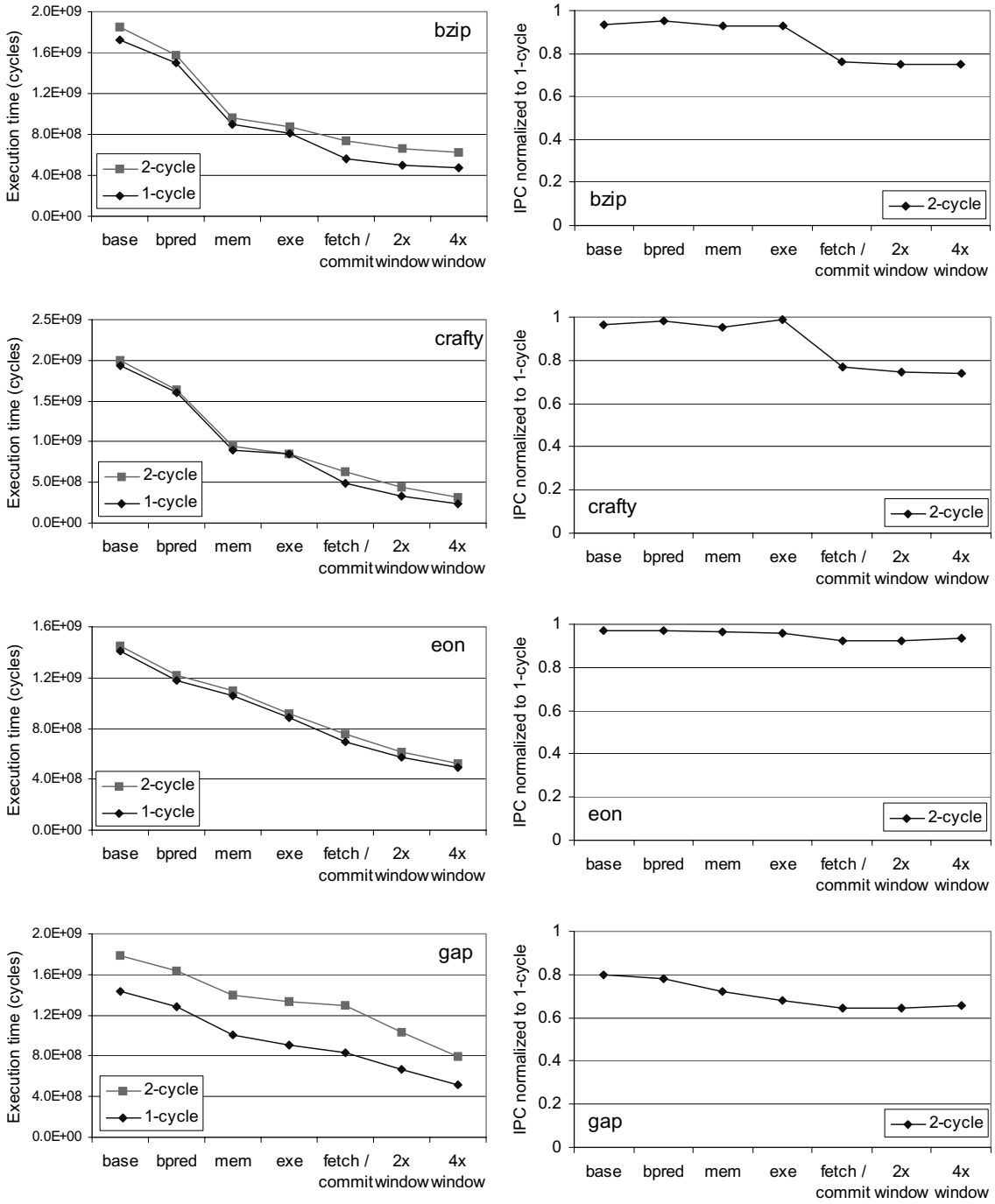


FIGURE 5-2. Execution time and relative performance of 2-cycle scheduling as the machine constraints are relaxed (bzip ~ gap).

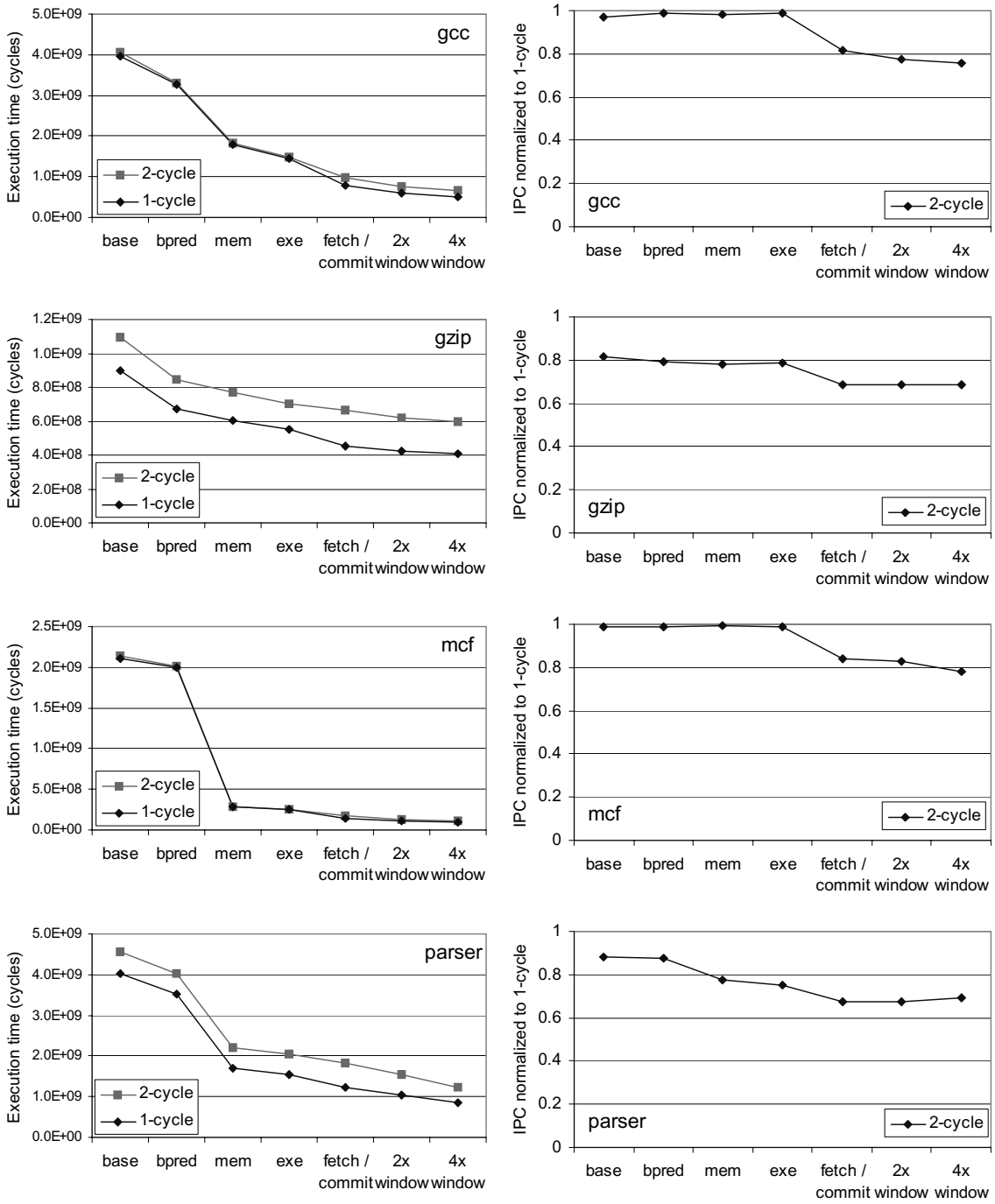


FIGURE 5-3. Execution time and relative performance of 2-cycle scheduling as the machine constraints are relaxed (gcc ~ parser).

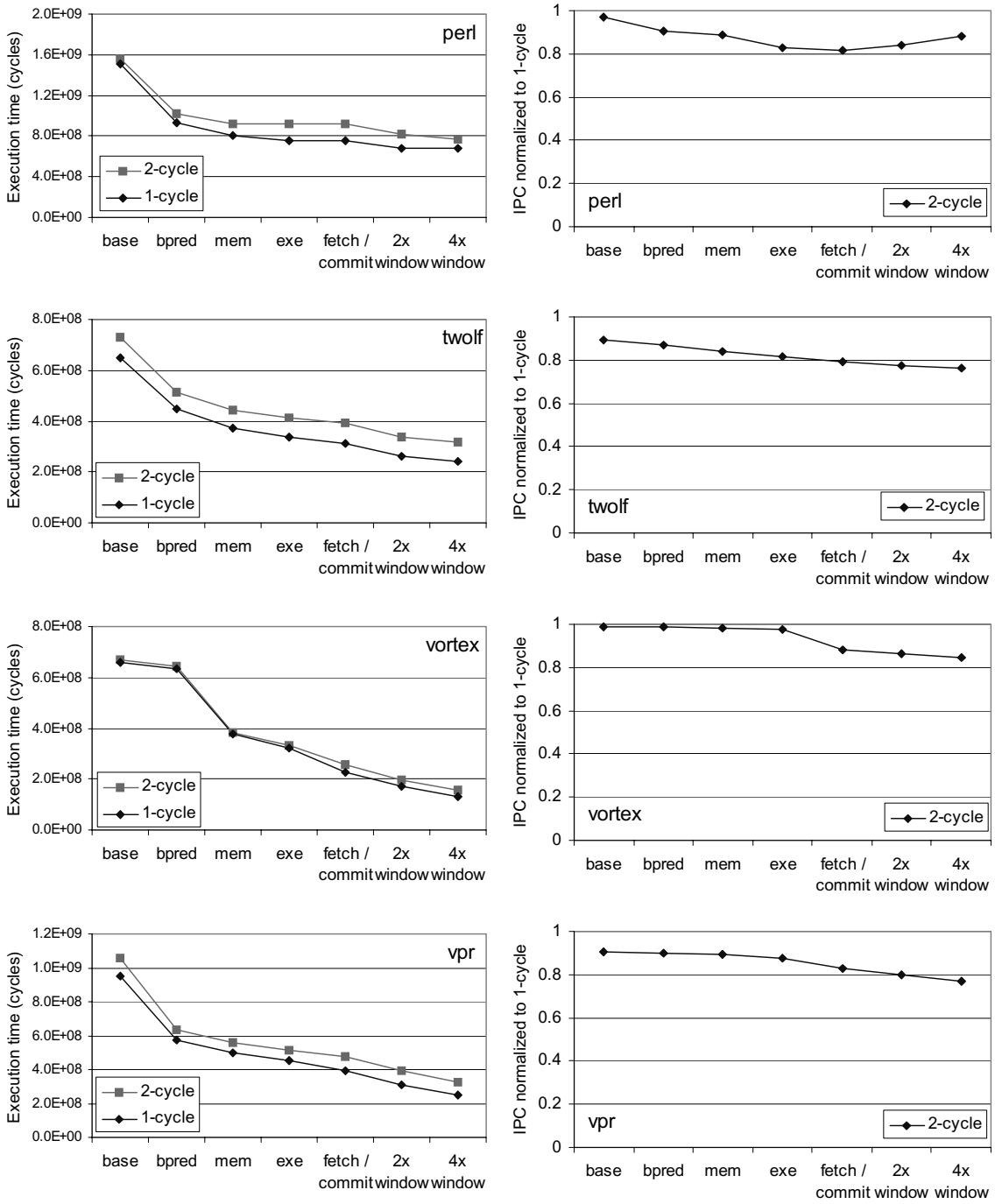


FIGURE 5-4. Execution time and relative performance of 2-cycle scheduling as the machine constraints are relaxed (perl ~ vpr).

5.4 Performance Insensitivity Caused by Program Characteristics

In the previous section, we discussed the performance impact of hardware constraints on 2-cycle scheduling and found that machine bandwidth greatly affects the performance sensitivity. The next question will be which program characteristic strengthens or weakens such trends given the hardware constraints.

Instruction mix may partly explain the performance insensitivity of *eon*, which has the lowest 1-cycle integer instruction rate among the benchmarks as shown in Table 3-2 (*% 1-cycle ALUs* category). Since there are fewer 1-cycle instructions in the program than others, it is also likely that fewer instructions in the critical path of the program are negatively affected by 2-cycle scheduling, assuming that a program and its critical path have a similar instruction mix. Other benchmarks relatively insensitive to 2-cycle scheduling, such as *gcc*, *mcf* and *vortex* also have relatively lower percentages of 1-cycle instructions than other sensitive benchmarks. However, their differences are within 10 ~ 15% of total instructions so they do not fully explain the observed results, although they are correlated to some extent.

It is a difficult task to find the exact causes of performance insensitivity from program characteristics, since the program structure is not simply defined by only a few parameters. Instead of formulating a statistical model to predict the performance on 2-cycle scheduling using multiple parameters that describe the program characteristics, I will focus on one parameter -- dependence edge distance, and try to find its correlation with performance, because it is directly related to what the proposed macro-op scheduling complements for 2-cycle scheduling.

According to the critical path model [31], the critical path is formed through the

data dependences (i.e. memory and register dependences) as well as structural dependences (i.e. hardware constraints discussed in the previous section). To measure the impact of data dependences on the performance insensitivity, a simple experiment is performed. Figure 5-5 compares the performance of 2-cycle scheduling measured on the infinite machine and the base machine with a finite bandwidth. The performance data are normalized to the 1-cycle scheduling case on each machine. For controlled experiments, the base machine has a perfect branch predictor and perfect memory (same as the *mem* category in Figure 5-2 through Figure 5-4) so that the base machine result is not affected by those hardware constraints. The infinite machine does not have any hardware constraint so the critical path will be formed only through the data dependences (as opposed to structural dependences). If performance sensitivity is determined primarily by the data dependences, we should observe a similar performance trend in both infinite and base machines, although the degree of slowdown may differ. However, the graph shows that there is no correlation between the two results. Especially, the benchmarks that show virtually no slowdowns on the base machine, such as *eon, gcc, mcf* and *vortex* actually exhibit more performance degradation on the infinite machine. Therefore, we can infer that performance sensitivity to 2-cycle scheduling is primarily determined by other factors, i.e. structural dependences.

5.4.1 Not all dependences are created equal

At this point, it is worth highlighting the difference between the infinite and base machines. Infinite machine bandwidth implies that all register and memory dependence edges in the program equally affect performance regardless of their distance (the instruction count between the parent and the dependent child instructions in program order). In

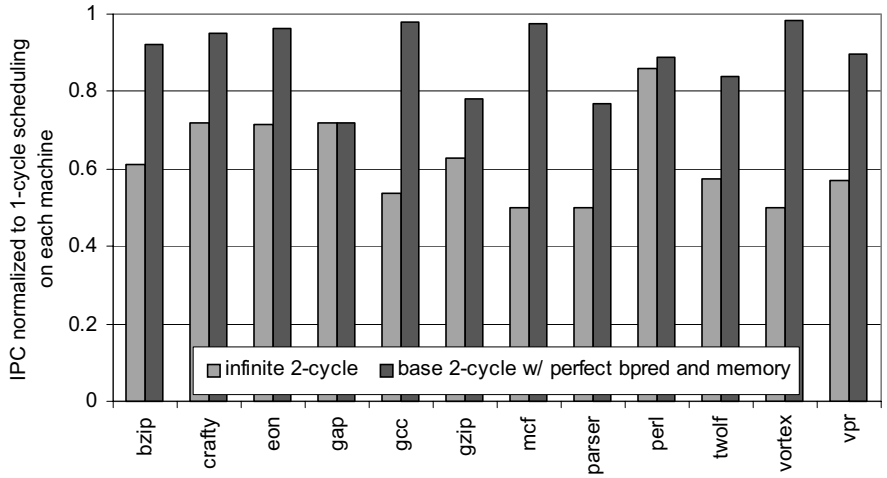


FIGURE 5-5. Impact of 2-cycle scheduling on infinite and base machines.

fact, the finite issue queue and ROB sizes limit the scope of dependence edges visible within the out-of-order window. For example, a dependence edge with a distance greater than 128 instructions never affects performance on a machine with the 128-entry ROB, no matter whether the instruction scheduler performs 1-cycle or even 10-cycle scheduling since the parent instruction should have committed before its dependent instruction enters the window. Even if the dependence edge is shorter and both parent and dependent instructions may fit within the ROB, longer-distance dependence edges are less likely to affect performance than shorter-distance edges because instruction fetch is a serial process and the parent instruction may have been issued ahead of the arrival of its dependent instructions into the window. In this case, this dependence edge becomes insensitive to 2-cycle scheduling because the critical path will not be formed through this edge.

In order to measure this impact of edge distances on instruction scheduling, I first characterize the cumulative distribution of 1-cycle dependence edges categorized by their distances in Figure 5-6. Note that this graph shows a program characteristic, and is not dependent on machine configuration. Also note that the curves in the graph corresponding

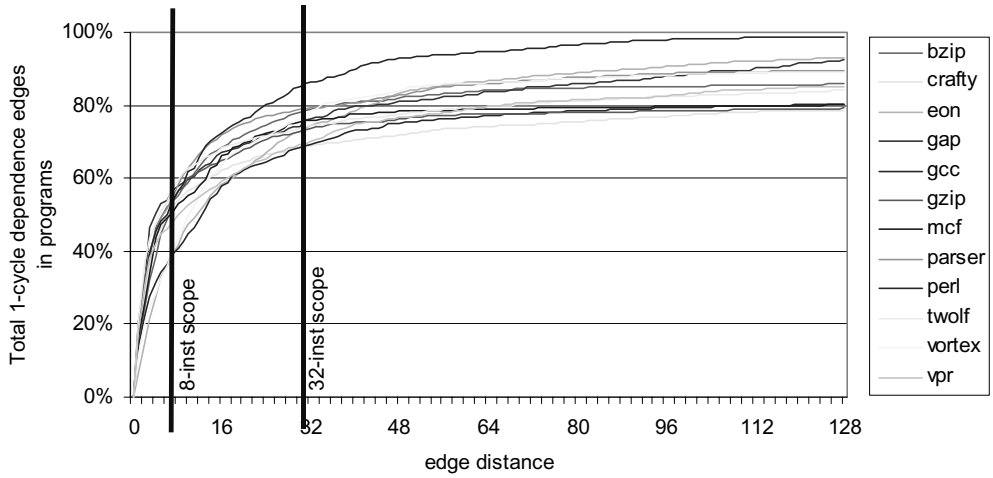


FIGURE 5-6. Cumulative distribution of 1-cycle dependence edges defined in programs.

to each of the benchmarks may not be distinguishable but showing the differences is not important at this point. 100% on the y-axis represents the total 1-cycle dependence edges in a program. The x-axis represents the edge distances, which are shown up to 128 instructions to match the ROB size of the base machine. The graph indicates that 40~58% of 1-cycle dependence edges connect dependent instruction pairs placed within eight instructions in program order (the vertical line labeled as *8-inst scope*). The 32-instruction scope captures at least ~70% of all 1-cycle dependence edges. With a 128-entry ROB, the base machine will not observe other longer edges (up to 20% of all 1-cycle edges). Source operands corresponding to these longer edges should be ready when instructions are fetched into the pipeline, hence not affecting performance.

Now, in Figure 5-7, we will observe the dynamic behavior of 1-cycle dependence edges on the base machine with a perfect branch predictor and perfect memory. The scheduler performs 1-cycle scheduling. The issue queue is set to the maximum size that matches the ROB with 128 entries. The graph is plotted in the same way as Figure 5-6, except that this graph presents the 1-cycle dependence edges that directly awaken their

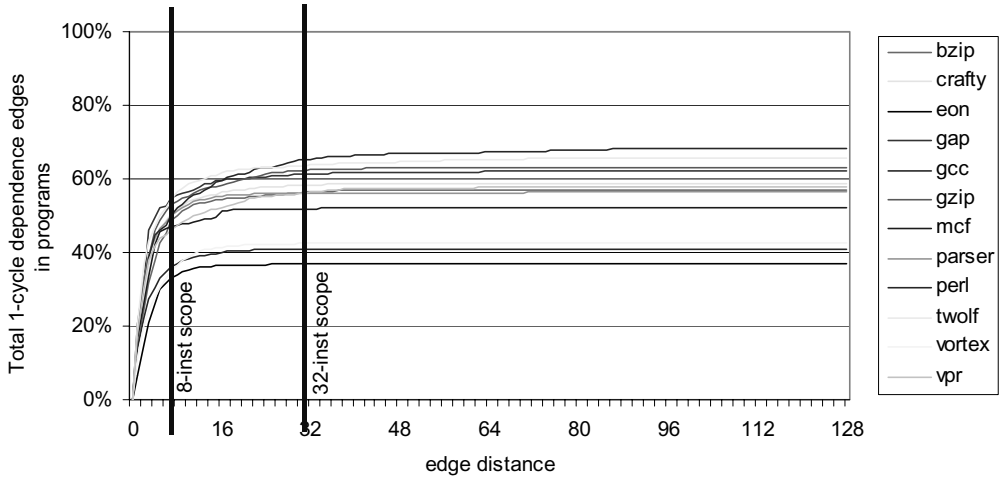


FIGURE 5-7. Cumulative distribution of 1-cycle edges that participate in instruction scheduling.

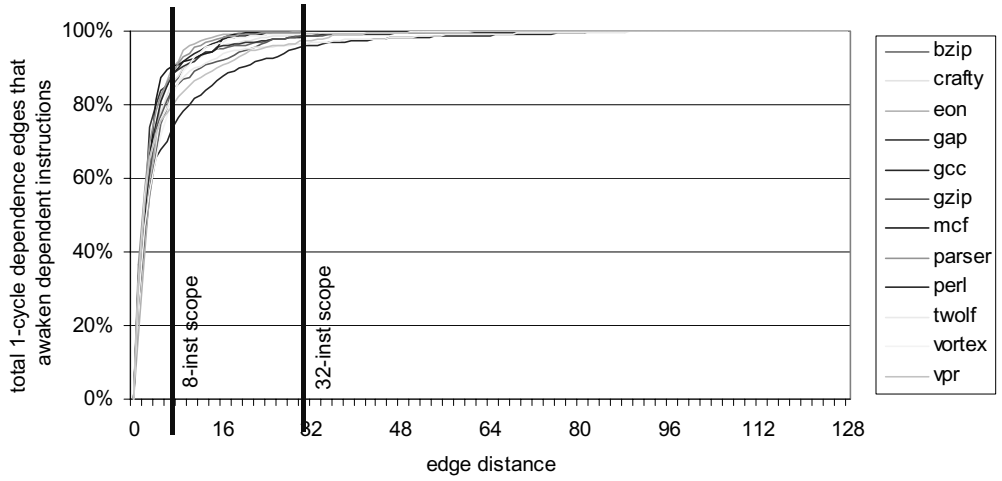


FIGURE 5-8. Cumulative distribution of 1-cycle edges that participate in instruction scheduling (normalized).

dependent instructions in the issue queue (regardless of whether the dependent instruction is issued or not). We find that only a few 1-cycle dependence edges participate in instruction scheduling, comparing to the data in Figure 5-6; only 37% (*eon*) ~ 68% (*perl*) of the total 1-cycle edges awaken their dependent instructions (observed at a distance of 128 instructions). The curves in the graph rise rapidly within a short range of distance and then begin to lose momentum as dependence edges become longer. To better show this behav-

ior, Figure 5-8 presents the data normalized to the total number of 1-cycle dependence edges that awakened dependent instructions. An interesting observation from this graph is that short-distance edges account for most of scheduling activities. Over 80% of 1-cycle dependence edges are captured within an 8-instruction scope in most benchmarks. These short-distance dependence edges are more likely to affect 2-cycle scheduling performance since they are not hidden by structural dependences created by finite fetch bandwidth. Other longer dependence edges do not directly participate in instruction scheduling because dependent instructions are fetched into the out-of-order window after parent instructions have already been scheduled.

5.4.2 Dependence edge distance and performance insensitivity

To present the correlation between 2-cycle scheduling performance and dependence edge distance, I plot the same characterization data as Figure 5-6 for the most sensitive (*gap, gzip, perl, twolf*) and the least sensitive (*eon, gcc, mcf, vortex*) benchmarks using two separate graphs in Figure 5-9 and Figure 5-10.

The graphs clearly show the differences in edge distances between the two groups; insensitive benchmarks tend to have longer dependence edges than sensitive benchmarks. This trend does not apply to *mcf*, which has relatively shorter dependence edges compared to other insensitive benchmarks. Ignoring *mcf*, the sensitive benchmarks in Figure 5-9 have 14 ~ 17% more dependence edges captured within an 8-instruction scope than the other group in Figure 5-10. Note that the vertical line denoted as *8-inst scope* is plotted at the edge distance of 7 on the x-axis, since an n-instruction scope should count the parent instruction as well. The reasons for choosing an 8-instruction scope as a point of comparison are 1) the vast majority of performance-degrading 1-cycle edges are captured within

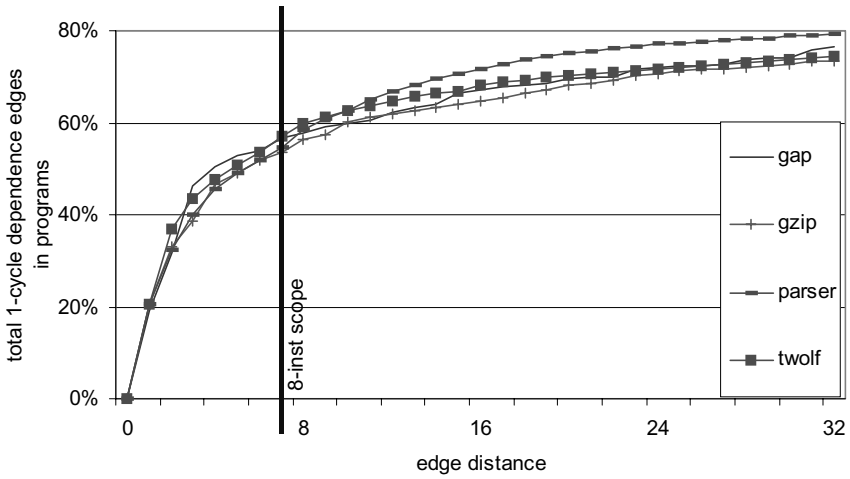


FIGURE 5-9. Cumulative distribution of 1-cycle dependence edges in sensitive benchmarks.

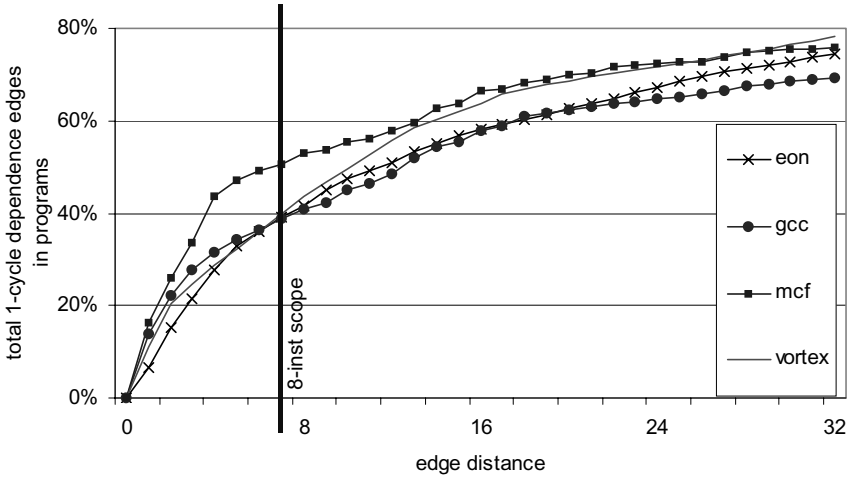


FIGURE 5-10. Cumulative distribution of 1-cycle dependence edges in insensitive benchmarks.

this scope, and 2) fetching eight instructions can be translated into at least a 2-cycle delay on the base 4-wide machine, which can potentially hide the extra latency for 2-cycle scheduling.

It is important to discuss the impact of the dependence edge distances on the residency of instructions in the issue queue. An instruction can leave the issue queue after its computation is completed. The residency between insertion and eviction of an instruction

is primarily determined by when it is inserted into the window and when its source operands become ready. If an instruction *I0* waits for its source operands for *n* cycles in the issue queue, for example, a dependent instruction *I1* inserted in the same cycle as *I0* should wait for at least *n* + *MAX (scheduling latency, execution latency of I0)* cycles before it is issued. If another instruction *I2*, which is dependent on *I1*, enters the issue queue earlier than the issue of *I1*, the residency of the parent instruction *I1* will be transferred along the dependence chain and accumulated, further increasing the residency of dependent instructions and exposing them to the scheduler's performance. Conversely, if the instruction *I2* enters the window after the issue of *I1*, *I2* can be immediately issued and residency accumulation does not occur. So, if many 1-cycle dependence edges in a program have distances greater than *MAX (scheduling latency, execution latency)*, the degree of residency accumulation becomes lower and therefore reduces the probability of long-distance dependence edges' involving in the instruction scheduling, because instructions are likely to leave the queue earlier than their dependent instructions enter the window.

Table 5-3 presents these effects measured on the base machine with a perfect branch predictor and memory. The average residency from insertion to issue on 1- and 2-cycle scheduling is presented along with IPCs for two groups of benchmarks. The benchmarks most sensitive to 2-cycle scheduling show longer residencies than the other group. Transitioning from 1-cycle to 2-cycle scheduling, the average residencies in the sensitive benchmarks are more significantly affected than insensitive benchmarks. Note that the issue queue residency is not simply a reverse metric of IPC, as the numbers in the 1-cycle scheduling column indicate. As we observed in Figure 5-10, *mcf* has a relatively longer residency compared to other benchmarks in the same group, since it has more short-dis-

Table 5-3: Comparison of issue queue residency.

Benchmarks	1-cycle scheduling		2-cycle scheduling	
	Residency from insertion to issue (cycles)	IPC	Residency from insertion to issue (cycles)	IPC
Benchmarks most sensitive to 2-cycle scheduling				
gap	10.49	2.99	16.64	2.15
gzip	13.58	2.97	22.09	2.31
parser	10.13	2.66	18.98	2.06
twolf	14.99	2.60	22.70	2.19
Benchmarks least sensitive to 2-cycle scheduling				
eon	3.29	2.84	4.56	2.73
gcc	3.01	2.84	4.73	2.79
mcf	7.95	2.77	10.39	2.74
vortex	2.87	3.06	4.38	3.00

tance 1-cycle edges.

Figure 5-11 and Figure 5-12 present more detailed data on this residency accumulation effect. Each graph shows the probability of awakening dependent instructions at a given edge distance, plotted for each benchmark group. For example, the datapoint *A* in Figure 5-11 indicates that 1-cycle dependence edges with a distance of 88 instructions awakened their dependent instructions 75% of the time. For the other 25% of the time, they were issued before dependent instructions arrived in the issue queue. The graph has a thick black line labeled as *average*, which is the average probability across all benchmarks, and given as a reference point for comparing the two graphs. The data indicate that the benchmarks insensitive to 2-cycle scheduling have much lower probabilities of awakening dependent instructions at any given edge distance, resulting in low performance impact because many 1-cycle dependence edges negatively affected by 2-cycle scheduling do not delay their dependent instructions.

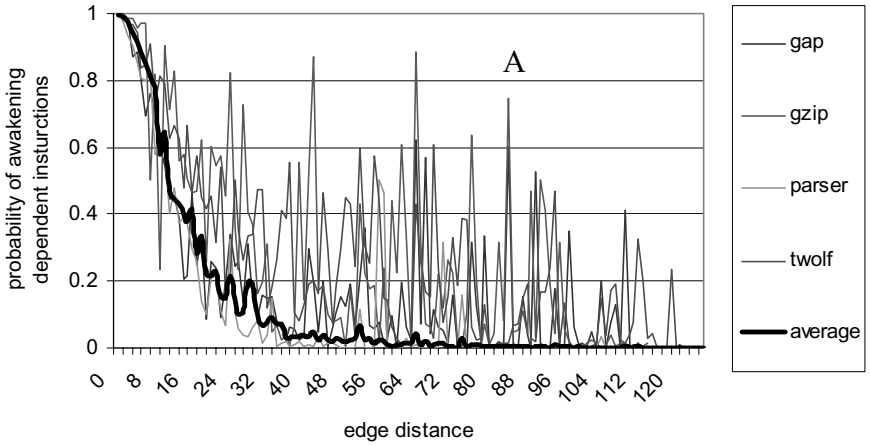


FIGURE 5-11. Probability of awakening dependent instructions in sensitive benchmarks.

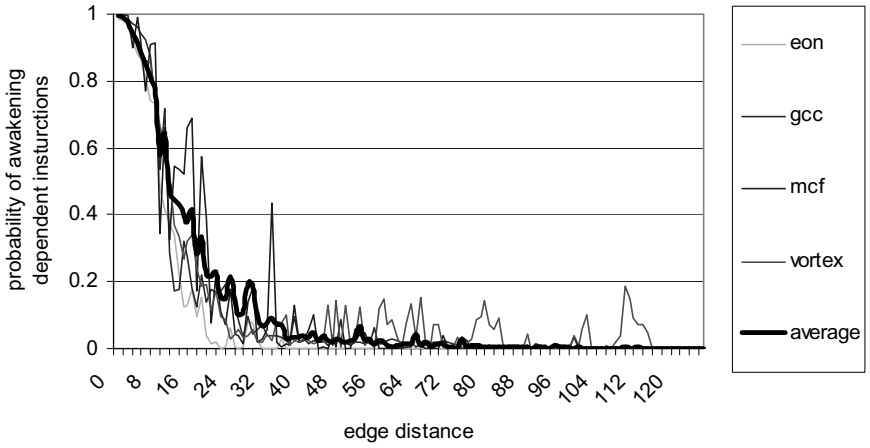


FIGURE 5-12. Probability of awakening dependent instructions in insensitive benchmarks.

5.5 Correlating Dependence Edge Distance and Performance

Based on the observations so far, I correlate the distance distributions of dependence edges and performance. Figure 5-13 compares the fraction of 2-cycle-scheduling-insensitive dependence edges and performance of 2-cycle scheduling. The purpose of this study is not to develop a statistical model for 2-cycle scheduling performance, but to show the correlation between the two different metrics. This result should not be interpreted as a

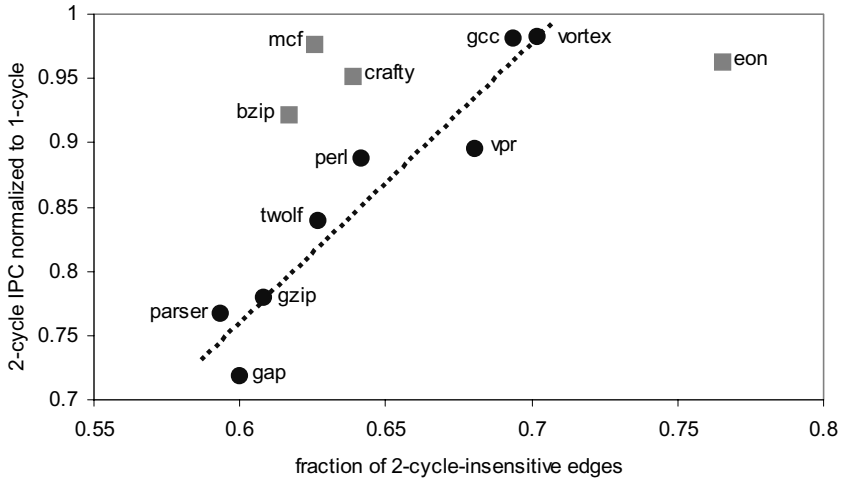


FIGURE 5-13. Correlating the fraction of 2-cycle-scheduling-insensitive dependence edges with 2-cycle scheduling performance.

prediction of 2-cycle scheduling performance because it does not take any hardware constraint (except for fetch serialization) nor execution latencies into consideration.

The y-axis labeled as *2-cycle IPC normalized to 1-cycle* is self-explanatory. The performance data are measured on the base machine with a perfect branch predictor and perfect memory. The x-axis labeled as *fraction of 2-cycle-insensitive edges* represents the ratio of the register dependence edges unaffected by 2-cycle scheduling out of the total register dependence edges created by all types of value-generating instructions. These data were calculated as follows: First, I multiply the 1-cycle dependence edge distribution shown in Figure 5-6 by the probability of awakening dependent instructions at each distance presented in Figure 5-11. The sum of the results across all dependence edge distances becomes the total number of 1-cycle dependence edges that participate in instruction scheduling. A program with more short-distance dependence edges will have a higher result since shorter edges have higher probabilities of awakening dependent instructions than longer edges. Then, I divide the sum by the total number of register dependence edges. Finally, the result of the previous computation is subtracted from 1, in order to

acquire the fraction of the edges insensitive to 2-cycle scheduling.

Since the first operation (multiplying two vectors) potentially induces a feedback of the actual performance outcome to our result, the average data across all benchmarks are used for this calculation, instead of the probabilities measured individually for each benchmark. Therefore, the generated result will not be dependent on the dynamic behaviors of each benchmark. If the actual probability data for each benchmark are used, the effect of issue queue residency accumulation is reflected to the result and it tends to show stronger correlations in many cases. Note that the instruction mix has indirectly been factored in our results as we divide the sum by the total number of register edges.

In the graph, a trend line is plotted for the benchmarks (shown as circles) that show correlation between the two metrics to some extent. As more dependence edges in the program are affected due to shorter dependence edge distances, the program tends to suffer from 2-cycle scheduling. Compared to them, some other benchmarks including *mcf* do not exhibit strong correlation, and are marked separately as squares in the graph. The overall trend indicates that a reasonable level of correlation exists between performance degradation and dependence edge distances. This implies that benchmarks with relatively short dependence edges tend to be more sensitive to 2-cycle scheduling. Conversely, benchmarks with relatively long dependence edges are tend to be less sensitive to the negative impact of 2-cycle scheduling.

5.6 Summary and Conclusions

When conventional atomic instruction scheduling logic is pipelined into two separate stages, it loses the capability for issuing instructions dependent on a single-cycle

instruction in the next consecutive clock cycle, resulting in performance degradation. This chapter examines several aspects of pipelined instruction scheduling and provides an insight into the reasons for different levels of performance sensitivity to pipelined scheduling observed in various benchmarks. Pipelined scheduling may not significantly degrade performance. This is primarily caused by hardware constraints that hide the extra delays of pipelined scheduling. In particular, the machine bandwidth that determines how fast instructions are delivered to the out-of-order window greatly affects the sensitivity. We empirically observed that the distributions of dependence edge distances in a program weaken or strengthen the tendency of performance sensitivity. A program with long dependence edges tends to be insensitive to pipelined scheduling. This is because given finite fetch bandwidth, many instructions are issued before their dependent instructions enter the instruction window and therefore the extra wakeup delays in pipelined scheduling do not directly affect performance. Conversely, a program with short dependence edges tends to lose performance significantly for the opposite reason. Therefore, a technique to focus on short-distance dependent pairs is likely to achieve most benefits.

In the following chapters, I will propose techniques to overcome the limitations of conventional instruction scheduling, and study how they complement pipelined instruction scheduling, based on the findings made in this chapter.

Chapter 6

Macro-op Scheduling

In Chapter 4, I characterized the groupability of instructions and found that there are a significant number of instructions that can be processed together as a single unit that does not require fine-grained, instruction-level controls for schedule and execution. Chapter 5 described the problems with pipelining instruction scheduling logic and analyzed its performance impact.

In this chapter, I apply coarse-grained instruction processing to instruction scheduling. This technique called *macro-op scheduling* relaxes the atomicity constraint of conventional instruction scheduling, enabling pipelined scheduling logic that issues dependent instructions consecutively. In addition, macro-op scheduling relaxes the scalability constraint, increasing the effective size of the window because multiple instructions are processed as a single unit in the scheduler and hence an issue queue entry can logically hold multiple original instructions. These combined benefits enable pipelined, 2-cycle scheduling logic to potentially outperform conventional atomic scheduling logic.

This chapter is laid out as follows: Section 6.1 presents an overview of macro-op scheduling and its benefits. Section 6.2 discusses the MOP grouping policies. Section 6.3 and Section 6.4 detail the key components in macro-op scheduling: MOP detection and formation logic. Finally, Section 6.5 to Section 6.7 discuss other performance and implementation considerations.

6.1 Relaxing the Scheduling Atomicity and Scalability via Macro-ops

As discussed in Chapter 5, scheduling atomicity is a major constraint in scaling instruction scheduling logic. This constraint is in fact imposed by the minimal execution latency of instructions; many ALU operations execute in a single clock cycle and hence scheduling of dependent instructions should be fast enough to keep up 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 [44][42] that allow dependent computations to overlap.

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

6.1.1 A scenario for macro-op scheduling

Figure 6-1 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.

Macro-op scheduling relaxes the atomicity constraint of instruction scheduling, enabling pipelined scheduling logic that issues dependent instructions consecutively. In the base case (Figure 6-1a), the wakeup and select operations must be performed within a

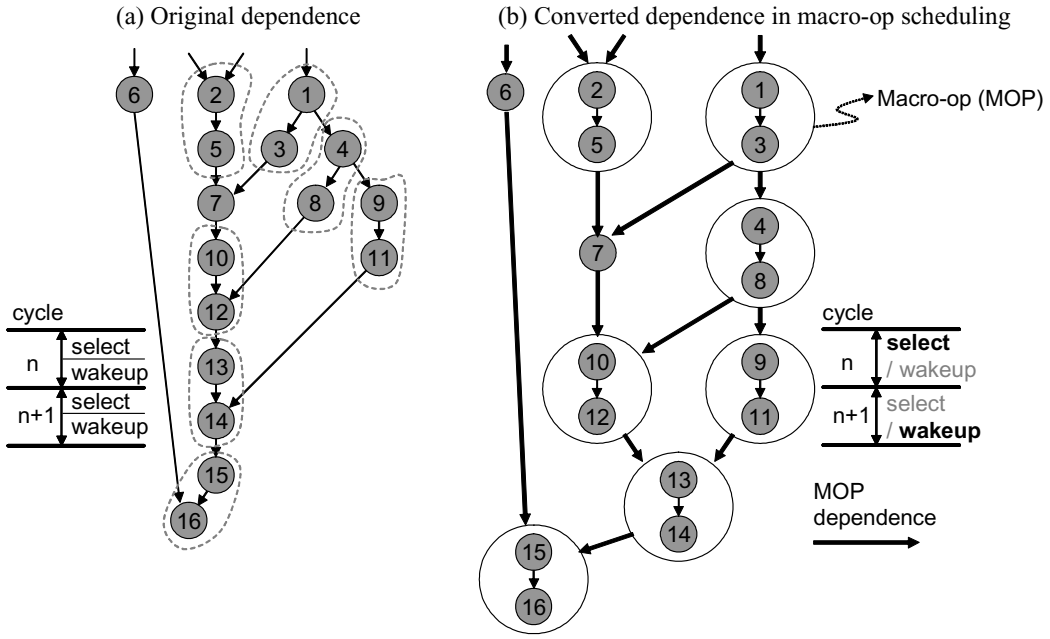


FIGURE 6-1. An example of macro-op scheduling.

single clock cycle to achieve consecutive execution of dependent instructions. In contrast, a MOP has a two-cycle latency and hence macro-op scheduling (Figure 6-1b) can perform a set of wakeup and select operations every two clock cycles. Instructions not grouped into MOPs (instructions 6 and 7 in the figure) 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 can logically hold multiple original instructions. In this 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 6-2 presents detailed scheduling timings in conventional 1-cycle, 2-cycle,

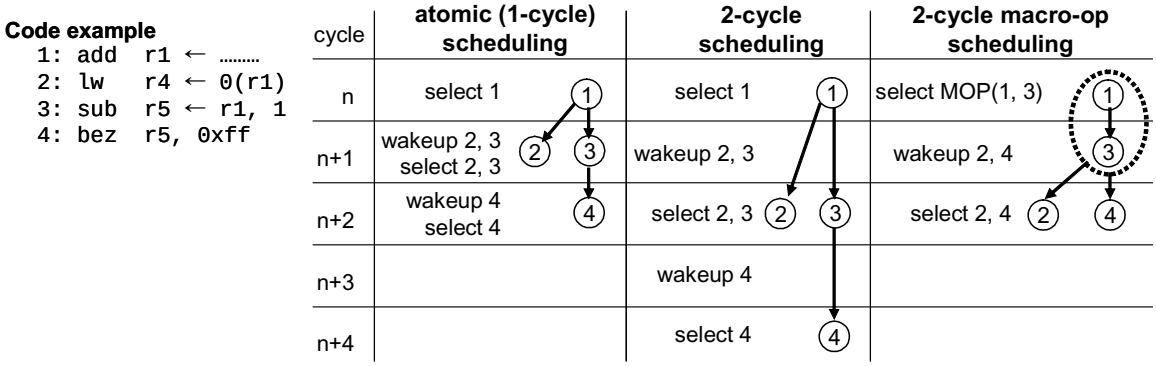


FIGURE 6-2. Wakeup and select timings.

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 1-cycle dependence edges lengthened by 2-cycle scheduling 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.

In summary, macro-op scheduling can relax the atomicity constraint of the instruction scheduling logic by processing multiple instructions as a single schedulable unit, and potentially outperforms conventional 1-cycle scheduling due to its improved scalability.

6.1.2 An overview of a microarchitecture with macro-op scheduling

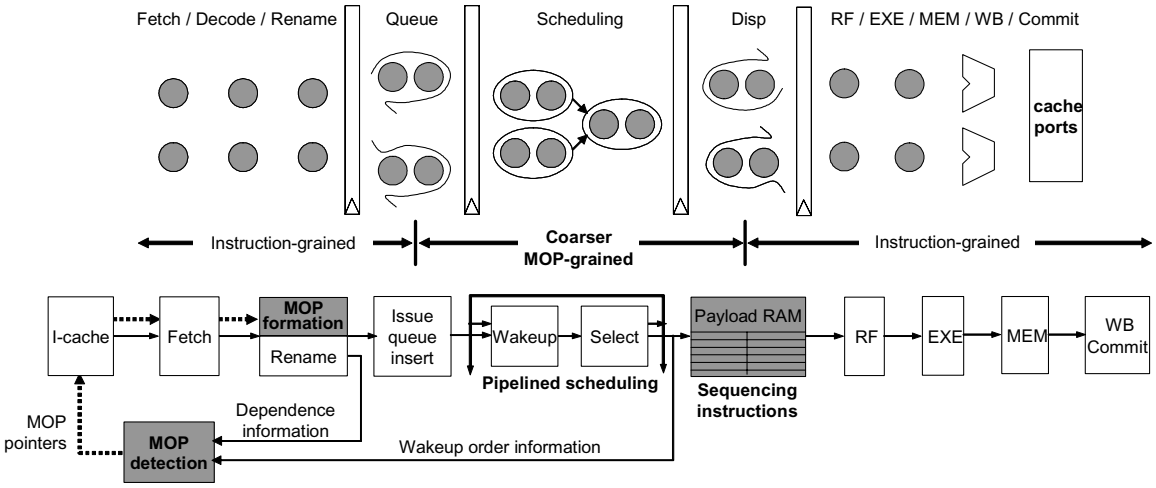


FIGURE 6-3. An overview of macro-op scheduling.

Figure 6-3 illustrates an overview 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*. MOP pointers are stored in the instruction cache, and specify 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 a single issue entry.

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 depen-

dences 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 and the MOP tail does not incur a tag broadcast. These dependence conversions replace data dependence edges with MOP dependence edges, abstracting the data dependence graph without violating the true register dependences.

After the 2-cycle instruction scheduler issues multi-cycle MOPs when all source MOP dependences become ready, they access the payload RAM [9], which sequences the original instructions in the instruction-grained execution pipelines. At the same time, the original register identifiers for source operands are obtained from this structure. In the execution stage, the two instructions will be executed separately within two clock cycles. Since macro-op scheduling simply alters the way instructions are scheduled, it ensures correctness of execution and the register values are accessed based on the original data dependences. After execution completes, the reorder buffer commits ungrouped original instructions separately in program order. Therefore, macro-op scheduling still preserves correct architectural state even when branch misprediction recovery or exception handling is required.

6.2 Policies to Group Instructions

There are two major issues in determining which instructions are grouped and processed together in macro-op scheduling: *performance* and *complexity*. For performance, macro-op scheduling requires a judicious grouping policy to be beneficial, since the grouping process potentially alters the way instructions are scheduled and improper grouping may degrade performance by serializing instruction execution. The complexity

of macro-op detection and formation logic may be significantly affected by the MOP scope, i.e. the number of instructions that should be searched and examined to find groupable instructions, although a larger scope enables more groupable instructions and would be more beneficial.

6.2.1 Candidate instruction types

Since the primary goal of macro-op scheduling is to relax the atomicity of the scheduling loop and to pipeline instruction scheduling logic, macro-op scheduling targets single-cycle operations: single-cycle ALU, store address generation, and control (e.g. branch) 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. However, grouping these types of instructions can be also beneficial by reducing the number of schedulable units in the scheduler and reducing the issue queue pressure. Therefore, we will also evaluate the potentials for grouping those instructions later in Chapter 9, in conjunction with the discussion of macro-op execution.

6.2.2 MOP scope

Macro-op scheduling groups a chain of dependent instructions and converts them into a multi-cycle latency MOP. As we characterized dependence edge distance between two instructions in Chapter 4, many dependent instructions are placed near each other and most cases are captured with an 8-instruction scope. There is also an important performance issue in determining the MOP scope. As we measured in Section 5.4.1, the vast majority of 1-cycle dependence edges that awaken their dependent instructions are captured within an 8-instruction scope on our base 4-wide machine. This implies that a wider

scope does not necessarily benefit macro-op scheduling since longer dependence edges are less likely to degrade performance, although capturing more instructions further reduces issue queue contention and may improve performance. Therefore, the focus of our work is on an 8-instruction scope. Later in Chapter 7, I will study the sensitivity of macro-op scheduling to MOP scope and evaluate the potentials for other configurations.

6.2.3 MOP sizes

The policy for macro-op scheduling is to group two directly dependent instructions, or two independent instructions with identical source operands into a MOP (*2x MOP configuration*). Although bigger MOP sizes may enable the scheduling loop to span over more clock cycles and further increase the effective machine bandwidth, we characterized the groupability of instructions in Section 4.6 and find that not many MOPs captures more than two instructions in general. A bigger problem is that irregular MOP sizes incur low resource utilization of the tag comparators (CAM-style wakeup logic) or in the payload RAM structure, which should be built to satisfy the worst-case requirement of a MOP (more than three tag comparators or more than two instructions per entry in the issue queue or the payload RAM). Therefore, the focus of our work is 2x MOPs. Note that the low groupability in terms of MOP sizes is not a fundamental limitation of macro-op scheduling. If a more advanced MOP detection and formation mechanism (e.g. relying on a software-based dynamic binary translator) can generate more MOPs big enough to fully utilize the given hardware resources, larger MOPs would be a reasonable design choice. Later, in Chapter 7, I will study the sensitivity of macro-op scheduling to MOP size and evaluate the potentials for other configurations.

6.2.4 MOP dependence tracking

Section 4.3 described two approaches for tracking MOP dependences: MOP offset tracking and MOP latency tracking. The base macro-op scheduling groups only two single-cycle instructions and the execution latency of a MOP matches the scheduling latency of the pipelined 2-cycle scheduling logic. This attribute forces the instruction scheduler to use MOP latency tracking, since the scheduler cannot track the offset of the instructions finer than the scheduling latency (i.e. two clock cycles), and therefore the MOP offset tracking is not feasible for the current scheduling constraints.

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

6.3.1 Cycle conditions through register dependences

Macro-op scheduling abstracts true data dependences and creates false dependence edges when instructions are grouped (explained in Section 4.4 and Section 6.1.1). These false dependences may prevent instructions from being issued if they induce cycles in data dependence chains. Figure 6-4 illustrates possible deadlock conditions through register dependences created by improper MOP grouping. In Figure 6-4a, a MOP that contains instructions 1 and 3 has both incoming and outgoing edges to instruction 2. Figure 6-4b

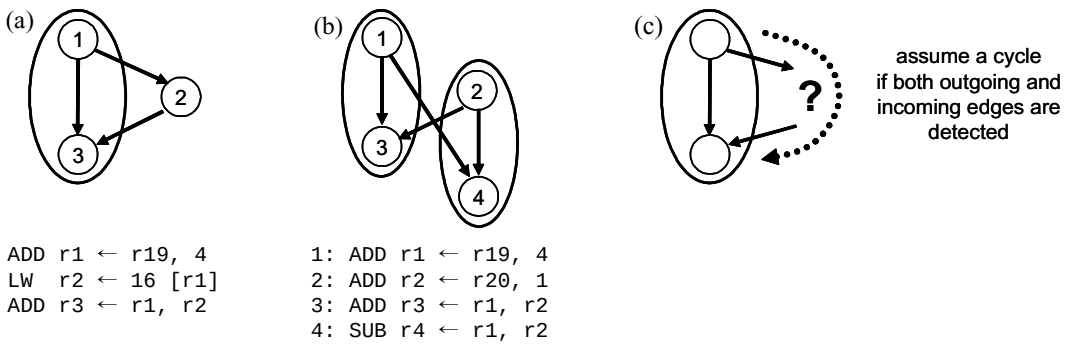


FIGURE 6-4. Cycle conditions and a detection heuristic.

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. To avoid these conditions, MOP detection logic should filter out improper MOP groupings that create cycles in the data dependence chains. However, precisely detecting cycle conditions may significantly increase the complexity of the detection process because it requires the detection logic to track multiple levels of dependences along with register dependence chains.

Therefore, MOP detection logic uses a simple heuristic to detect possible cycle conditions conservatively, as shown in Figure 6-4c; 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, the experimental results in Section 7.8.4 will present that false detections are negligible and this heuristic achieves almost all of possible MOP formation opportunities compared to the precise cycle detection.

6.3.2 Cycle conditions through memory dependences

Unlike register dependences that directly affect scheduling decisions and force instructions to execute in correct register dependence order, memory dependences may not force memory operations (i.e. loads and stores) to be scheduled in correct dependence order. Instead, the correctness of scheduling is verified later, after they are speculatively scheduled, since memory dependences cannot be detected at schedule time before effective addresses are calculated. For example, some processor implementations such as the IBM POWER4 [71] optimistically schedule loads and stores based only on register dependences when no memory aliasing is expected, and later recover from the dependence misspeculation if a dependent load has generated an incorrect result due to an aliased earlier store. In this memory disambiguation policy, MOP cycle conditions are not induced through memory dependences since memory dependences are not reflected in scheduling decisions and hence incorrectly grouped instructions may be executed without leading to a deadlock, as long as aliasing is properly handled after execution.

However, special considerations should be made if the memory disambiguation policy used in the underlying microarchitecture affects scheduling decisions and prevents certain instructions from being executed. In the memory disambiguation policy assumed in our base machine model (described in Section 3.1.4), which is similar to the one used in the Pentium 4 [42], a load can be issued only after its source operand dependence is satisfied and all prior store instructions in program order have been issued or executed. Due to this attribute, a deadlock may occur even though stores and loads are not actually aliased because instruction scheduling logic conservatively creates dependences from an unresolved prior store to all load instructions later in program order. Figure 6-5 presents three

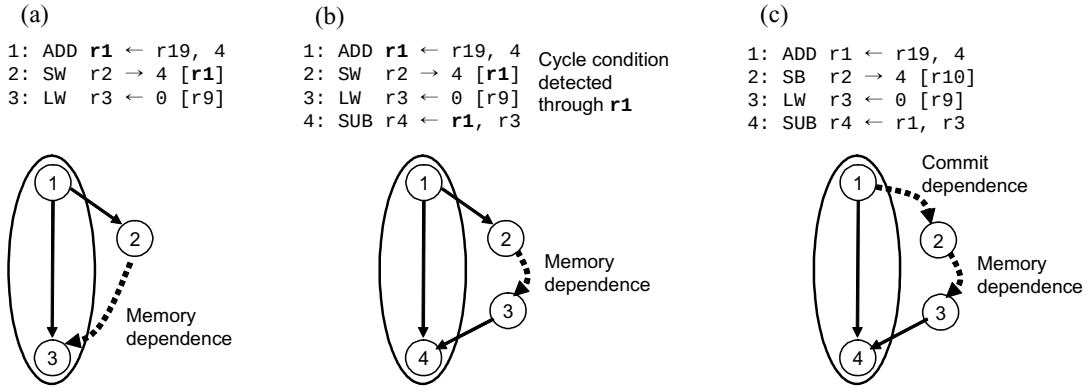


FIGURE 6-5. Scenarios for cycle conditions created through memory dependences.

different scenarios for cycle conditions created through memory dependences by improper MOP groupings in our memory disambiguation model.

Figure 6-5a shows a cycle through memory dependence induced among three instructions. This scenario assumes that a load (instruction 3) can be grouped as a MOP tail, although a MOP grouping policy that disallows loads does not experience this problem. Regardless of whether SW (instruction 2) and LW (instruction 3) are aliased, the memory disambiguation policy forces the instruction scheduling logic not to issue LW until after SW (store AGEN operation) is issued. Since SW is also dependent on ADD, which is grouped with LW, a deadlock condition similar to Figure 6-4a is encountered. This condition is not detected by the cycle detection heuristic discussed in Section 6.3.1 since LW (instruction 3) has only one source register and hence an incoming dependence edge to the MOP tail is not observed at detection time. To avoid this case, a load should not be grouped as a MOP tail only when there is an intervening store between the MOP head and tail in program order.

Problems still exist even though the MOP grouping policy does not consider loads as MOP candidates. Figure 6-5b illustrates such a scenario in which cycle conditions may

be induced without grouping loads. The memory dependence between SW and LW completes a closed dependence cycle in this case. Fortunately, the cycle detection heuristic discussed in Section 6.3.1 captures this cycle condition due to the incoming and outgoing register dependences of the MOP. Precise cycle detection would fail to detect this case since memory dependences are not easily determined statically.

A more complicated scenario is illustrated in Figure 6-5c, where the underlying hardware supports fully overlapped store-to-load forwarding only. This configuration is similar to the one used in the Pentium 4 [42], where a partially overlapped aliasing (i.e. some bits for a load are available from a prior store, and others are available from memory) is handled only through memory; the load can acquire the value by accessing the cache after the prior store reaches the head of ROB and commits its data to the memory system. Consider a case in which a partial aliasing is detected between SB and LW. This restriction in the store-to-load forwarding creates two dependences edges among instructions in this example: a *commit dependence* that prevents SW from writing the store value to memory until after ADD is committed, and a memory dependence that prevents LW from being executed until after SW writes its store value. Combined with the register dependence between LW and the MOP of ADD and SUB, this MOP grouping induces a cycle among instructions in the example, which is not captured by the cycle detection heuristic discussed in Section 6.3.1 because no register dependence exists between ADD and SUB.

In order to correctly support the case in which the partial forwarding is not supported, the MOP grouping policy can conservatively avoid grouping instructions across intervening store-and-load pairs, although it may unnecessarily lose some grouping opportunities. If the underlying hardware supports partial store-to-load forwarding by merging

Table 6-1: Cycle avoidance heuristics.

Memory dependence	Partial store-to-load forwarding is supported	Partial store-to-load forwarding is not supported
Grouping loads is allowed	A load cannot be grouped as a MOP tail across intervening stores.	A load cannot be grouped as a MOP tail across intervening stores. A MOP cannot be created across store-load pairs.
Grouping loads is not allowed	None	A MOP cannot be created across store-load pairs.
Register dependence	A MOP cannot be created 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.	

uncommitted store data in the window and memory values, the commit dependence is removed and hence this scenario does not experience a deadlock.

Table 6-1 summarizes the cycle avoidance heuristics discussed in Section 6.3.1 and Section 6.3.2 for different MOP grouping policies and store-to-load forwarding configurations. We will examine the impact of these heuristics on groupability and performance later in Section 7.8.4 and Section 7.8.5.

6.3.3 MOP detection process

Figure 6-6 illustrates 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 pairs using dependence matrices through steps n to n+2. The dependence-matrix-based detection process described here is to illustrate key operations to perform for MOP detection, and the detailed implementation issues will be discussed in the following section. To simplify the discussion, we assume that 1) loads are not MOP candidates, 2) the underlying microarchitecture supports partial store-to-load forwarding, 3) the instruction scheduler has matrix-based wakeup logic and therefore

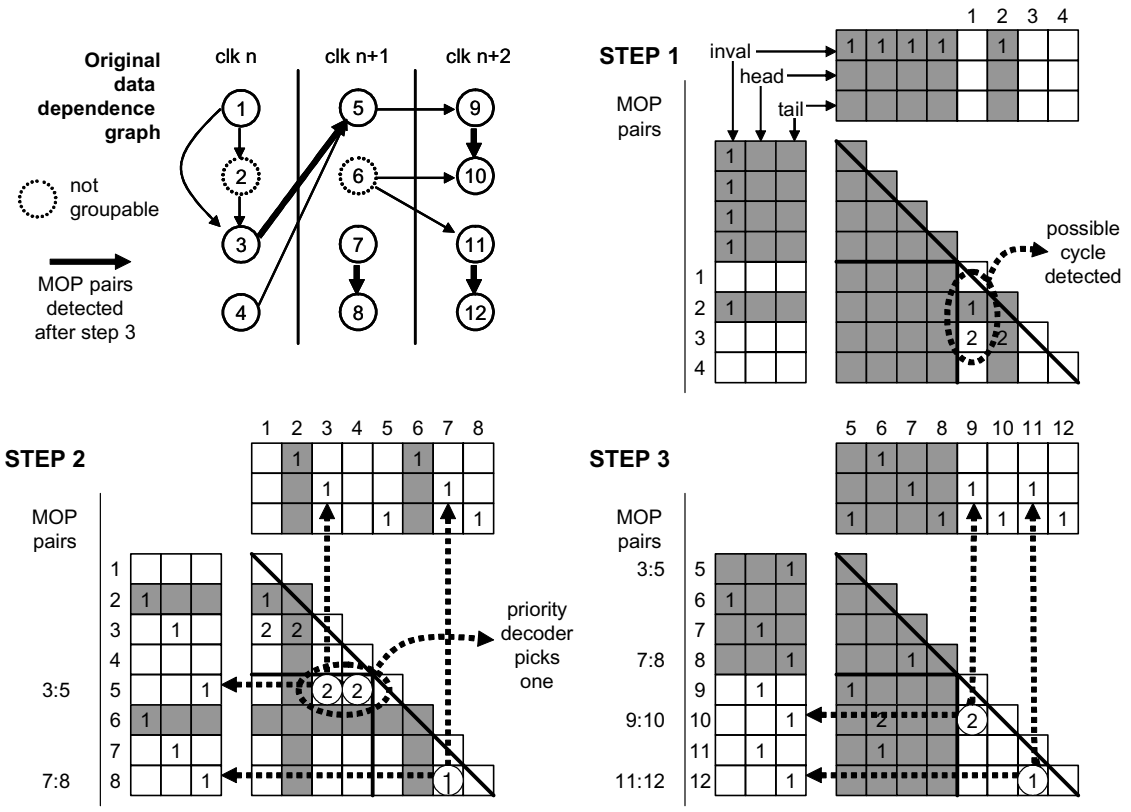


FIGURE 6-6. MOP detection process.

counting the number of source operands is unnecessary and 4) the MOP scope is eight instructions. Based on these assumptions, this MOP grouping policy needs the cycle detection heuristic discussed in Section 6.3.1 for register dependences only.

In the figure, a triangular 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 6.3.1.

In step n, instructions 1 to 4 fill the bottom right portion of the triangular 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 to 8 fill the triangular matrix from the bottom right portion, and instructions 1 to 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 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 pairs are generated. Selected instructions mark corresponding *head* or *tail* fields so that they will not be examined again. Similarly,

instructions 9 to 12 are examined in step n+2 and four MOP pairs are finally generated.

To avoid cases in which the number of source operands exceeds the number of tag comparators in the wakeup array (for CAM-style wakeup logic), MOP detection logic may need to *statically* (without considering readiness of operands) count the source operands of instruction pairs and filter out MOPs with too many sources. Dynamically counting source operands right before inserting instructions in the wakeup logic would increase opportunities for grouping more instructions, since some operands may not need tag comparators due to retired parent instructions. However, our experimental result will show that they do not occur frequently and the static approach captures almost all opportunities.

6.3.4 Implementation issues

The MOP detection process can be implemented in either hardware or software. There are tradeoffs between two approaches; a software-based approach (i.e. dynamic binary translation) may remove considerable complexity from the hardware and enable more sophisticated detection algorithms with a wider MOP scope. However, it may incur a high performance overhead if the software-based MOP detection process is frequently invoked; a hardware-based approach can minimize or completely eliminate extra overhead required for the detection process, but realizing sophisticated algorithms in hardware would be challenging, considering the complexity of operations to be performed.

As a study of software-based MOP detection, which is directly related to macro-op scheduling presented here, Hu and Smith [46] proposed a dynamic binary translator that cracks x86 instructions into RISC micro-operations, uses heuristics to group or *fuse* pairs of dependent micro-ops, and then converts them into an internal ISA running on a co-designed virtual machine. They demonstrated and concluded that using dynamic binary

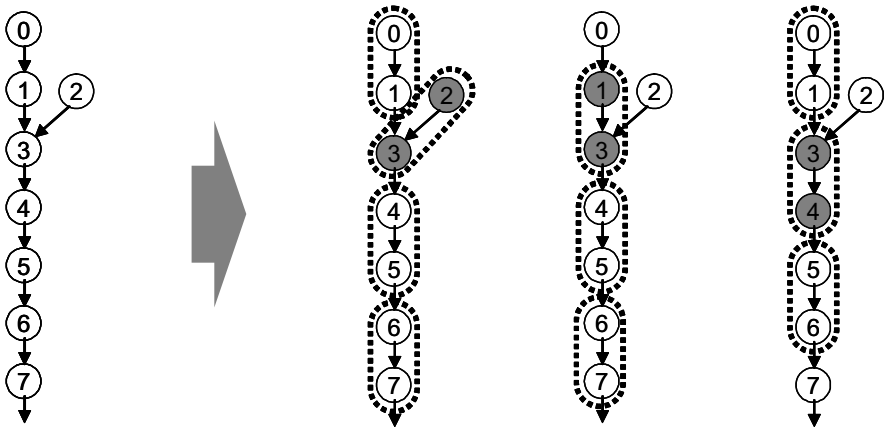


FIGURE 6-7. Dependences in MOP detection processes.

translation is a reasonable and feasible approach to instruction fusing, especially for implementing the x86 instruction set. Since many important aspects of software-based approaches are covered in this work, this section will primarily discuss the issues in hardware-based MOP detection.

In hardware, the MOP detection process can be implemented in a way similar to prior proposals for instruction preprocessing at trace cache line construction time [50][38][16]. Since the information on detected MOP pairs can be saved as predecode bits (e.g. MOP pointers in this study) in IL1 or trace cache, and reused repeatedly until evicted, a long detection latency may not significantly affect the benefits of grouping instructions.

An important issue in hardware-based MOP detection is *pipelinability*. Given the MOP detection scope of *S* instructions, each MOP head candidate needs *S* associative search operations for a dependent groupable MOP tail in *forward scan*. In *backward scan*, each MOP tail needs to examine up to two antecedent instructions to find a groupable MOP head. The forward and backward scan algorithms will be evaluated in Section 7.8.2. In either case, operations required for finding one matching pair may not be prohibitively complex. However, in order to avoid piling up of instructions, the MOP detection logic

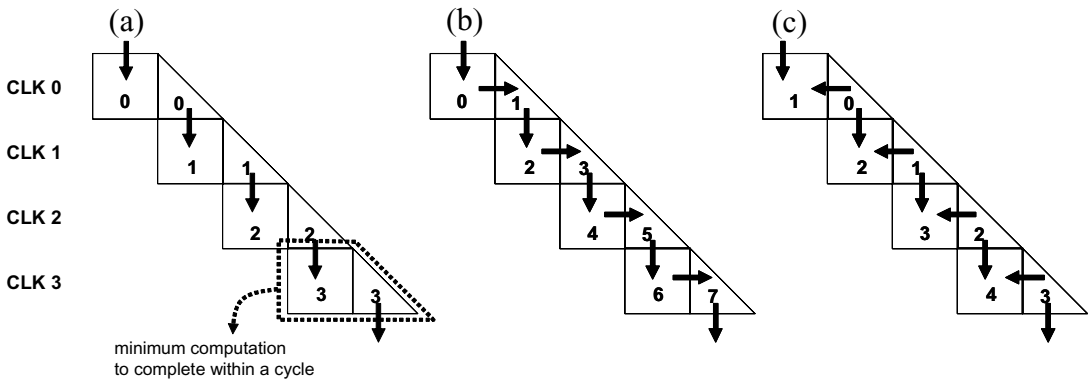


FIGURE 6-8. Pipelinability of MOP detection.

should be able to process multiple instructions at the same rate, i.e. four instructions per cycle, as they are sent from the rename stage. Due to the nature of MOP detection, detecting one MOP pair is dependent on another, which makes it challenging to detect multiple MOP pairs in parallel and to achieve a pipelined implementation. Figure 6-7 presents an example of dependent MOP detection processes, in which instruction 3 can be grouped with multiple possible candidates. Depending on the grouping decision made for instruction 3, grouping decisions for other instructions are also affected, generating different combinations of MOP pairs.

Figure 6-8 illustrates this problem with pipelining MOP detection process. In this figure, the dependence matrices in Figure 6-6 are expanded over time. The numbers in matrices represent at which cycle the matrices are processed. The arrows represent dependences among matrices.

In an ideal case, both rectangular (which examines inter-rename-group dependences) and triangular (which examines intra-rename-group dependences) matrices should be processes simultaneously every clock cycle in order to achieve a pipelined implementation, as shown in Figure 6-8a. This implies that four dependent computations to examine four instructions should be performed within a cycle. Suppose only one matrix (either

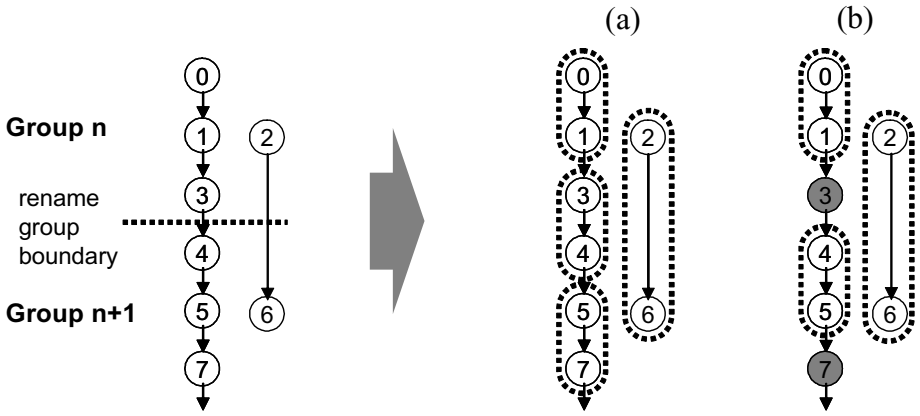


FIGURE 6-9. Impact of pipelining MOP detection on MOP coverage.

rectangular or triangular) can be processed at each cycle because processing both matrices does not fit within a cycle. Since processing each matrix is dependent on one another and serialized, this incurs piling up of instructions and halves the effective throughput, as shown in Figure 6-8b.

One possible solution to pipelining the MOP detection process is to start the process at multiple points and later resolve dependences in a reverse order. This is possible since detecting MOP pairs needs not be performed in program order. Figure 6-8c illustrates this, where triangular matrices are first examined each cycle and rectangular matrices are examined later, enabling a pipelined implementation. However, the benefit of this approach comes with a penalty; it potentially reduces the efficiency of the detection process. Consider the example shown in Figure 6-9. Figure 6-9a illustrates MOPs detected using a non-pipelined process, and Figure 6-9b shows the result with the MOP detection process pipelined in a way discussed with Figure 6-8c. The *pipeline interval* for this MOP detection is four instructions, which is the number of instructions between two starting points of MOP detection processes. Instructions 0 to 3 are examined separately from instructions 4 to 7. During this process, MOP (0, 1) and MOP (4, 5) are detected in each

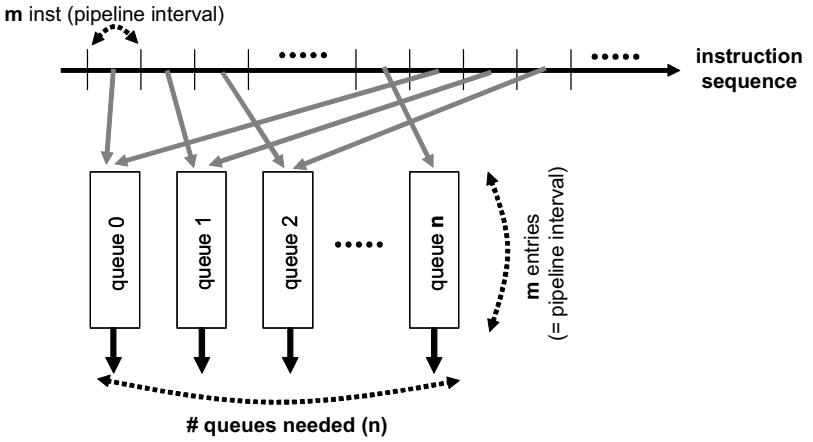


FIGURE 6-10. Pipelined MOP detection logic with multiple detection queues.

group. However, instructions 3 and 7 cannot be grouped because of no matching pair; otherwise all eight instructions would have been grouped into four MOPs, as shown in Figure 6-9a. This negative effect can be reduced by increasing the pipeline interval so that longer sequences of instructions are examined continuously.

Even though fewer instructions than the machine bandwidth can be processed at a time (e.g. only one instruction per cycle), the MOP detection mechanism can be configured to achieve the necessary throughput by extending this approach. Figure 6-10 illustrates this MOP detection mechanism with multiple detection queues that detect MOP pairs at multiple points in parallel. As instructions are sent from the rename stage, MOP detection logic first assigns them to detection queue 0 until the queue becomes full, when incoming instructions are directed to the next queue. The size of each queue is determined by the pipeline interval. MOP detection processes are performed in multiple queues in isolation, generating MOP pairs in each queue. The tail of one queue and the head of the next queue are logically connected so that MOP pairs can be generated across different queues after the both queues finish examining their own instructions. By the time the last detec-

Table 6-2: Complexity estimation of pipelined MOP detection logic.

Pipeline interval (insts)	1 instruction / queue per cycle, 8-inst scope, 4-wide machine			
	# entries in each queue (m)	Minimum detection latency	# detection queues needed (n)	Total detection queue entries
4	4	14	14	56
8	8	14	7	56
16	16	20	5	80
32	32	36	4.5	144
64	64	68	4.25	272

tion queue becomes full, the first queue completes MOP detection process so that it can process new incoming instructions.

Table 6-2[1] summarizes the queue size and the detection latency of the pipelined MOP detection logic that avoids piling up of instructions on the base 4-wide machine. The MOP scope is eight instructions in all cases. Each detection queue examines one instruction per cycle, generating at most one MOP pair. Note that the detection latency and the total queue entries for the 4-instruction pipeline interval case are not lower than the 8-instruction interval case. This is because the current MOP scope (eight instructions) is greater than the pipeline interval (four instructions) and it takes more time to resolve inter-group dependences. Also note that the number of detection queues in the table (fourth column) may not be an integer number. The actual implementation may require more detection queues, e.g. five queues instead of 4.5 queues with a 32-instruction interval.

For instance, if the pipeline interval is 16 instructions, the MOP detection logic requires five detection queues with 16 entries each, containing maximum 80 in-flight instructions. The minimum MOP detection latency (for the last instruction in each detec-

1. The minimum detection latency is primarily determined by (pipeline interval) / (throughput of each detection queue) + (latency for examining inter-queue dependences). The number of total queue entries is calculated by (detection latency) X (fetch bandwidth).

tion queue) becomes at least 20 cycles.

In Section 7.8, I will measure the impact of MOP detection latency and pipeline interval on the MOP coverage, and show that they do not significantly degrade the efficiency of the MOP detection process.

6.3.5 MOP pointers

To avoid placing MOP detection logic in the processor's critical path and to tolerate its long detection latency, it is desirable to cache the generated information on MOP pairs. One issue is how to encode the information on MOP pairs. One approach is to store a pointer (called *MOP pointer* in our discussion) in each instruction to its groupable instruction pair, maintaining the original program order among instructions. Another approach is to place groupable instructions pairs together in a trace cache or decoded instruction cache. Since non-adjacent instructions can be grouped, this requires reordering instructions and may create difficulties in instruction fetch and rename for maintaining correct architectural state. However, some prior proposals [50][38][16] have studied reordering instructions within a trace cache line to realize their benefits and hence this approach should not be restricted by those difficulties. Specifically, a software-based approach proposed by Hu and Smith [46] reorders instructions with the virtual machine monitor support for reconstructing precise machine state. A software-based approach is beyond the scope of this thesis, which focuses on the pointer-based approach.

A MOP pointer that specifies a groupable instruction pair can be either backward or forward:

- Backward MOP pointer: A MOP tail points to the groupable MOP head. The pointer is stored along with the tail instruction. A downside of this approach is that

the MOP formation logic cannot determine MOP pairs until the MOP tail is fetched and therefore grouping instructions across multiple fetch groups may not be feasible. Moreover, tracking control flow within a groupable pair is complicated due to possible multiple branch / jump sources.

- Forward MOP pointer: A MOP head points to the groupable MOP tail. The pointer is stored along with the head instruction. Since the pointer cannot rely on register specifiers that point to antecedent instructions, it may need more bits to specify groupable instructions explicitly. However, MOP formation logic can headstart the grouping process by looking at MOP heads only. Also, tracking control flow between groupable instructions becomes easier. Therefore, I advocate forward MOP pointers because of these advantages.

A MOP pointer is composed of two parts: control and offset bits. The control bits represent possible control flow change between MOP head to tail. We use one control bit that 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 is taken, MOP detection logic does not generate a pointer and foregoes the opportunity. The offset bits are simply the instruction count from the MOP head to the tail. The number of bits in the offset field depends on the MOP scope, e.g. a 3-bit offset field can cover up to eight instructions. An example of MOP pointers with a MOP scope of eight instructions is shown in Figure 6-11.

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 (`beq` in the example), and the branch is later predicted to be not taken,

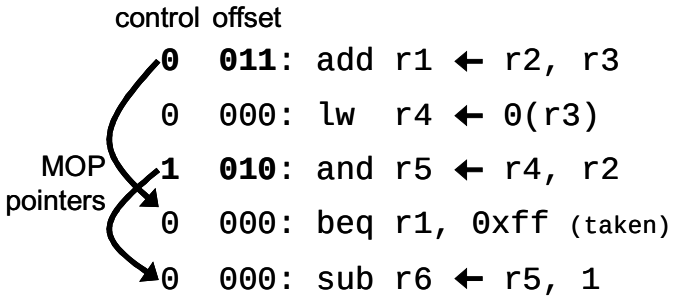


FIGURE 6-11. Macro-op pointers.

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. In this example, `and` can be grouped only with `sub` across the taken branch but not with others, until it is re-created across a not-taken branch.

6.4 MOP Formation

MOP formation is responsible for checking control flow, locating MOP pairs using 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.

6.4.1 Locating MOP pairs

Locating MOP pairs is the reverse process to MOP pointer generation; 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.

6.4.2 MOP dependence translation

Macro-op scheduling abstracts the original data dependence chains and therefore

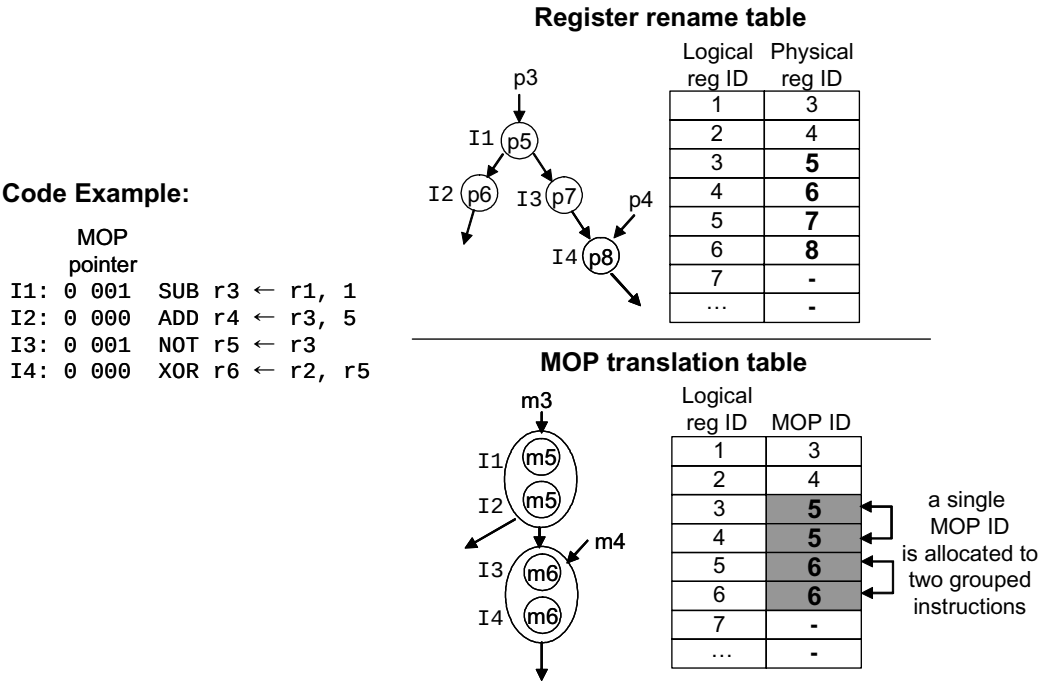


FIGURE 6-12. Dependence translation in MOP formation.

requires dependence conversion from register IDs to MOP IDs so that scheduling logic keeps track of dependences in a separate name space. I do not believe that translating to this name space will incur much delay. As discussed in Section 3.1.2, 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 6-12 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 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

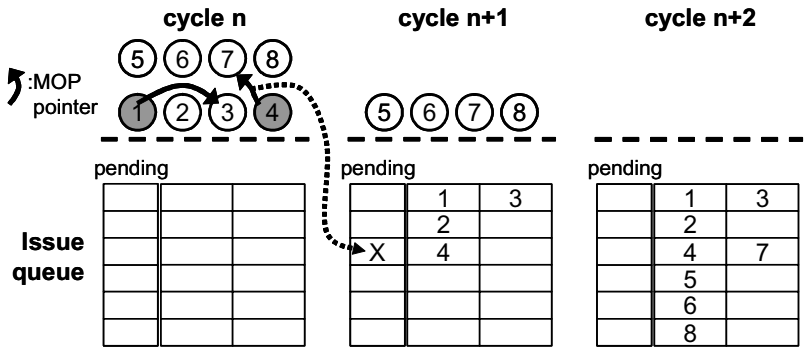


FIGURE 6-13. Inserting instructions into issue queue.

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.

6.4.3 Queue stage and 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 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 6-13 illustrates this non-atomic insertion policy of grouped instructions

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 to 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, indicating 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, this completes the MOP of the two instructions and the pending bit is cleared. The MOP will be issued when all source operands become ready.

6.5 Instruction Scheduling Logic for Macro-op Scheduling

Although macro-op scheduling affects the scheduling of instructions, it does not require significant changes in the instruction scheduling logic itself, except that wakeup and select processes can be pipelined. This section discusses how MOPs are processed in wakeup and select logic.

6.5.1 Wakeup logic

Figure 6-14 shows issue queue entries occupied by the instructions `I1` through `I4` grouped in two MOPs (based on the same code example as in Figure 6-12). Each entry has source identifiers that are the union of all source operands of two original instructions, except for the operand that links the two instructions. The number of source operands is not limited for wired-OR wakeup logic, but is limited to two for the wakeup array with two source comparators. Since MOP detection logic has filtered out 3-source cases for 2-source CAM-style wakeup logic (discussed in Section 6.3.3), no additional handling is necessary for source operands in the wakeup logic.

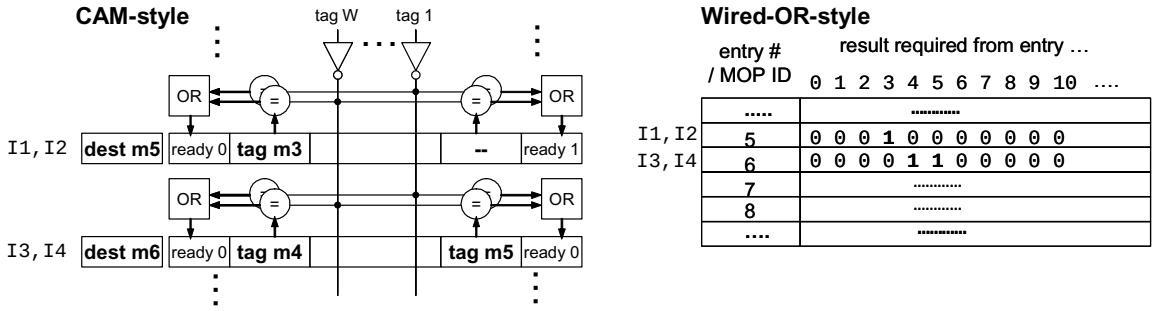


FIGURE 6-14. Grouped instructions in CAM and wired-OR-style wakeup logic.

The basic operation of wakeup logic is the same as the base case: an instruction or a MOP is issued when all source operands become ready. Since MOPs have multi-cycle latencies (2-cycle latency in our current configuration), waking up their dependent instructions should be handled accordingly in the same way as normal multi-cycle operations are handled. Handling instructions with different execution latencies in a single scheduler may incur some difficulties in scheduler design. However, some processor implementations such as the POWER4 [71] schedule all types of integer instructions with different execution latencies in a unified scheduler and hence this should not be a fundamental limitation. A more important observation to advocate this unified scheduler approach is that handling 1-cycle instructions is no different than handling 2-cycle instructions in pipelined 2-cycle scheduling logic since they will be treated equally as 2-cycle instructions, from the wakeup logic's perspective. Therefore, scheduling MOPs may not necessarily increase the complexity of wakeup logic due to extra timers for tracking multi-cycle latencies.

6.5.2 Select logic

The basic operation of select logic is same as the base case; 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 instruc-

A MOP is equivalent to a non-pipelined operation from the select logic's perspective; multiple operations in a MOP are sent to the execution pipeline through a selected issue slot and occupy their execution resources (i.e. ALUs and cache ports). To support this functionality, select logic blocks the issue slot and does not send other instructions through the same slot for the execution latency of the MOP.

6.6 Pipeline Considerations

6.6.1 Dispatch stage and sequencing instructions

An issued MOP is converted back to the original instructions that are executed sequentially. This functionality is achieved by a multiple-entry payload RAM that stores multiple logical instructions grouped in a MOP in a single line. Note that macro-op scheduling requires the same number of read and write ports to the payload RAM as the base case. When an issued MOP accesses the payload RAM, the opcodes and register specifiers of the original instructions are acquired, and each instruction is sent down to the appropriate execution pipeline in consecutive cycles. If the base machine does not have a payload RAM structure, sequencing instructions can still be performed by the scheduling logic, similar to the AMD K7 or the Intel Pentium M [21][39].

As we discussed in Section 6.5, a MOP is equivalent to a non-pipelined, multi-cycle operation from the scheduler's perspective. In order to give time window for the payload RAM to sequence instructions, the select logic does not select and issue other instructions through the same issue slot in which a MOP is being sequenced.

6.6.2 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, this does not affect correctness of the architectural state since the ROB commits ungrouped original instructions separately in program order.

The scheduling replay mechanism invalidates and reschedules instructions dependent on mis-scheduled loads. In this case, all instructions grouped in a MOP are replayed as a single unit since the scheduler keeps track of dependences in the MOP name space. Although this policy may replay some instructions unnecessarily, our experiments determined that the performance impact is negligible.

6.7 Performance Considerations

6.7.1 Grouping independent instructions

So far, I have discussed MOP grouping of dependent instructions (*dependent MOP*). As discussed in Chapter 4, grouping dependent instructions is more beneficial than independent instructions since it relaxes both atomicity and scalability constraints simultaneously, without negatively affecting performance by serializing parallel and independent instructions. However, grouping independent instructions (*independent MOP*) is also beneficial as long as ILP is not sacrificed, since fewer schedulable units, created from both

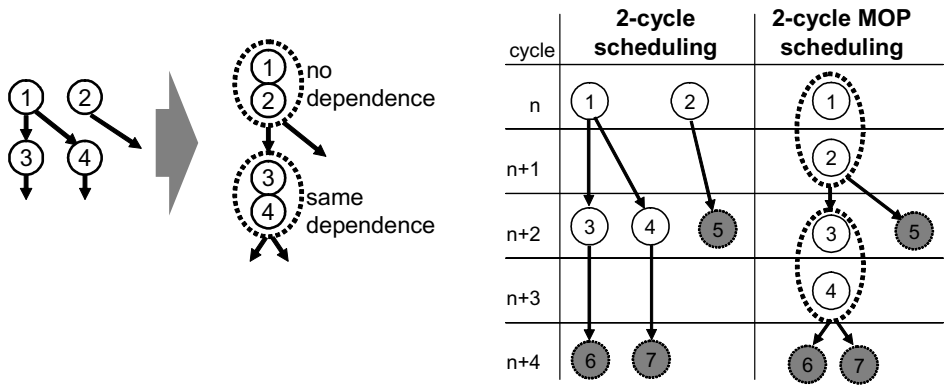


FIGURE 6-15. Grouping independent instructions and their execution timings.

dependent and independent MOPs, reduce the issue queue pressure.

The first consideration to make is which independent instructions are grouped. In principle, any instructions that are issued in parallel could be grouped and processed as a single unit without reducing the ILP. However, detecting such cases can be a complex task since dynamic events may create distortion in the sets of parallel instructions. Moreover, finding transitively parallelizable instructions requires multiple levels of dependence tracking. To avoid these difficulties, macro-op scheduling is restricted to grouping instructions with either no source operands or identical source operands, which are guaranteed to execute in parallel as long as there is no structural hazard. Potentially, these conditions can be relaxed to, e.g. grouping instructions with the same last-arriving operand [27][56] that triggers instruction issue, but such a relaxation that relies on dynamic information is not considered in this study.

The second consideration is the performance impact of sequencing instructions. Serializing independent instructions may affect performance negatively in some timing-critical cases (e.g. mispredicted branch resolution) because they can be issued in parallel in the base case. However, due to the attribute of pipelined instruction scheduling which cannot awaken dependent instructions immediately, instructions dependent on indepen-

dent MOPs can be executed in the same clock cycle as the base 2-cycle scheduling case as long as the execution latency of the MOP is no greater than the scheduling loop delay.

Figure 6-15 shows an example of grouping independent instructions and their execution timings in base and macro-op cases. Two independent MOPs are created; instructions 1 and 2 are grouped due to no source operand; instructions 3 and 4 are grouped due to identical source operands, both of which depend on instruction 1. Although the execution of the grouped instructions is serialized, the scheduling of their dependent instructions 5 through 7 is not affected because the 2-cycle execution latency of a MOP hides the wakeup operation, enabling consecutive issue of dependent instructions.

Supporting independent MOPs does not require any fundamental modification in the base macro-op scheduling; independent MOPs are processed and scheduled in the same way as dependent MOPs. Independent MOPs are captured by the MOP detection logic after detecting all possible dependent MOPs, so that it does not lose any of the 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. Unlike detecting dependent MOPs which requires examining dependences, counting the number of source operands, and checking cycle conditions, independent MOPs require dependence checking only. One exception is for grouping a store address operation and a load. To prevent them from inducing a cycle through memory dependences (discussed in Section 6.3.2), such instruction pairs are not allowed.

In Section 7.5, I will present the number of independent MOPs captured by our proposed mechanism, measure the impact of independent MOPs on performance, and

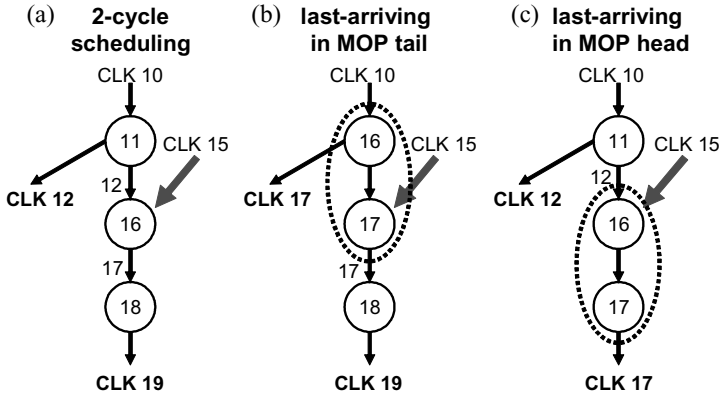


FIGURE 6-16. The effects of last-arriving operands.

demonstrate that they positively affect performance in many cases.

6.7.2 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 [27][56] that triggers issue of the MOP. Figure 6-16 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 6-16a and b). However, we observed that macro-op scheduling experiences difficulties in some benchmarks, 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 6-16b).

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 6-16c.

In Section 7.8.3, I will present the number of harmful MOPs, measure their impact on performance, and the effectiveness of filtering them.

6.8 Summary and Conclusions

The principles of macro-op scheduling and the details of the microarchitectural supports for enabling macro-op scheduling are discussed in this chapter. Ensuring back-to-back execution of dependent instructions requires scheduling logic to perform at the same rate as they are executed. Macro-op scheduling groups multiple instructions into macro-ops, and performs non-speculative pipelined scheduling of multi-cycle operations, enabling back-to-back execution of dependent instructions.

MOP detection logic 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. MOP formation logic reads MOP pointers and groups the original instructions into macro-op. The instruction scheduler performs pipelined scheduling of multi-cycle MOPs. An issued MOP is converted back to the original instructions, which are executed consecutively as if conventional atomic scheduling is performed.

There are several issues in implementing macro-op scheduling. Grouping instructions may induce cycles in the data dependence chain, which result in deadlock. To prevent this, MOP detection logic uses a simple heuristic that detects cycle conditions conservatively. The complexity of the detection logic is also considered. MOP detection logic can be pipelined by adopting multiple detection queues that examine instructions independently at multiple points. Several performance considerations are also made in this

chapter. A MOP of dependent instructions provides the most benefit since it relaxes the both scheduling atomicity and scalability simultaneously. A MOP of independent instructions does not relax the scheduling atomicity but is beneficial since issue queue contention is reduced. A MOP may degrade performance by unnecessarily delaying instructions due to last-arriving operands in MOP tail instructions. To prevent this, macro-op scheduling requires a filtering mechanism in MOP detection logic that depreciates harmful grouping.

In the next chapter, the effectiveness of macro-op scheduling, as well as many other aspects of detailed implementations and policies, will be evaluated.

Chapter 7

Experimental Evaluation of Macro-op Scheduling

Several aspects of macro-op scheduling are examined. The simulator as well as the parameters used for the base machine was described in Section 3.2.2 and Section 3.2.4. The necessary modifications in the microarchitecture for macro-op scheduling were described in Chapter 6.

7.1 Scheduler Configurations

To measure the effectiveness of macro-op scheduling, I modeled base scheduling, 2-cycle scheduling, macro-op scheduling with 2-source and 3-source wakeup logic. As mentioned in Section 3.2.4, base scheduling has ideally pipelined scheduling logic, which is conceptually equivalent to conventional atomic 1-cycle scheduling with one extra pipeline stage. All performance data are normalized to this case. 2-cycle scheduling has pipelined wakeup and select stages, resulting in a one-cycle bubble between a single-cycle instruction and its dependent instructions.

Macro-op scheduling is built on top of 2-cycle scheduling. The base macro-op scheduling configuration is as follows: the MOP detection and formation groups two dependent MOP candidate instructions (excluding loads). The MOP detection and formation has a 2-cycle scope, which captures up to eight instructions on the base 4-wide machine. The MOP detection logic is fully pipelined and its latency is three clock cycles from examining dependences to generating MOP pointers. MOP formation is performed in parallel with register renaming and hence the total number of pipeline stages is the same

```
1:      /* loop count is 1000 */
2:      ADDL $31, 1000, %0
3:  Loop:
4:
5:      /* N is the number of dependence chains */
6:      ADDL %1, 1, %1
7:      ADDL %2, 1, %2
8:      ADDL %3, 1, %3
9:      ....
10:     ADDL %N, 1, %N
11:
12:     /* repeat the above 50 times to reduce the loop overhead */
13:
14:     SUBL %0, 1, %0
15:     BNE %0, Loop
```

FIGURE 7-1. The basic structure of the microbenchmark.

in all scheduler configurations. The MOP detection mechanism performs a heuristic-based cycle detection described in Section 6.3.1. To prevent harmful MOP grouping, a filtering mechanism is used, as described in Section 6.7.2. In later sections, we will evaluate various configurations for MOP detection and formation.

7.2 Microbenchmark Results

To ensure the correct behavior of macro-op scheduling implemented on the timing simulator and demonstrate its potential, several microbenchmarks are tested. The basic code structure of the microbenchmark is illustrated in Figure 7-1. The details on compiling microbenchmarks were explained in Section 3.2.3.

In lines 6 to 10, a series of ADDL instruction create multiple dependence chains so that N instructions in the loop body generate N dependence chains, resulting in an IPC of N on an infinite machine if other conditions are ignored. For controlled experiments, the 4-wide base machine is configured to have a perfect branch predictor and perfect memory.

Figure 7-2 shows the IPCs measured on 1-cycle, 2-cycle and macro-op scheduling when N varies from one to eight. Base 1-cycle scheduling scales linearly until dependence

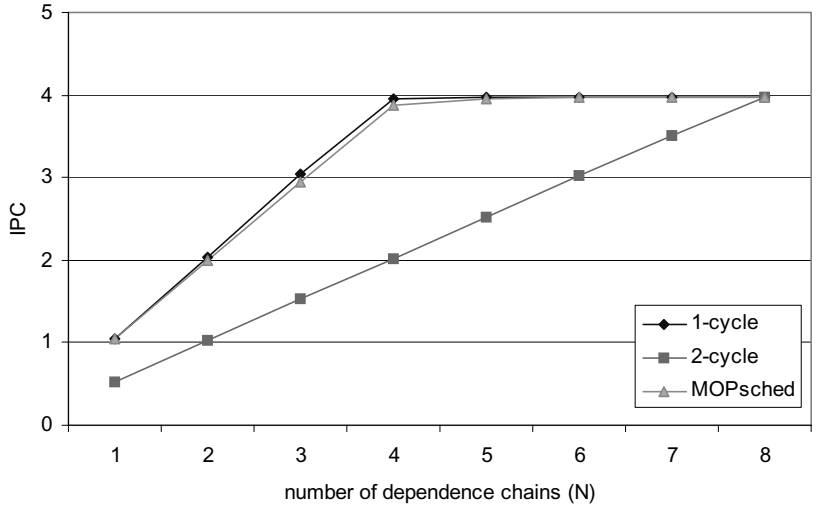


FIGURE 7-2. Performance of macro-op scheduling with microbenchmarks.

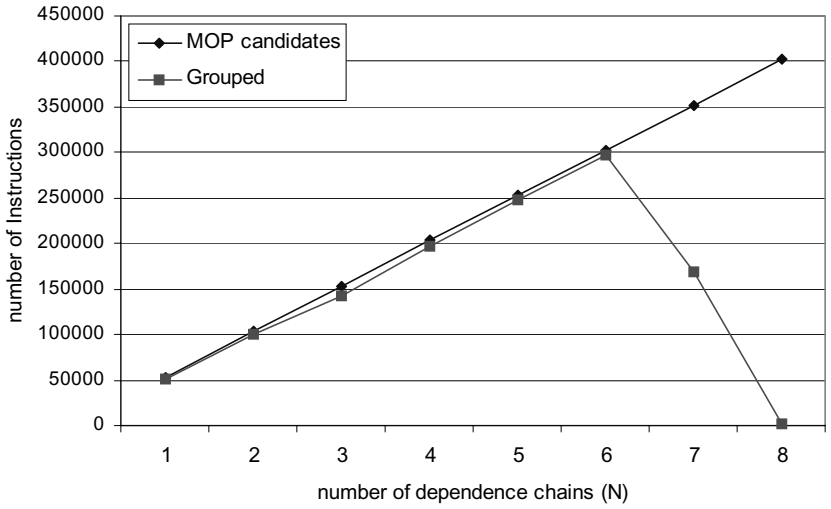


FIGURE 7-3. Instruction grouped for macro-op scheduling with microbenchmarks.

chains saturate the machine bandwidth (4-wide) at the point of N=4. 2-cycle scheduling achieves half the performance of 1-cycle scheduling until N becomes four since the height of the dependence chains becomes twice long. Macro-op scheduling performs as nearly well as 1-cycle scheduling, indicating that it successfully enables the pipelined scheduling logic to issue dependent instructions consecutively. Note that performance gaps between

1-cycle and macro-op scheduling cases mostly come from warming up MOP detection logic to generate pointers.

Figure 7-3 presents the MOP grouping coverage in microbenchmarks with different number of chains. Almost all candidate instructions are grouped into MOPs until N becomes seven, where the MOP coverage is significantly degraded. This is because the repetition of seven instructions frequently places two dependent instructions across three fetch cycles, while the MOP detection and formation only has a 2-cycle scope. If N becomes eight, which exceeds the current MOP scope, no dependent pairs are grouped. However, these cases already have plenty of independent instructions to issue every clock cycle and hence macro-op scheduling still achieves the maximum performance of the base machine, although it behaves just as conventional 2-cycle scheduling.

The microbenchmark results indicate that macro-op scheduling enables pipelined 2-cycle scheduling logic to achieve the full performance of conventional 1-cycle scheduling, which is the expected behavior that this study is aiming at. In the following sections, macro-op scheduling will be evaluated using SPEC2K benchmarks.

7.3 Instructions Grouped

Figure 7-4 shows the percentage of dependent instructions grouped for macro-op scheduling. Each benchmark has four stacked bars. The first two bars from the left (*max grouped* and *no EXE timing change*) represent the characterization data collected by the grouping policies described in Section 4.5.3. Compared with Figure 4-10 and Figure 4-11, the number of grouped instructions in both cases is lower because of the differences in grouping policy (2x, 8-instruction scope vs. 8x, 32-instruction scope). The third and the

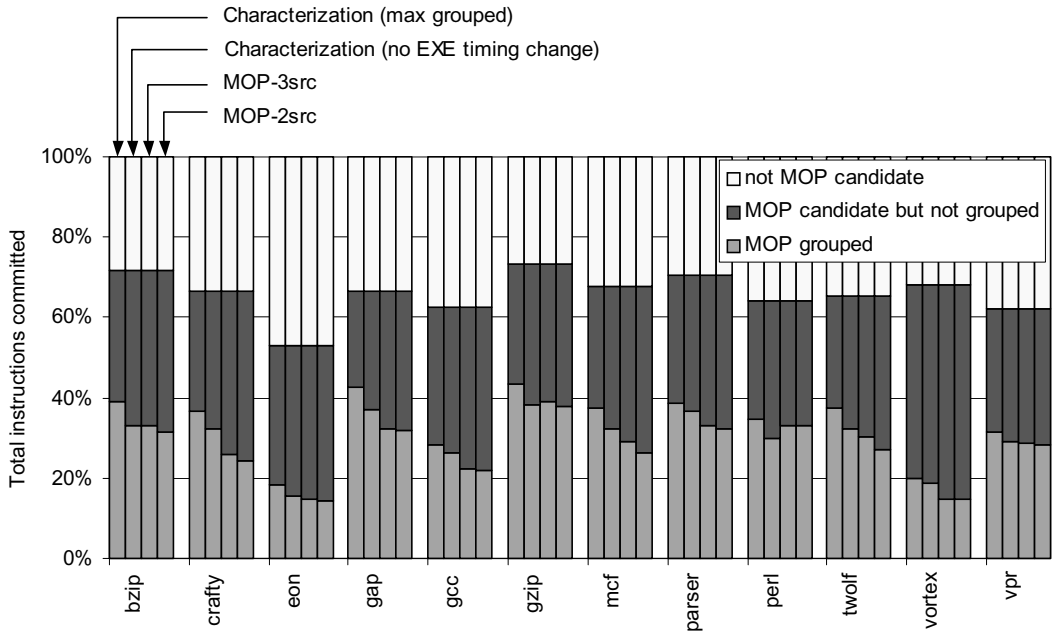


FIGURE 7-4. Instructions grouped in MOPs (2x, 2-cycle scope, no independent MOP).

last bars in each benchmark present the actual instructions grouped by the MOP detection and formation mechanisms implemented on the base machine configuration (4-wide, 128 ROB, 128 IQ) when the wakeup logic allows three (*MOP-3src*) or only two (*MOP-2src*) sources in a MOP. Therefore, the difference between the data here and the characterization data in the first two bars should be interpreted as coming from implementation limitations, such as discontinuous instruction fetch, 2-cycle scope instead of 8-instruction scope, cycle detection heuristic, and MOP pointer restriction across control instructions. Note that a MOP created across a mispredicted branch was not included in the *MOP grouped* category.

Across the benchmarks, our mechanism (third and fourth bars) captures 14.4 (*eon*) ~ 38.6% (*gzip*) of total instructions. Compared with the *max grouped* policy (left bar), some benchmarks lose many grouping opportunities since the *max grouped* policy does not consider performance. Compared with the *no EXE timing change* policy (second bar

from the left), the current mechanism achieves on average 92% and 89% of grouping opportunities for the 3-source and 2-source MOP cases, respectively. *Gzip* and *perl* capture slightly more MOPs than the characterization data because of the execution timing difference in the infinite and base machines. In addition, the grouped instructions captured by our mechanism contain a few harmful MOPs that the filtering mechanism cannot eliminate.

It is important to note that the MOP-2src case does not lose many grouping opportunities of the MOP-3src case. This result comes from the fact that not many instructions have two unique source operands in the programs we tested [56]. In addition, two instructions grouped in a MOP may share identical source operands. This result is consistent with the characterization data presented in Figure 4-10 and Figure 4-11, as the *min grouped* policy does not allow 2-source instructions as MOP tails but still captures many grouping opportunities.

Figure 7-5 categorizes the grouped instructions as a function of the source operands in MOP head and tail instructions. The y-axis represents the total grouped instructions in the 3-source case. Each benchmark has two stacked bars that represent the 3-source and the 2-source MOP cases. The 2-source MOP case is normalized to the 3-source case so that we can see the reduced grouping opportunities. As the legend illustrates, each category corresponds to the MOPs with different number of source operands. For example, the category labeled as *2-1* implies that the MOP head has two source operands and the tail has only one operand (except for the source dependence that links the two instructions). Only the unique source operands are counted and therefore the 2-source MOP case still has instructions in the *2-1* category since the MOP head and tail may share the same

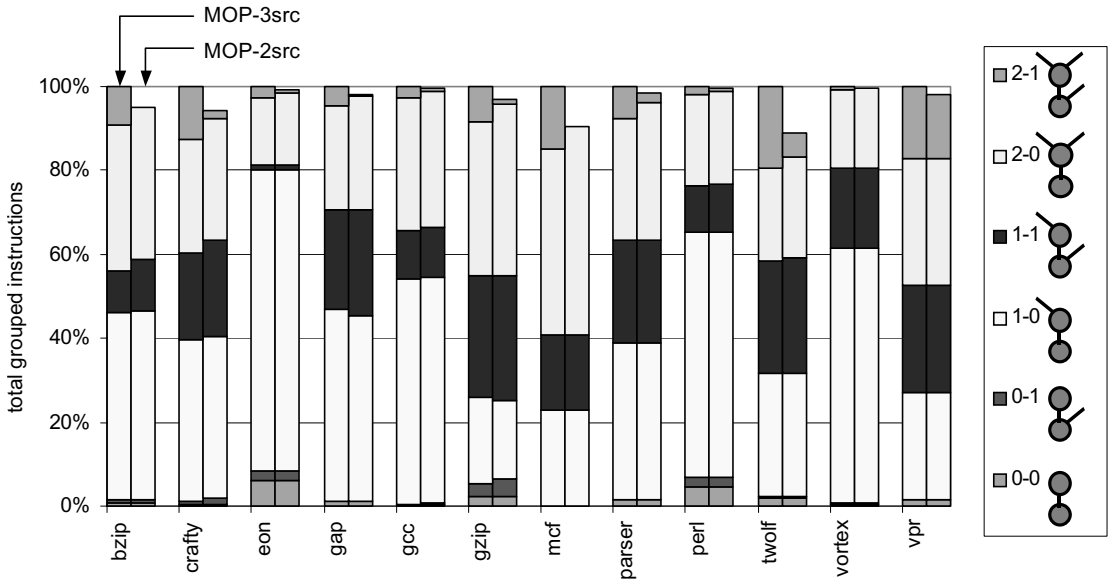


FIGURE 7-5. Grouped instructions categorized by the source operands mix.

source operands (e.g. `ADD r1 <- r3, r2` grouped with `AND r1 <- r1, r2`), resulting in only two unique source operands. Similarly, some MOPs in the *1-1* category have only one unique operand for the same reason.

The result in Figure 7-5 indicates that there are not many MOPs with three source operands (*2-1* category) even in the MOP-3src case so the MOP-2src loses only a small portion of the opportunities. Besides the attribute of instruction mix, this result is also partly affected by the filtering mechanism to avoid harmful MOPs, which avoids MOP tails with last-arriving operands and hence discourages MOP detection logic from grouping 2-source instructions as MOP tails.

Since the objective is to recover 2-cycle scheduling performance, we want to know how many value-generating candidate instructions are grouped, since these cases enable such instructions to be scheduled as if 1-cycle scheduling is performed. Figure 7-6 presents these data, in which the total value-generating candidate instructions are shown as dotted lines. The grouped instructions are shown in two separate categories; the *MOP-val-*

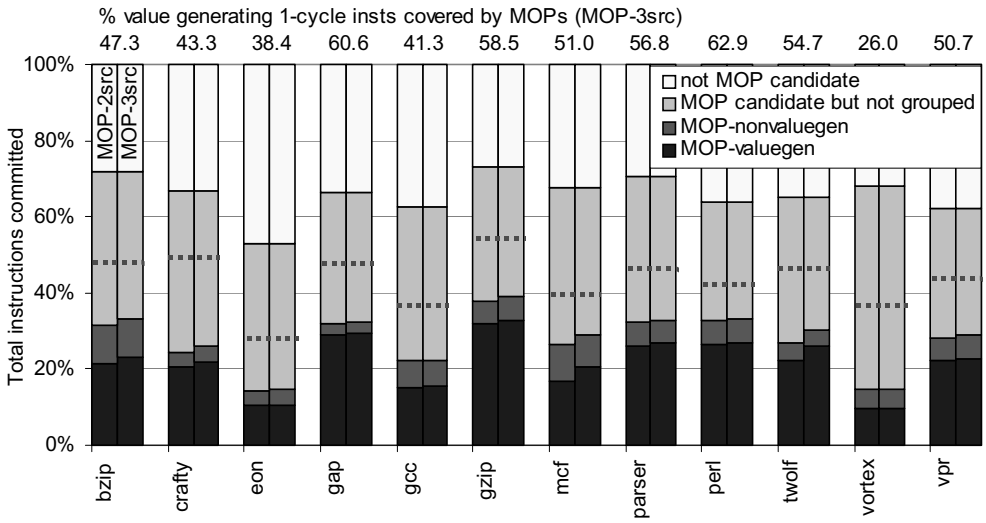


FIGURE 7-6. Coverage of value-generating candidate instructions.

uegen category represents value-generating candidate instructions grouped in MOPs; the *MOP-nonvaluegen* category represent other candidate instructions grouped as MOP tails that does not generate register dependences (e.g. branches or store address generations). Across the benchmarks, the data indicate that MOPs capture 26 ~ 63% of value-generating candidate instructions, which degrade performance in 2-cycle scheduling by preventing dependent instructions from being scheduled consecutively. If the MOP head has multiple dependent instructions that are awakened within the window, the wakeup timing of macro-op scheduling is identical to the 2-cycle scheduling case, and hence only one 1-cycle dependence edge between the MOP head and tail is recovered. Note that the MOP tail instructions do not experience this shortcoming and all dependent instructions will be scheduled consecutively. Table 7-1 presents the number of MOP head instructions with multiple dependent instructions. Note that these data do not directly represent the value degree of use [12] for MOP heads since only the dependence edges observed by their dependent instructions within the scheduler are measured. The second column of the table

Table 7-1: MOP heads with multiple dependent instructions (MOP-3src).

Benchmarks	MOP heads with multiple consumers / total MOP heads	MOP head candidates with multiple consumers / MOP head candidates with consumer(s)	MOP heads with multiple consumers / total instructions	Value-generating MOP candidates grouped / total instructions (same as *MOP-valuegen* in Figure 7-6)
bzip	23.2%	20.8%	3.8%	23.2%
crafty	20.5%	15.8%	2.7%	22.0%
eon	40.7%	37.8%	3.0%	10.7%
gap	31.4%	29.2%	5.1%	29.4%
gcc	24.0%	28.6%	2.7%	15.4%
gzip	7.2%	14.8%	1.4%	32.9%
mcf	19.8%	33.2%	2.9%	20.5%
parser	21.0%	21.4%	3.5%	27.0%
perl	24.5%	25.2%	4.1%	26.7%
twolf	20.6%	21.3%	3.1%	26.1%
vortex	25.4%	25.1%	1.9%	9.8%
vpr	11.4%	20.2%	1.6%	22.6%

represents the percentage of the grouped MOP heads that have multiple consumers (*MOP heads with multiple consumers / total MOP heads*). These numbers are correlated to the degree of use for MOP head candidate instructions, as the percentage of MOP head candidates with multiple consumers (the third column) shows a similar trend with a few exceptions. Their actual contribution to the total number of instructions is presented in the fourth column. Comparing it with the last column that presents the *MOP-valuegen* in Figure 7-6, we find that the majority of value-generating candidate instructions grouped in MOPs truly behave as in the 1-cycle scheduling case. On average 86% of the instructions in the *MOP-valuegen* category fully cover their 1-cycle dependence edges. The others (MOP heads with multiple consumers) cover one edge per each, which still benefits in many cases since short-distance dependence pairs are likely to be performance-critical, as discussed in Chapter 5.

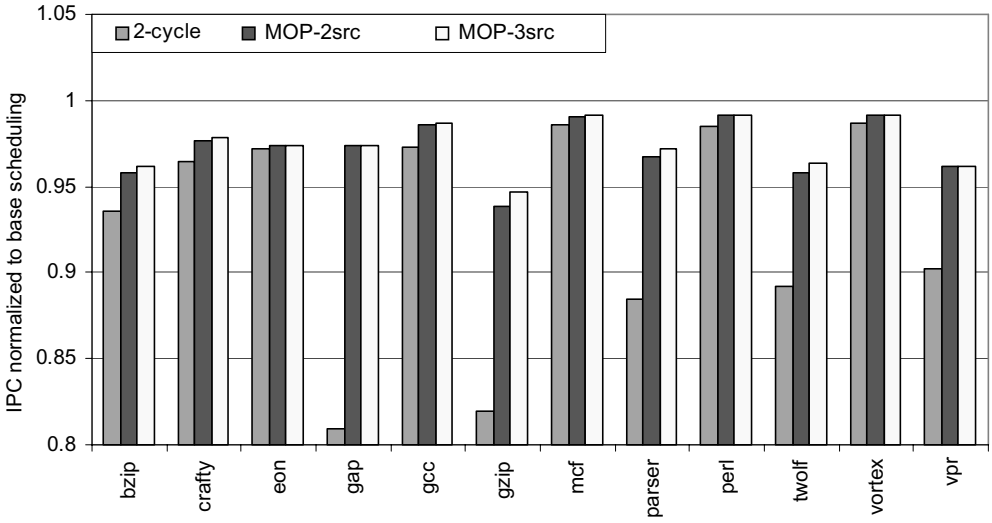


FIGURE 7-7. Performance of macro-op scheduling (no issue queue contention).

Figure 7-7 present macro-op scheduling performance when no issue queue contention exists (128-entry issue queue and ROB). Here, macro-op scheduling does not benefit from queue contention reduction. The y-axis represents the performance of 2-cycle scheduling, and macro-op scheduling with 2-source and 3-source MOPs. As discussed in Chapter 5, 2-cycle scheduling (shown in the left bars) suffers a performance drop of 1.3% (*vortex*) ~ 19.1% (*gap*).

Macro-op scheduling (middle and right bars) achieves 97.3% of the base performance on average. Since 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. Specifically, *gap, gzip, parser, twolf* and *vpr* experiences 10% or more performance degradation with 2-cycle scheduling but macro-op scheduling makes up a significant portion of this. Performance with 2-source MOPs is almost identical to that with 3-source MOPs, since both cases cap-

ture a similar number of MOPs as presented in Figure 7-5.

The relatively low coverage of MOP candidates due to long dependence edges 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 such 2-cycle-scheduling-insensitive benchmarks. 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. Specifically, short-distance dependence pairs are likely to deteriorate the ILP extractable by 2-cycle scheduling and to be performance critical. This was presented in Figure 5-8, which indicates that the instructions within a small range (i.e. eight instructions in program order) account for a significant portion of the wakeup activities in the instruction scheduler. The MOP detection algorithm is set to capture these short-distance dependent pairs, which increases the chances of complementing many performance-degrading instructions.

Figure 7-8 presents the percentage of 1-cycle dependence edges recovered by MOP scheduling. The y-axis in the graph represents the total dependence edges that actually trigger instruction issue. The dependence edges that are not observed by the scheduler, not last arriving operands, or cannot trigger issue due to structural hazards were not included in this graph. Each benchmark has two stacked bar for 2-source and 3-source MOP cases. Note that the 100% line does not represent the same number of dependence edges because of the timing difference between the two cases, although they are almost identical. Similarly, this graph does not strictly present how many dependence edges in 2-cycle scheduling are shortened by MOPs because of the timing difference between macro-op scheduling and the base 2-cycle scheduling. Each stacked bar has multiple categories;

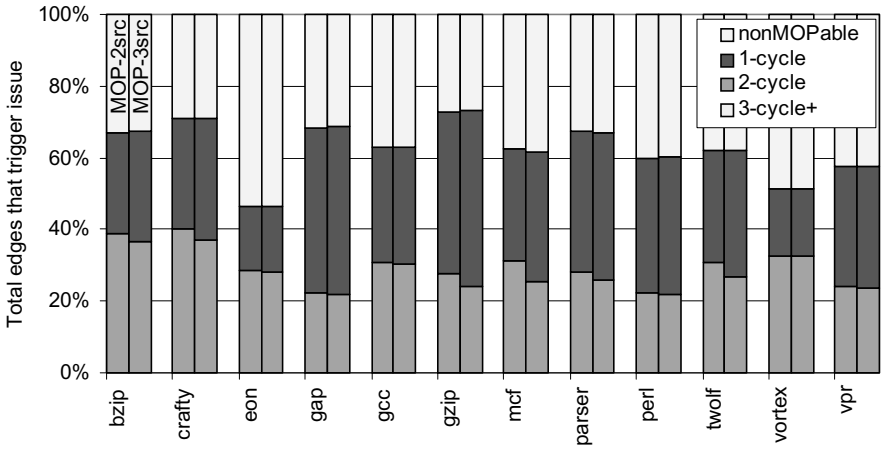


FIGURE 7-8. Dependence edges that trigger instruction issue in macro-op scheduling.

nonMOPable represents the ones that are created by non-MOP-candidate instructions with multi-cycle latencies. Hence, they are not negatively affected by 2-cycle scheduling; *1-cycle* represents the ones that MOPs enable to schedule dependence instructions consecutively, as if 1-cycle scheduling is performed. In the base 2-cycle scheduling case, these dependence edges will be included in the next *2-cycle* category. The dependence edges from a MOP head to its dependent instruction outside the MOP are not included here; *2-cycle* represents the dependence edges not covered by MOP scheduling so they behave as in 2-cycle scheduling; finally, *3-cycle+* represents the edges that are delayed by the last-arriving operands in MOP tails (discussed in Section 6.7.2). Since there are extremely few dependence edges in this category (maximal 0.2% in *twolf*), they do not show in the graph.

Macro-op scheduling enables 36% (*vortex*) ~ 68% (*gap*) of 1-cycle dependence edges to behave as if 1-cycle scheduling is performed, recovering a significant portion of the base 1-cycle scheduling performance. To highlight the correlation between performance and coverage, Table 7-2 compares the data in Figure 7-8 with the performance recovered by macro-op scheduling. The second column represents the IPC degradation in

Table 7-2: 1-cycle dependence edges recovered by macro-op scheduling (MOP-3src).

Bench-marks	IPC loss due to 2-cycle scheduling	IPC loss recovered by macro-op scheduling	Value-generating candidate instructions grouped in MOPs (identical to Figure 7-6)	1-cycle dependence edges recovered by MOP scheduling (identical to Figure 7-8)
bzip	6.4%	40.4%	47.3%	46.0%
crafty	3.5%	40.1%	43.3%	47.7%
eon	2.8%	6.0%	38.4%	39.0%
gap	19.1%	86.0%	60.6%	68.0%
gcc	2.7%	50.3%	41.3%	51.8%
gzip	18.1%	70.4%	58.5%	66.7%
mcf	1.4%	37.7%	51.0%	58.8%
parser	11.6%	76.0%	56.8%	61.5%
perl	1.5%	45.0%	62.9%	63.2%
twolf	10.8%	66.2%	54.7%	56.9%
vortex	1.3%	36.0%	26.0%	36.6%
vpr	9.8%	60.4%	50.7%	59.2%

2-cycle scheduling compared with the base case. The third column shows how much IPC degradation is recovered by macro-op scheduling. The fourth column represents the coverage of value-generating candidate instructions (shown in Figure 7-6). The last column shows the percentage of 1-cycle dependence edges recovered (shown as *1-cycle* category in Figure 7-8). If we assume that the degree of performance recovery is proportional to the dependence edge coverage, the numbers in the third and the fifth columns should be similar. In fact, they follow a similar trend, with a few variations. From the table, we find that the benchmarks sensitive to 2-cycle scheduling tend to recover slightly more performance compared to the edge coverage, while the other insensitive benchmarks achieve a lower degree of performance recovery.

7.5 Impact of Independent MOPs

In Section 6.7.1, I discussed grouping two independent instructions into MOPs and its potential benefits of reducing queue contention. These independent MOPs are

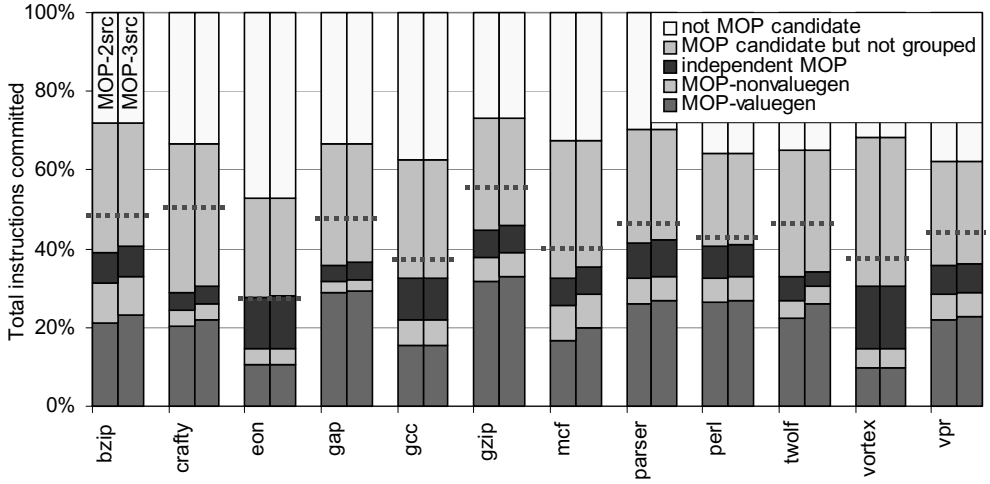


FIGURE 7-9. Instructions grouped in MOPs (with independent MOPs).

restricted to grouping two instructions with either no source operands or identical source operands, which are guaranteed to execute in parallel as long as there is no structural hazard. Figure 7-9 shows the number of independent instructions grouped into MOPs captured within the same 2-cycle scope as the base case. The data in this graph are plotted in the same way as in Figure 7-6, except for an added category of independent MOPs. Independent MOPs increase the total number of grouped instructions to 32.4% (from 28% when no independent instructions are grouped). This potentially achieves an average 16.2% reduction in the number of instructions inserted into the scheduler.

Since the base machine configuration has the 128-entry issue queue and the same size ROB, no issue queue contention exists, so the increase in the number of grouped instructions does not help improve performance. Rather, the negative effects of independent MOPs (discussed in Section 6.7.1) may degrade performance in some timing-critical cases. To measure this effect, Figure 7-10 presents the performance of macro-op scheduling with independent MOPs and compares it against the base macro-op scheduling case. In each benchmark, the first three bars are identical to those of Figure 7-7. The last two

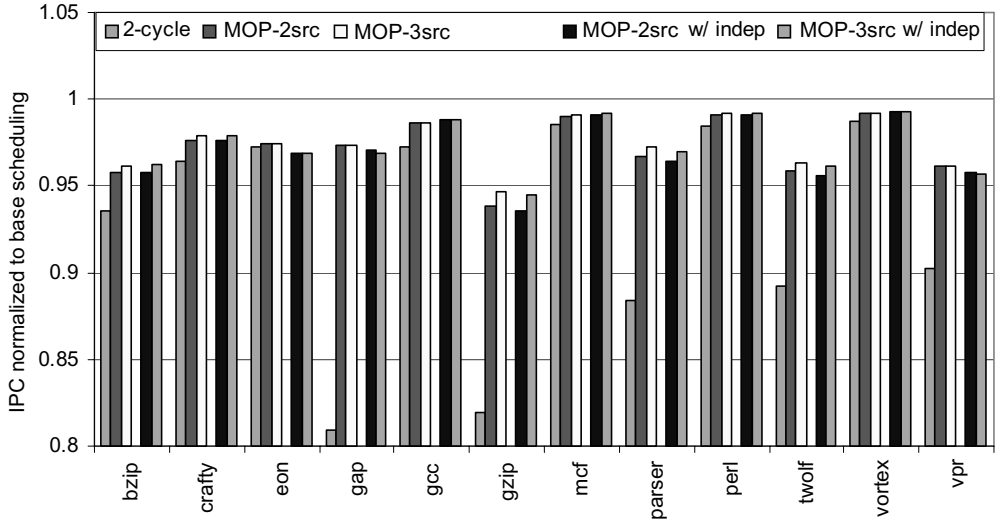


FIGURE 7-10. Performance impact of independent MOPs (no issue queue contention).

bars represent the effect of independent MOPs (*MOP-2src* and *-3src with indep*). Macro-op scheduling experiences slight performance degradation over the base case in some benchmarks. In particular, *eon* shows even worse performance than 2-cycle scheduling, although the difference is not significant and the performance gain from base macro-op scheduling is marginal. The overall performance is almost identical with or without independent MOPs and the negative performance impact of independent MOPs is negligible.

7.6 Performance of Macro-op Scheduling with Queue Contention

As the issue queue size is reduced and the contention increases, macro-op scheduling starts benefiting from relaxed scalability, since fewer schedulable units will be inserted into the issue queue. To measure this effect, the graphs in Figure 7-11 through Figure 7-13 present the performance of macro-op scheduling as the issue queue size is reduced down to 64, 48 and 32 entries. The average macro-op scheduling performance with various issue queue sizes is also plotted in Figure 7-14. For these experiments, both independent and

dependent MOPs are used to maximize the benefits of reducing issue queue contention.

The benefits of macro-op scheduling are the relaxed atomicity and scalability constraints. The graph in Figure 7-14 indicates that macro-op scheduling performs better when issue queue contention is higher. With a 32-entry issue queue, macro-op scheduling performs even better than the base 1-cycle scheduling because of high degrees of issue queue contention. As the issue queue size increases to 48 entries or more and the contention is reduced, the benefit from the relaxed scalability also decreases and therefore the two performance curves of 1-cycle and macro-op scheduling diverge. Although macro-op scheduling compensates for only on average ~60% of the IPC degradation due to 2-cycle scheduling with the 64- or 128-entry issue queue, it is effective especially for the benchmarks sensitive to 2-cycle scheduling.

Comparing the 2-source and 3-source macro-op scheduling cases, the result shows that 2-source MOPs reap most of the benefits of the 3-source case. Hence, macro-op scheduling does not necessarily require tag matching logic to support three source operands for better performance. In some benchmarks, the MOP-2src case performs slightly better than the MOP-3src case although slightly less MOPs are captured. This comes from the negative effect of last-arriving operands in the MOP tail, even though the detection logic filters out many MOPs exhibiting this behavior.

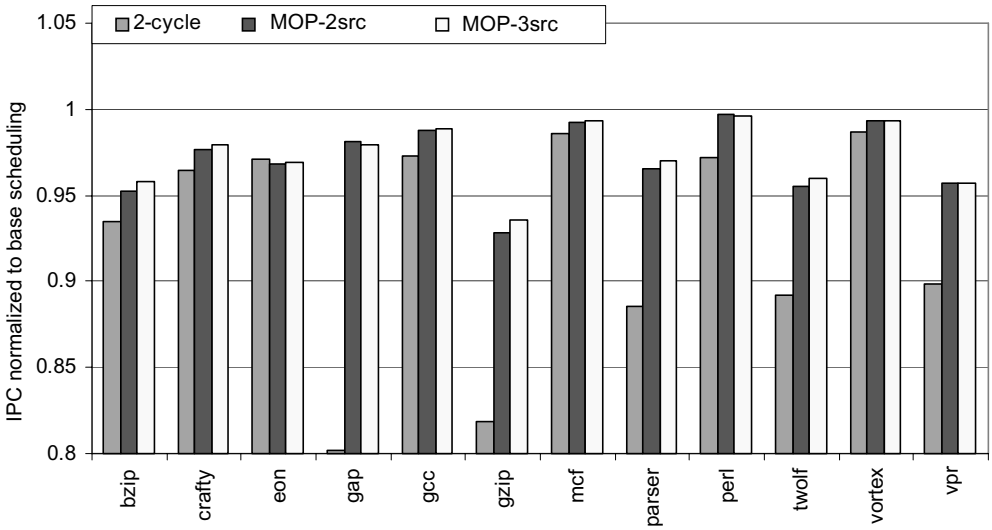


FIGURE 7-11. Performance of macro-op scheduling (64-entry issue queue).

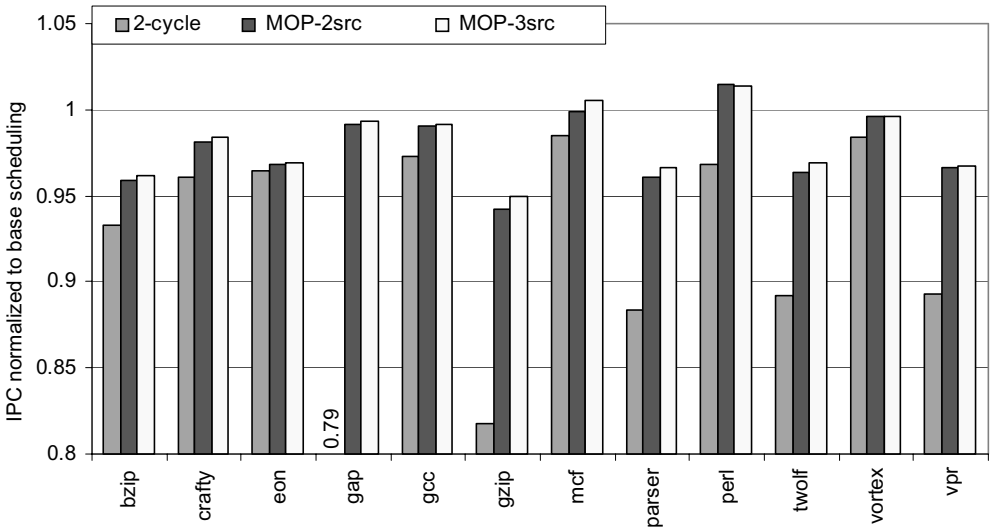


FIGURE 7-12. Performance of macro-op scheduling (48-entry issue queue).

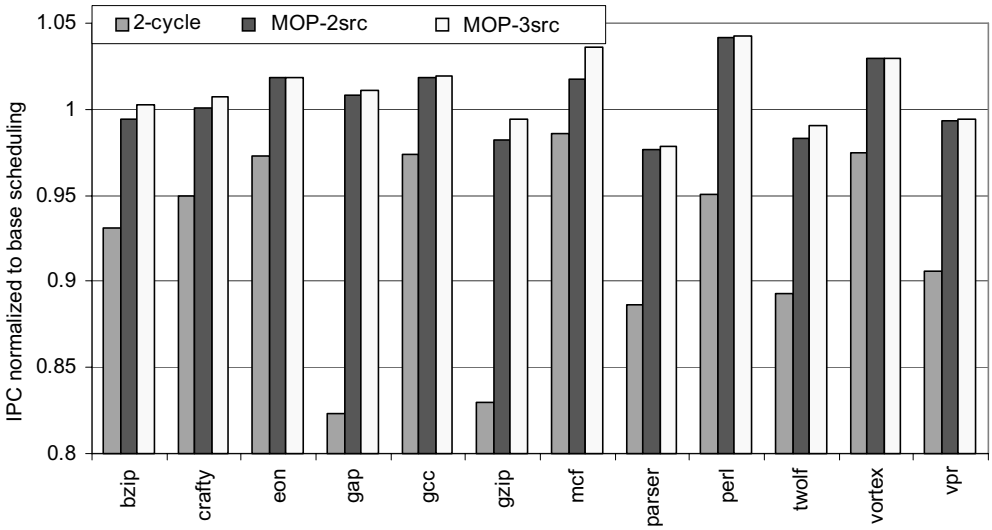


FIGURE 7-13. Performance of macro-op scheduling (32-entry issue queue).

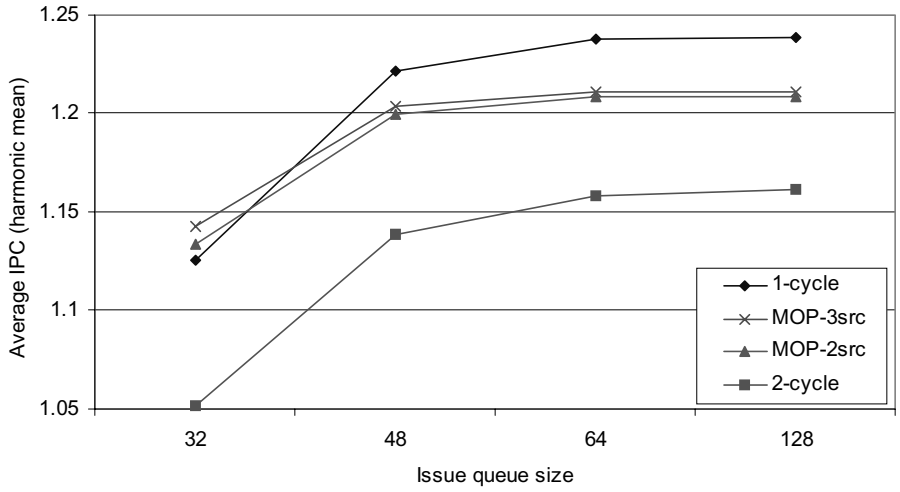


FIGURE 7-14. Performance of various instruction schedulers.

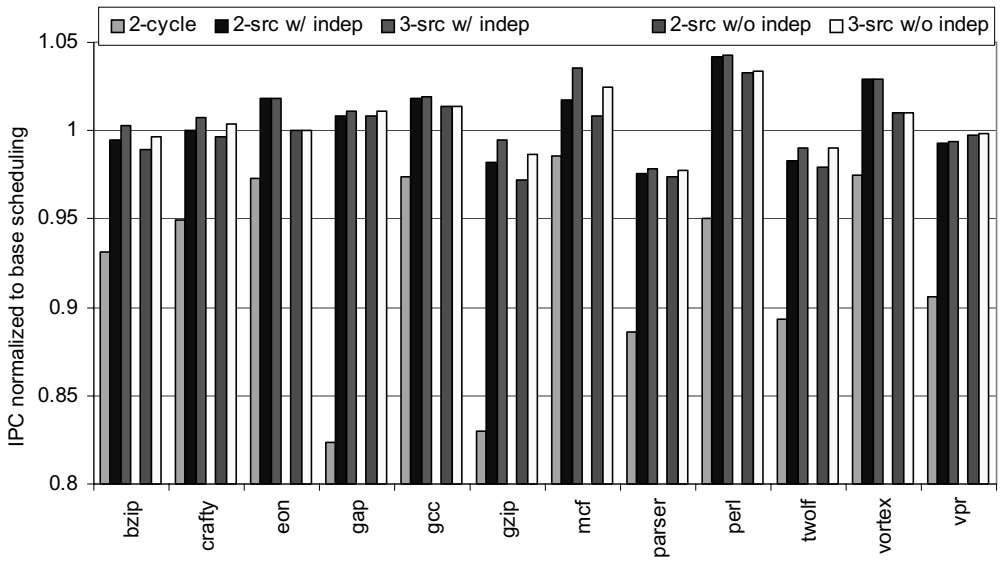


FIGURE 7-15. Performance impact of independent MOPs (32-entry issue queue).

In order to show the performance benefit from independent MOPs when the queue contention is high, Figure 7-15 compares the performance of macro-op scheduling with and without independent MOPs. The base machine has the 32-entry issue queue for this measurement. The first three bars from the left in each benchmark are identical to those in Figure 7-13, in which both dependent and independent MOPs are generated. The last two bars (*2-src* and *3-src w/o indep*) represent performance when independent MOPs are not generated and hence the queue contention is reduced less compared with the previous case. A few benchmarks such as *eon, mcf* and *vortex* show measurable performance gaps over 1% (up to 2.0% in *vortex*). This result is consistent with the data in Figure 7-9 where those benchmarks capture more independent MOPs than the others. *Vpr* shows a slight slowdown due to a secondary effect of independent MOPs, as we already observed in Figure 7-10. Although the current result indicates that the performance benefits from independent MOPs are not significant, they are still effective in reducing queue contention and achieving better performance for some cases. Especially, they are critical in increasing the

machine bandwidth for macro-op execution, which will be discussed in Chapter 8.

7.7 Impact of MOP Formation

7.7.1 MOP scope

In Section 6.2.2, I characterized the distribution of the dependence edges that wake up their dependent instructions, and found that capturing a short range of instructions, i.e. an 8-instruction scope covers the vast majority of those edges. The current macro-op scheduling is configured to capture MOPs with a 2-cycle scope based on this result. To measure the impact of MOP scope on coverage and performance, we vary the scope from one cycle (capturing up to four instructions) to four cycles (capturing up to 16 instructions). The base machine has the 128-entry issue queue and hence macro-op scheduling does not benefit from reduced queue contention, although a wider scope increases the number of MOPs and reduces more contention.

Figure 7-16 shows the number of dependent instructions grouped in MOP-3src (without independent MOPs) with various MOP scopes. The graph was plotted in the same way as Figure 7-4. Each benchmark has four stacked bars that represent MOP scopes of one cycle through four cycles. With the 1-cycle scope, macro-op scheduling loses a significant portion of grouping opportunities compared to other wider scopes. Note that the result may not seem consistent with other characterization data of the dependence edge distance between MOP candidate instructions shown in Figure 4-7, where many of them are placed within four instructions. This is because the MOP detection and formation logic uses a 1-cycle scope (instead of a 4-instruction scope), which creates discontinuity in detecting and formatting MOPs. For example, if a candidate instruction is placed in the

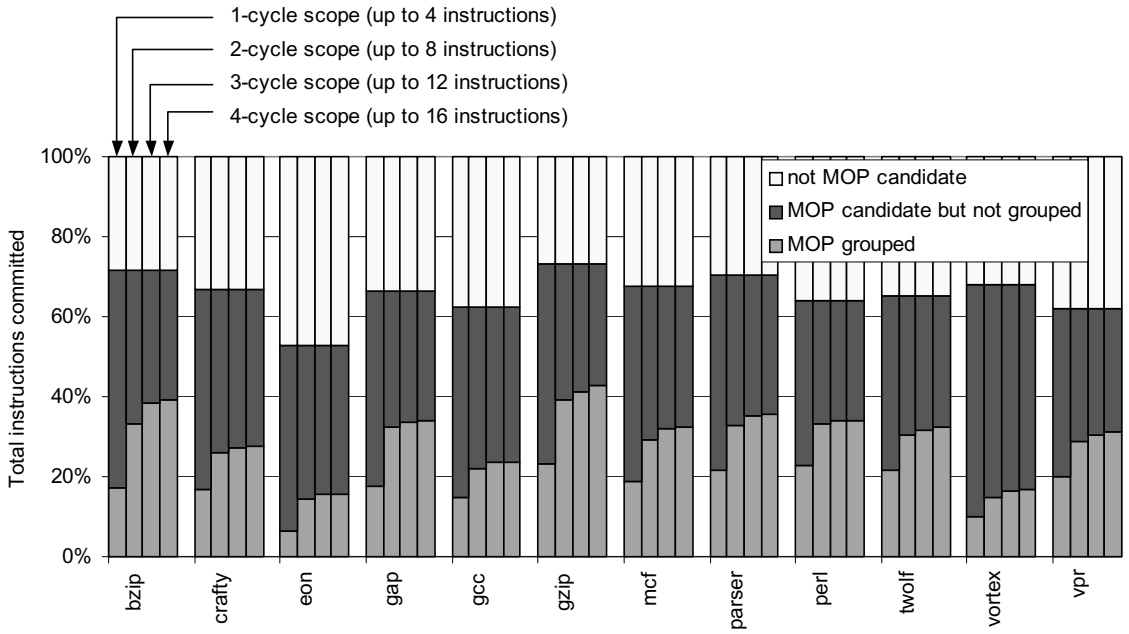


FIGURE 7-16. Impact of MOP scope on MOP coverage (MOP-3src, no independent MOP).

fourth slot in a fetch group on the base 4-wide machine, it cannot be grouped as a MOP head with other candidate tail in the following pipeline stage. If we ignore other secondary effects, a 1-cycle scope is similar to an effective 2.5-instruction scope[1], which decreases the effectiveness of MOP formation.

Although the 2-cycle MOP scope cannot fully achieve an 8-instruction scope for the same reason, its effective scope of 6.5 instructions[2] is large enough to capture many candidate pairs. Wider 3- or 4-cycle scopes achieve measurable increases over the 2-cycle scope in some cases, although they may not be significant. Compared with the 4-cycle

1. Assuming that instruction fetch fills all instruction slots (four instructions) every clock cycle, a MOP head can be placed in one of the four slots with the probability of 0.25 each. If a MOP head is placed in the first instruction slot, an effective MOP scope is four instructions. If the MOP head placed in the second or third slot, the effective MOP scope is reduced to three or two instructions, respectively. Therefore, the combined MOP scope is in effect 0.25 X (4 + 3+ 2 + 1), which is 2.5 instructions.
2. Based on the same reasoning as the previous case, a 2-cycle scope captures 0.25 X (8 + 7 + 6 + 5), which is 6.5 instructions.

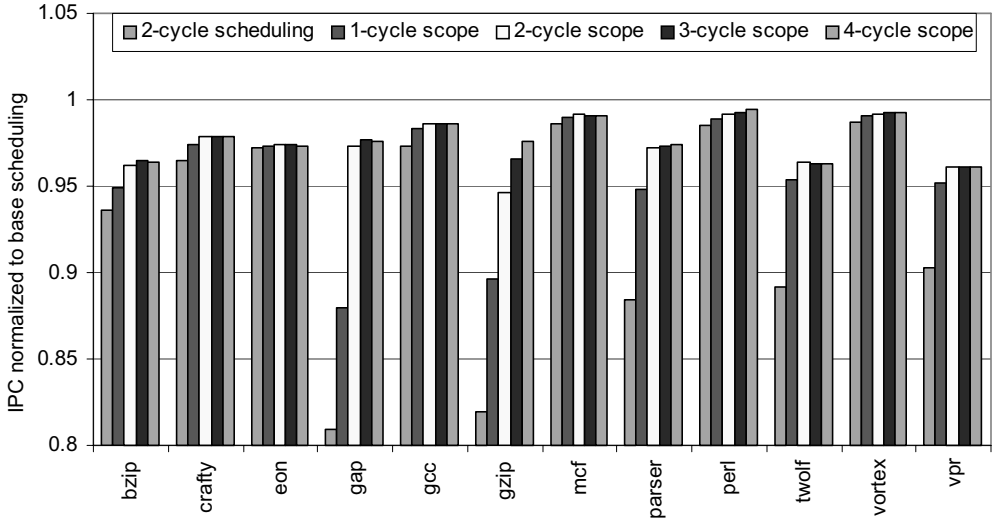


FIGURE 7-17. Performance impact of MOP scope (MOP-3src, no independent MOP, no issue queue contention).

scope, the 2-cycle scope achieves 92% of MOP grouping opportunities captured by the 4-cycle scope on average.

Figure 7-17 presents the performance sensitivity of macro-op scheduling to various MOP scopes. The first bar on the left in each benchmark represents the base 2-cycle scheduling performance with a 128-entry issue queue. The remaining four bars correspond to 1-cycle to 4-cycle scopes, where the 2-cycle scope (middle bar) is identical to the MOP-3src case in Figure 7-7. MOP detection captures dependent MOPs only. The negative impact of the 1-cycle scope is substantial. The differences between 1-cycle and 2-cycle scopes in *gap* and *gzip* are particularly significant since the 1-cycle scope achieves only 52% and 59% of MOPs that the 2-cycle scope captures for the two benchmarks, respectively. The 2-cycle scope achieves most benefits of wider scope in all benchmarks except *gzip* where the 3- and 4-cycle scopes further improve its benefit, achieving 97.5% of the performance in the base 1-cycle scheduling case. Note that a wider scope does not necessarily improve performance over the 2-cycle scope in some cases. This result partly comes

from the negative effects of harmful MOP grouping (Section 6.7.2). In addition, the insertion policy to put instructions into the issue queue (Section 6.4.3) may negatively affect performance. When the MOP head is inserted, it sets the pending bits and waits for the matching MOP tail to be inserted. If the two instructions are placed far away from each other in a wider MOP scope case, issuing the MOP head can be unnecessarily delayed, degrading performance compared to the base case. It is also important to note that the benefit of reduced queue contention due to more MOPs captured in wider scopes was not reflected in this experiment, since the current base machine (128-entry issue queue) does not have any issue queue contention. It may be critical to capture more MOPs on a machine with high issue queue contention.

7.7.2 MOP size

Figure 7-18 shows performance when macro-op scheduling allows bigger MOP sizes. The base macro-op scheduling uses 2x MOPs that group two instructions each. Two larger MOP sizes are tested; the *3x MOP* configuration allows up to three instructions. Grouping two instructions are allowed; the *8x MOP* configuration allows up to eight instructions grouped in a MOP. Fewer than eight instructions are also allowed. For the 3x and 8x MOP cases, four and nine source operands in a MOP should be allowed to achieve their full benefits, respectively. Although these configurations potentially create difficulties in MOP pointer generation and formation process, I optimistically assume that the existing mechanism can correctly handle bigger MOPs for this experiment. The performance data in Figure 7-18 are normalized to the base macro-op scheduling (MOP-3src) with the 32-entry issue queue. Independent MOPs are generated in all configurations to maximize the benefit of reducing issue queue contention.

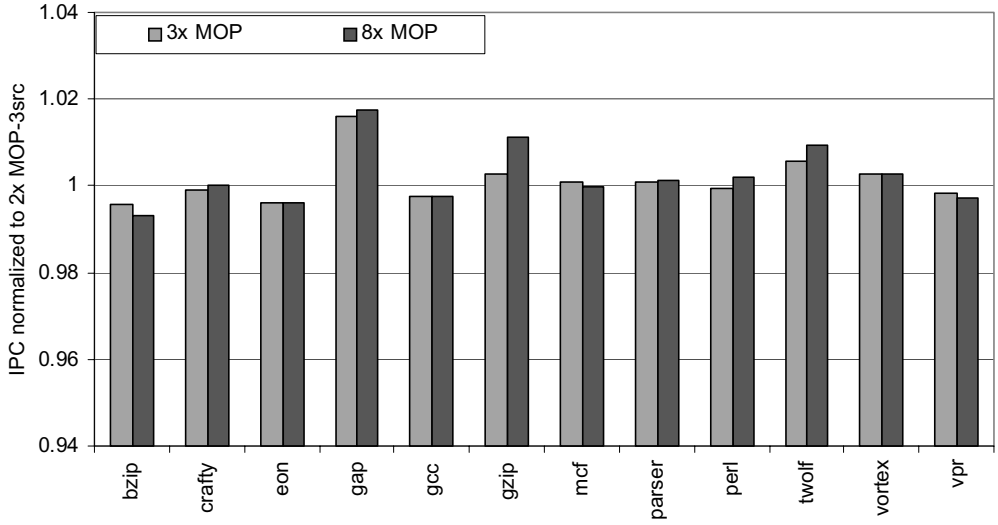


FIGURE 7-18. Performance impact of MOP size (32-entry issue queue).

Since the characterization data of bigger MOP sizes shown in Figure 4-14 and Figure 4-16 indicate that capturing more than three instructions in a MOP is not frequent enough, further relaxing the scheduling atomicity constraint (i.e. deeper pipelining the scheduling logic) is not considered for this experiment. Therefore, the performance benefits in this experiment come only from further reducing queue contention. Note that 3x macro-op scheduling with 3-cycle scheduling provides only a marginal performance benefit due to a limited number of bigger MOPs.

The results indicate that larger MOP sizes do not always improve performance. *Gap*, *gzip* and *twolf* benefit from bigger MOPs, whereas *bzip*, *eon*, *gcc* and *vpr* show slight performance degradations. This is primarily because bigger MOPs do not necessarily capture significantly more instructions in many cases but rather increase the probability of instruction serialization. Considering the extra complexity incurred by bigger MOPs and marginal performance benefits, the current policy of grouping two instructions would be a good design choice for the programs that we tested.

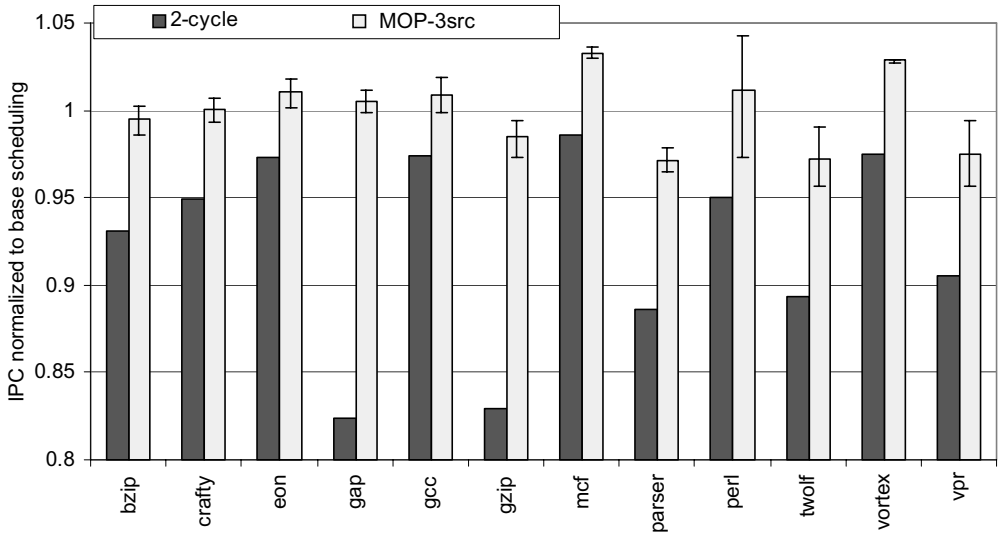


FIGURE 7-19. Performance impact of extra MOP formation stages (32-entry issue queue).

Although the MOP formation process can be overlapped with register renaming, possible extra logic complexity may require a few pipeline stages to be added to the front-end of the pipeline. To measure its performance impact, I evaluate macro-op scheduling with one or two extra stages between the rename and the queue stage, which increases the branch resolution loop [6] and hence potentially degrades performance due to the increased branch misprediction penalty. The branch predictor performance in each benchmark, as well as other runtime characteristics was presented in Table 3-3.

Figure 7-19 presents the performance of macro-op scheduling with zero, one or two extra pipeline stages, measured on the base machine with a 32-entry issue queue. Independent MOPs are included for this measurement. In the graph, the solid bars represent performance with one extra MOP formation stage, and the error bars indicate performance with zero or two extra stages. The performance with zero stage (presented with upper error bars) is identical to that of Figure 7-13. The benchmarks with low branch pre-

diction performance such as *perl, twolf* and *vpr* show relatively high sensitivity to the extra pipeline stages. *Perl* is measured to be the most sensitive because of the highest misprediction rate (18.16%) among the benchmarks. Other benchmarks with higher prediction performance lose less than 2% of IPC even with two extra stages.

7.8 Impact of MOP Detection

7.8.1 MOP detection latency and pipelinability

So far, I assumed fully-pipelined MOP detection logic with an optimistic 3-cycle latency from starting MOP detection to generating MOP pointers. Because of the complexity of the detection process, the actual hardware implementation may require longer latency. The complexity issues in implementing MOP detection logic were discussed in Section 6.3.4. In this section, I will first measure the impact of detection latency on performance, and then evaluate its pipelinability.

Table 7-3 presents the impact of detection latency on the number of MOPs and performance. The base macro-op scheduling has fully pipelined MOP detection logic with 0 extra detection latency (in addition to the base 3-cycle latency, which is unavoidable in the current simulator implementation). For the comparison cases, we test pessimistic 50- and 100-cycle extra latencies for the detection process. The second and third columns in the table show the number of MOPs and IPCs normalized to the base case.

On average, macro-op scheduling with 50-cycle and 100-cycle detection latencies achieve 98.2% and 97.4% of the MOPs in the base case, respectively. The worst case is measured to be 94% in *twolf*. *Bzip* exhibits a slight increase within the noise. The IPC degradation due to longer detection latency is on average 0.23% and 0.31% on the 50- and

Table 7-3: Impact of MOP detection latency (MOP-3src, no independent MOP, 128-entry issue queue).

Bench-marks	# MOPs with extra latency / # MOPs in the base case		IPC with extra latency / IPC in the base case	
	50 cycles	100 cycles	50 cycles	100 cycles
bzip	1.001	1.001	1.000	1.000
crafty	0.961	0.943	0.999	0.999
eon	0.994	0.983	0.999	0.998
gap	0.977	0.963	0.996	0.988
gcc	0.968	0.945	0.999	0.998
gzip	0.998	0.998	1.000	1.000
mcf	0.999	0.998	1.000	1.000
parser	0.954	0.945	0.989	0.988
perl	0.999	0.999	0.999	0.999
twolf	0.947	0.940	0.989	0.989
vortex	0.987	0.978	0.999	0.999
vpr	0.997	0.997	0.999	0.999

100-cycle cases, respectively (worst 1.2% in *parser*). The data in the table indicate that the MOP detection latency does not significantly affect macro-op scheduling, since MOP pointers stored in the instruction cache are used repeatedly.

A more important issue in MOP detection logic implementation is its pipelinability. To measure the impact of pipelining the logic by creating discontinuity in the process and reversing the order of dependence checking, we test several detection logic configurations with various pipeline intervals, as shown in Table 7-4. The details of each parameter in the table were discussed in Section 6.3.4. The MOP generation rate is fixed to one instruction per cycle in each detection queue, assuming that each instruction serially looks up a table to find its matching MOP head. The MOP detection and formation scope is fixed to two cycles (up to eight instructions) for all configurations.

Table 7-5 presents the impact of the pipelined MOP detection on the number of MOPs captured. Figure 7-20 presents the performance impact of pipelined MOP detection logic. The 4-interval case loses many opportunities since the discontinuous detection pro-

cess creates many “left-overs” in the instruction stream, as explained in Section 6.3.4. The intervals of eight or 16 instructions achieve most of the benefit of the fully pipelined detection logic. Note that longer pipeline intervals do not further improve the MOP coverage and performance over 8- or 16-interval cases since they increase the detection latency. The result indicates that pipelining the MOP detection logic with an interval of eight or more instructions does not significantly degrade the efficiency of the MOP detection process. Considering the number of queue entries required and performance, a pipeline interval of eight instructions would be a good design choice.

Table 7-4: Configurations for pipelining MOP detection logic.

Configuration	MOP detection scope (cycles)	Pipeline interval (= # entries / queue)	MOP generation rate (MOPs / cycle)	Extra detection latency (base 3 cycles)	Total detection queue entries
4-interval	2	4	1	14	56
8-interval	2	8	1	14	56
16-interval	2	16	1	20	80
32-interval	2	32	1	36	144
64-interval	2	64	1	68	272

Table 7-5: Impact of pipelined MOP detection on MOP coverage (MOP-3src, no independent MOP, 128-entry issue queue).

Benchmarks	# MOPs captured / # MOPs in the base case (3 cycles, fully pipelined)				
	4-interval	8-interval	16-interval	32-interval	64-interval
bzip	0.842	1.012	1.001	1.000	1.001
crafty	0.788	0.967	0.971	0.967	0.953
eon	0.709	0.981	0.989	0.994	0.985
gap	0.876	0.988	0.990	0.983	0.970
gcc	0.850	0.974	0.980	0.975	0.959
gzip	0.853	1.027	1.000	0.999	0.998
mcf	0.907	1.002	1.000	0.999	0.998
parser	0.870	0.968	0.964	0.959	0.951
perl	0.732	0.998	0.999	0.999	0.998
twolf	0.933	0.968	0.959	0.954	0.945
vortex	0.847	0.978	0.988	0.989	0.983
vpr	0.906	0.999	0.998	0.998	0.996

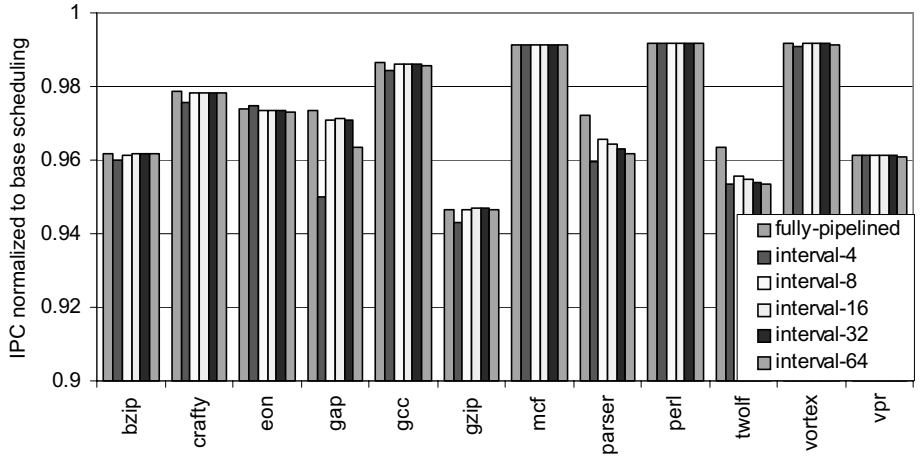


FIGURE 7-20. Performance sensitivity to pipelined MOP detection (MOP-3src, no independent MOP, 128-entry issue queue).

7.8.2 MOP detection algorithm

For the MOP detection process, each MOP head candidate searches for a matching MOP tail candidate in the forward scan algorithm. In contrast, MOP tail candidates search for matching MOP heads in the backward scan algorithm. The base macro-op scheduling uses the backward scan algorithm, in which the nearest parent instruction is chosen for a matching MOP head, if the instruction has two source operands.

Table 7-6 presents the efficiency of the forward scan algorithm compared with the backward scan algorithm. The second and third columns show the number of MOPs and IPCs in forward scan normalized to the backward scan case. The results indicate that the two scan algorithms perform similarly. From the perspective of hardware complexity, the backward scan algorithm would be preferable since MOP detection logic can be configured to examine only the instructions that correspond to source operands, without examining all instructions within the given MOP scope.

Table 7-6: Impact of MOP detection algorithm (MOP-3src, no independent MOP, 128-entry issue queue).

Benchmarks	# MOPs with forward scan / # MOPs with backward scan	IPC with forward scan / IPC with backward scan
bzip	1.001	1.000
crafty	1.004	1.000
eon	0.999	0.999
gap	1.005	0.998
gcc	0.997	0.999
gzip	1.019	0.998
mcf	1.001	1.000
parser	0.997	0.997
perl	0.999	1.000
twolf	0.996	0.998
vortex	0.994	0.999
vpr	1.000	1.000

7.8.3 Filtering harmful grouping

In Chapter 4 where the groupability of instructions is measured, we observed that inconsiderate MOP grouping reduces parallelism by serializing instructions. In Section 6.7.2, I discussed the negative effect of last-arriving operands in MOP tails. The MOPs unnecessarily delayed can be divided into two categories: *not useful* and *harmful* MOPs. If the MOP head has only the MOP tail as dependent instruction, it is not useful but does not degrade performance compared with 2-cycle scheduling since the execution timing of instructions dependent on the MOP does not change. If the MOP head has other dependent instructions outside the MOP, it is harmful and degrades performance by delaying other instructions.

Table 7-7 presents the impact of filtering not useful and harmful MOPs on MOP grouping. The second column in the table shows the number of MOPs captured after filtering, compared with the no-filtering case. Some benchmarks such as *gap* and *twolf* are sensitive to the filtering, and lose 8~9% of grouping opportunities. On average, the filtering mechanism reduces the number of MOPs by 2.6% across benchmarks. The third and fourth columns present the percentage of not useful and harmful MOPs out of the total MOPs, with and without filtering them. With filtering, harmful MOPs (third column) account for a relatively small portion of total MOPs; they are less than 0.4% of total MOPs in all benchmarks except for *twolf*, where 1.1% of MOPs are measured to be harmful. This result is consistent with the data shown in Figure 7-8, where few *3-cycle+* dependence edges are observed (1.1% in *twolf* here is equivalent to 0.2% of *3-cycle+* edges in Figure 7-8). The percentage of not useful MOPs (fourth column) is higher; they account for 4.7% of MOPs on average across benchmarks (worst 7.7% in *gzip*). Note that such

MOPs can still be beneficial since they reduce issue queue contention.

Comparing this result to the *without filtering* case in the table, we find that our filtering mechanism is very effective in reducing not useful and harmful MOPs. There are significantly more not useful and harmful MOPs generated without the filtering mechanism. Especially, not useful MOPs in *gap* increase from 3.2% (with filtering) to 34.8% (without filtering), which reduces the efficiency of macro-op scheduling significantly. Remember that the no filtering case captures 9.1% more MOPs in *gap*. The difference between the two numbers (34.8% vs. 9.1%) implies that the filtering not only eliminates not useful MOPs but also helps the detection logic to find other alternative pairs that are useful.

The negative effects of not useful and harmful MOPs are reflected in performance. Figure 7-21 presents the performance of macro-op scheduling with and without filtering those MOPs. The left and middle bars in each benchmark are identical to Figure 7-7. The middle bars (*MOP with filtering*) represent macro-op scheduling with filtering. The right bars (*MOP without filtering*) show performance drops in most benchmarks compared with the filtering case, ranging from 0.02% in *vpr* to 7.41% in *gap*.

In summary, avoiding not useful and harmful MOPs is critical to ensure the benefits of macro-op scheduling. The filtering mechanism used in the base macro-op scheduling is effective in reducing their negative impact.

Table 7-7: Impact of filtering not useful and harmful MOPs (MOP-3src, no independent MOP).

Benchmarks	# MOPs with filtering / # MOPs without filtering	with filtering		without filtering	
		harmful MOPs / total MOPs	not useful MOPs / total MOPs	harmful MOPs / total MOPs	not useful MOPs / total MOPs
bzip	0.970	0.2%	6.9%	1.1%	10.9%
crafty	0.974	0.4%	6.8%	1.5%	13.3%
eon	0.997	0.0%	2.0%	0.5%	4.1%
gap	0.909	0.1%	3.2%	0.5%	34.8%
gcc	0.994	0.1%	3.8%	7.5%	15.0%
gzip	0.967	0.3%	7.7%	0.8%	17.7%
mcf	0.994	0.0%	3.8%	3.2%	9.1%
parser	0.986	0.4%	5.0%	2.7%	8.9%
perl	1.000	0.0%	3.5%	2.2%	13.1%
twolf	0.922	1.1%	7.1%	2.5%	13.5%
vortex	0.995	0.1%	7.3%	0.4%	10.6%
vpr	0.974	0.4%	3.6%	2.5%	8.4%

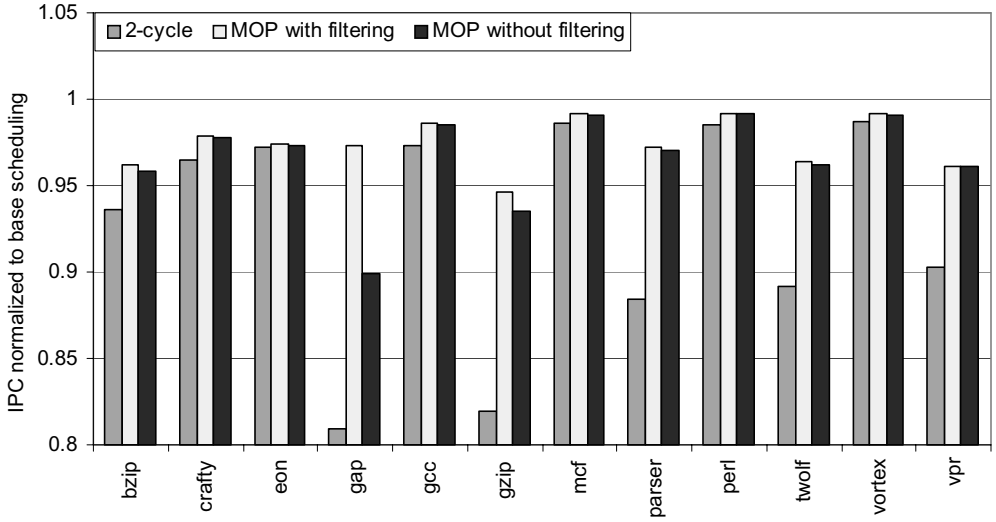


FIGURE 7-21. Performance impact of filtering not useful and harmful MOPs (MOP-3src, no independent MOP, 128-entry issue queue).

7.8.4 Cycle detection heuristics for register dependences

In Section 6.3.1, I discussed the problems with cycles induced by improper MOP grouping, and cycle detection heuristics to avoid tracking dependences through multiple levels of instructions. The cycle conditions falsely detected by the heuristics may degrade performance due to lost grouping opportunities. In order to measure how many grouping opportunities are lost due to the heuristics used in MOP detection logic, the simulator is configured to detect cycles precisely by tracking multiple levels of dependent instructions.

Table 7-8 compares cycle detection heuristics with precise detection in terms of the number of MOPs and IPC. The base machine has the 128-entry issue queue. To eliminate other secondary effects, macro-op grouping captures dependent MOPs only. In the table, the second and the third columns present the number of MOPs and IPCs normalized to the precise detection case. I find that the lost grouping opportunities due to the heuristic is negligible. The detection heuristic captures slightly more MOPs in some benchmarks but they are still within the noise. This result comes from the fact that 1) the vast majority of instructions have only one dependent instruction [12], 2) even though an instruction has multiple consumers, the MOP detection mechanism may not observe them between the two grouped instructions because of long dependence edge distances, 3) not many instructions have two source dependences [56], and 4) the filtering mechanism to avoid harmful grouping tends to avoid two-source instructions as MOP tails.

7.8.5 Cycle detection heuristics for memory dependences

The details of cycle avoidance heuristics for memory dependences on various macro-op scheduling and machine configurations were summarized in Table 6-1. Since the base machine supports partial store-to-load forwarding, MOP detection logic does not

check intervening store-to-load pairs to avoid cycles through memory dependence. If the base machine does not support partial store-to-load forwarding, those pairs should be avoided to prevent deadlock.

Table 7-9 presents the impact of disallowing intervening store-to-load pairs on MOP coverage and performance. In order to make the result consistent with other data, the base machine still allows partial store-to-load forwarding, since changing this potentially affects the base performance. Note that this heuristic is tested on a machine with partial store-to-load forwarding paths in order to ensure correct behavior.

From the table, I find that the impact of disallowing those pairs is negligible; less than 1% of MOPs are lost for this restriction. The worst case IPC degradation is measured to be 0.5% in *gap*. This result indicates that the effectiveness of macro-op scheduling does not significantly rely on the policies for store-to-load forwardings. Note that this result may not hold for register-starved ISAs like IA-32 [48].

Table 7-8: Performance of the cycle detection heuristic compared with precise detection.

Bench-marks	# MOPs with the heuristic / # MOPs with precise detection		IPC with the heuristic / IPC with precise detection	
	MOP-2src	MOP-3src	MOP-2src	MOP-3src
bzip	0.988	0.988	0.999	0.999
crafty	1.001	1.001	0.999	1.000
eon	0.999	0.999	1.000	1.000
gap	0.999	0.999	0.999	0.999
gcc	0.999	0.998	0.999	1.000
gzip	0.999	0.999	1.000	1.000
mcf	0.974	0.974	0.999	0.999
parser	1.000	0.999	1.000	1.000
perl	0.995	1.004	0.981	0.983
twolf	0.994	1.000	0.997	0.999
vortex	0.999	0.999	1.000	1.000
vpr	0.998	0.997	0.999	0.999

Table 7-9: Impact of store-to-load pair (MOP-3src, no independent MOP, 128-entry issue queue).

Benchmarks	# MOPs without SL pair / # MOPs in base case	IPC without SL pair / IPC in base case
bzip	0.994	1.000
crafty	1.000	1.000
eon	0.990	0.999
gap	0.996	0.995
gcc	0.999	1.000
gzip	0.994	1.000
mcf	0.991	1.000
parser	0.999	1.000
perl	1.000	1.000
twolf	1.000	1.000
vortex	1.000	1.000
vpr	0.999	1.000

A MOP pointer has a single control bit to represents a control flow change between MOP head and tail, and captures 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 is taken, the MOP is not created. To find the efficacy of the control bit, Table 7-10 shows the impact on the MOP detection efficiency and performance when no intervening branch or jump is allowed. The second and third columns show the percentage of MOPs that contain intervening taken or not taken control instructions. The fourth and fifth columns show the impact on the number of MOPs and IPC if no intervening control instruction is allowed. Note that the lost MOP opportunities in the fourth column tend to be lower than the sum of the second and third columns because MOP detection may find alternative pairs.

The results indicate that allowing intervening control instructions is critical in terms of the number of MOPs, although their performance impact is not that severe. The benefits of disallowing intervening control instructions are 1) the MOP detection and formation logic becomes simpler, and 2) branch misprediction recovery becomes simpler since the issue queue entries do not have to partially evict instructions grouped across the mispredicted branch. The effectiveness of macro-op scheduling does not significantly rely on grouping across control instructions. However, the performance impact may be higher if issue queue contention is a major performance bottleneck.

7.9 Summary and Conclusions

In this chapter, I evaluate the effectiveness of the proposed macro-op scheduling

Table 7-10: Impact of intervening branches (MOP-3src, no independent MOP, 128-entry issue queue).

Benchmarks	MOPs with intervening taken branches in base MOPsched	MOPs with intervening not taken branches in base MOPsched	# MOPs without branch / # MOPs in base MOPsched	IPC without branch / IPC in base MOPsched
bzip	1.4%	1.6%	0.974	0.997
crafty	2.1%	2.7%	0.979	0.999
eon	4.7%	7.0%	0.894	1.000
gap	3.2%	1.6%	0.962	0.990
gcc	1.2%	5.3%	0.958	0.999
gzip	0.5%	2.7%	1.001	0.998
mcf	11.0%	6.4%	0.832	0.999
parser	2.8%	3.0%	0.949	0.998
perl	1.4%	4.3%	0.958	1.000
twolf	1.4%	4.1%	0.950	0.999
vortex	1.4%	11.3%	0.877	0.999
vpr	0.6%	2.0%	0.975	0.993

and measure its sensitivity to various formation and detection policies. Given the policy of grouping two instructions within a 2-cycle scope, macro-op formation captures a significant portion of performance degrading instructions on 2-cycle scheduling, which are scheduled consecutively as if 1-cycle scheduling is performed. Without issue queue contention, macro-op scheduling recovers the performance drop of 2-cycle scheduling in many cases. Macro-op scheduling is effective in recovering performance of the benchmarks sensitive to 2-cycle scheduling since it enables consecutive issue of dependent instructions and reduces queue contention. I find that grouping independent instructions further improves the effectiveness of macro-op scheduling. Macro-op scheduling can outperform the base 1-cycle scheduling with the 32-entry issue queue due to these benefits. The restriction in the number of source operands does not significantly affects the effectiveness of macro-op scheduling. Macro-op scheduling that allows only two source operands in a MOP still achieves most benefits of the 3-source case, which implies that macro-op scheduling can be built without modifying the conventional CAM-style wakeup logic

I also examine several aspects of macro-op scheduling in terms of macro-op detection and formation policies. A 2-cycle scope that captures up to eight instructions achieves most benefits compared with other wider MOP scope. This result is consistent with the characterization data that show most data dependence edges that trigger instruction issue tend to be clustered within a small range of instructions. Regarding the pipelinability of MOP detection process, pipelining the process with 8- or 16-instruction intervals provides most of the benefit of fully pipelined detection logic. Also, the detection latency is not critical since MOP pointers are stored in the instruction cache and reused repeatedly. The negative effects of last-arriving operands in MOP tails may significantly degrade performance. The proposed filtering mechanism is effective in avoiding not useful and harmful MOPs, achieving significant performance gain over the no-filtering case. The effectiveness of macro-op scheduling is not particularly dependent on MOP detection and formation policies, nor potential restrictions in the underlying microarchitecture.

In conclusion, macro-op scheduling compensates for the performance loss due to 2-cycle scheduling by enabling consecutive issue of dependent instructions and widening the instruction window. The benefits of macro-op scheduling can be summarized as relaxing atomicity and scalability constraints of conventional instruction scheduling. The performance results indicate that pipelined, 2-cycle macro-op scheduling can achieve similar or better performance than atomic 1-cycle scheduling.

Chapter 8

Macro-op Execution

In Chapter 6, I proposed macro-op scheduling that processes instructions at a coarser level by grouping them into a single schedulable unit. This approach relaxes the atomicity and scalability constraints of conventional instruction scheduling. In Chapter 7, I showed that macro-op scheduling is a good approach to achieving pipelined instruction scheduling with similar or better performance than conventional scheduling.

One question that arises in macro-op scheduling is why macro-ops should be converted back to the original instructions immediately after they are issued. Of course, this is a benefit of macro-op scheduling since the technique does not require any major changes in the data path nor alter the way register values are communicated. However, it does not fully exploit the benefits of coarse-grained instruction processing. Macro-ops reduce the control overhead required for synchronizing the activities of individual instructions not only in instruction scheduling but also in complex out-of-order execution pipelines. Moreover, they potentially localize the register value communication among the grouped instructions.

In this chapter, I extend coarse-grained instruction processing to include the entire pipeline for out-of-order execution from instruction scheduling to execution stages. In addition to the benefits of macro-op scheduling, the proposed technique, *macro-op execution*, increases the effective machine bandwidth with similar or even less complexity in instruction scheduler, dispatch, register file access and bypass logic.

This chapter is laid out as follows: Section 8.1 discusses an overview of macro-op

execution and highlights its benefits. Section 8.2 details the key components and microarchitectural modifications for macro-op execution. The experimental evaluation of macro-op execution will be presented later in Chapter 9.

8.1 Increasing Machine Bandwidth via Macro-op Execution

8.1.1 Limitations of macro-op scheduling

In macro-op scheduling discussed in Chapter 6, a MOP that contains two single-cycle operations is handled as a non-pipelined, 2-cycle operation from the scheduler's perspective. Instructions are sequenced by the dual-entry payload RAM in the dispatch stage, which sends the two original instructions in two consecutive clock cycles. Since macro-op scheduling enables only the scheduler to operate at a MOP granularity, the benefits of coarse-grained instruction processing are limited to instruction scheduling. However, there are several observations to make about the way instructions behave in macro-op scheduling, which can potentially be exploited to achieve further benefits. First, a MOP forces a predetermined execution order among the grouped instructions. This deterministic execution implies that the way values are bypassed becomes simpler and therefore some datapaths for value communication may become unnecessary. Second, due to the attribute of instruction sequencing, the two original instructions flow in the same path with a delay of one clock cycle, which can be modified to force the two instructions to be managed together within a single instruction slot in one execution lane. Third, macro-op scheduling blocks the issue slot for one clock cycle to create the time window for sequencing instructions. These idle cycles in the select logic can potentially be utilized to achieve wider issue bandwidth.

Macro-op execution extends the range of coarse-grained processing to the whole out-of-order portion of the pipeline, including scheduling, dispatch, register access and bypass logic in the execution stage. Instead of sequencing the original instructions in the dispatch stage, macro-op execution processes MOPs until they reach the execution stage. Here, the functional units and bypass paths are configured in such a way that the original instructions are naturally sequenced as MOPs move through the datapath. In addition to the benefits of macro-op scheduling (i.e. larger instruction window and pipelined scheduling logic), moving the sequencing point further down to the execution stage enables narrowing the issue, dispatch and register access bandwidth while sustaining equivalent execution bandwidth. Moreover, the bypass logic complexity, which may be one of the major bottlenecks in current and future-generation microprocessors, can also be reduced by exploiting the nature of macro-op scheduling. Conversely, in a wider machine configuration, macro-op execution can improve execution bandwidth without significantly increasing hardware complexity.

Figure 8-1 illustrates macro-op execution and its corresponding pipeline stages. The basic operations to group instructions into MOPs are similar to those of macro-op scheduling; 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, a MOP is created by allocating a single issue queue entry to those instructions. In the scheduling stage, wakeup and select operations can be pipelined because MOPs have multi-cycle execution latencies.

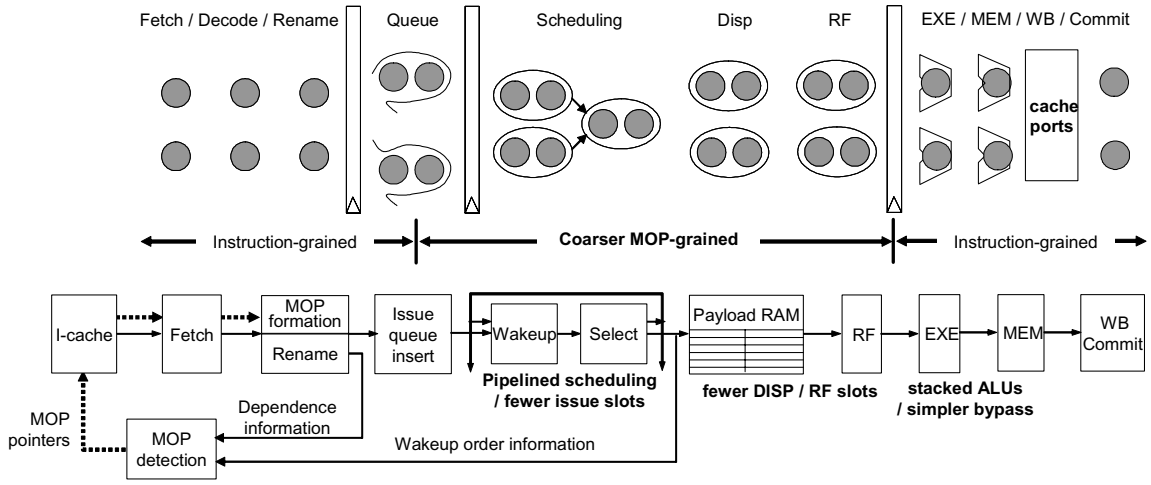


FIGURE 8-1. An overview of macro-op execution.

When all source operands of a MOP become ready, a MOP is issued as a single unit to the execution lane. Until this point, macro-op execution shares most of the key components and operations required for macro-op scheduling.

In the dispatch stage, each entry in the payload RAM has multiple fields for the original instructions grouped in a MOP. The actual register identifiers and opcodes of the original instructions are accessed in parallel using a single port. In the register access stage, the MOP accesses the register read ports for source values of the original instructions in parallel. Instruction sequencing occurs in the execution stage, which has stacked functional units that execute two original instructions sequentially. If a MOP contains load operations, the computed effective addresses are sent to data cache ports. Result values of two original instructions are individually written back to the register file. Back-to-back execution of two dependent instructions can occur either within a MOP or after the MOP tail is executed, which guarantees that no other instruction dependent on a MOP head is issued consecutively. Due to this attribute of macro-op scheduling, the bypass paths can also be simplified, since not all output ports of functional units need to be connected to all

of their input ports.

As in the macro-op scheduling case, the reorder buffer commits ungrouped original instructions separately in program order. Therefore, macro-op execution maintains correct architectural state even if a branch misprediction or even an exception occurs between two grouped instructions.

8.2 Microarchitecture for Macro-op Execution

In the previous section, I discussed the benefits of processing instructions at a coarser granularity throughout the entire execution pipeline and presented the principles of macro-op execution. In order to simplify the discussion, I set as the primary goal the ability to reap the IPC of a conventional 4-wide machine with atomic scheduling by using a machine with 2-instruction issue bandwidth and pipelined (2-cycle) instruction scheduling.

This section describes MOP grouping policy, the details of macro-op execution, and the proposed microarchitecture. Section 8.2.1 to Section 8.2.4 discuss the MOP grouping policy for macro-op execution. Note that the details of MOP detection and formation logic, which are the key components for grouping instructions, have been discussed in Section 6.3 and Section 6.4, so I will focus on describing the modifications required to support macro-op execution. Section 8.2.5 to Section 8.2.9 present how the pipeline stages from scheduling to execution process macro-ops, and highlight the benefits of macro-op execution. Section 8.2.10 estimates the effective machine bandwidth increased by macro-op execution.

8.2.1 Macro-op grouping policy

In order to enable consecutive execution of dependent instructions in the pipelined scheduling logic, and to increase issue bandwidth, macro-op detection and formation targets single-cycle operations: single-cycle ALU, control (e.g. branch), and load and store address operations. Since load instructions are not explicitly cracked into two separate operations (address generation and actual cache port access) on the base machine model described in Section 3.1.1, I will simply classify loads as groupable candidate instructions. Other types such as integer multiply or floating-point operations are not grouped into MOPs, and will require separate functional units.

The policy for macro-op execution is to group two directly dependent instructions into a dependent MOP, or two independent instructions with identical source operands into an independent MOP. Although bigger MOP sizes may enable the scheduling loop to span over more clock cycles and further increase the effective machine bandwidth, the characterized groupability of instructions in Section 4.5 indicates that not many MOPs capture more than two instructions in general. A bigger problem is that irregular MOP sizes incur low resource utilization. Unlike macro-op scheduling, macro-op execution requires modifications in the datapath and functional units, potentially placing more resources in each execution lane, although the total number of execution lanes can be reduced. Since each execution lane must satisfy the maximum resource requirement of a MOP, increasing the MOP size will result in wasting many resources such as functional units or register ports. Also, the bypass network complexity should be considered for bigger MOP sizes. Hence, this thesis studies the potential for 2x MOPs.

Due to the restrictions in the datapath in the execution stage (which will be discussed in Section 8.2.8), the possible combinations of grouped instructions are two depen-

dent or independent single-cycle operations (*S*-*S*), two independent loads *(L*-*L*), and a single-cycle operation and a dependent load (*S*-*L*). Note that independent MOPs for an S-L pair is also possible, but were not considered for macro-op execution because initial experiments determined that the number of such MOPs is not significant and that they tend to degrade performance when load issue is blocked by unresolved earlier store address operations. Also note that a machine with separate schedulers for different types of instructions would not be able to process S-L MOPs. However, the experimental results in Chapter 9 will show that this inability does not severely reduce the effectiveness of macro-op execution.

8.2.2 Macro-op 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 and possible cycle conditions, and to generate MOP pointers that represent MOP pairs. Generated MOP pointers are stored in the first-level instruction cache along with instruction words, and later direct the MOP formation process. The basic operations of MOP detection for macro-op execution are identical to those of macro-op scheduling. A few modifications are required to support macro-op execution. The detection logic should be able to capture S-L and L-L pairs, which were not allowed in the base macro-op scheduling. Including loads as candidate instructions does not fundamentally alter the detection process, although it requires detection of cycle conditions through memory dependences (this will be discussed in the following section). The added complexity to the detection logic should not significantly affect cycle time since the logic can be pipelined, as described in Section 6.3.4 and Section 7.8.1.

8.2.3 Cycle detection

Section 6.3.1 and Section 6.3.2 discussed the cycle conditions induced by improper MOP grouping, and avoidance heuristics that do not require multiple levels of tracking register and memory dependences. Since macro-op execution allows grouping load instructions as MOP tails, the cycle detection heuristics for memory dependences described in Section 7.8.5 will be used together with the one for register dependences. Therefore, L-L and S-L MOP pairs are not created across intervening store instructions to avoid cycles through memory dependences.

8.2.4 Macro-op formation

The basic operation of MOP formation for macro-op execution is identical to macro-op scheduling. 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 then inserted into a single issue entry in the queue stage, creating a MOP in the scheduler.

An important modification required for macro-op execution is the dependence tracking process described in Section 4.3. The base macro-op scheduling groups only two single-cycle instructions to match the scheduling latency of the pipelined 2-cycle scheduling logic. This attribute forces the instruction scheduler to use MOP latency tracking for register dependences, since the MOP offset tracking becomes meaningless. In macro-op execution, S-S pairs have the same execution latency (i.e. two clock cycles) as the scheduling loop so the MOP latency tracking approach can process them without incurring extra delay. However, L-L and S-L pairs need special consideration because their execution latency is greater (four clock cycles) than the scheduling latency (two clock cycles). MOP

latency tracking may unnecessarily delay other instructions dependent on the MOP head until after the two instructions grouped in a MOP completes, which incurs one or two clock cycles of extra delay for L-L or S-L MOPs, respectively.

To avoid this penalty, the MOP formation logic is configured to support MOP offset tracking. An extra field is added to each entry in the MOP translation table discussed in Section 6.4.2; when MOP IDs are assigned to the instruction during the dependence translation and stored into the MOP translation table, each instruction marks a single-bit extra field, indicating whether it is grouped as a MOP head or tail, so that dependent instructions that access the MOP translation table know whether they are dependent on which instruction in the MOP. If the map table for register renaming already keeps track of the execution latency for the parent instructions on the base microarchitecture, storing extra information of the execution latencies into the MOP translation table is not necessary. When an instruction or a MOP is inserted into the issue queue entry, it adjusts the execution latency of the parent instructions corresponding to source operands, based on the extra field in the MOP translation table, and sets an appropriate execution latency in the timer logic that tracks the readiness of source operands.

An example of MOP offset tracking is illustrated in Figure 8-2. There are four instructions *I1* ~ *I4*, among which MOP detection logic created a S-L MOP of *I1* and *I3*. When the two instructions are registered to the translation table, the instruction *I1* sets the *MOP head* bit, indicating that the instruction is grouped as the MOP head. The instruction *I3* does not set the bit, indicating it is a MOP tail. At the same time, their execution latencies are also stored into *EXE lat* fields (assuming that the MOP translation table also tracks execution latencies). The dependent instructions *I2* and *I4* access to the MOP trans-

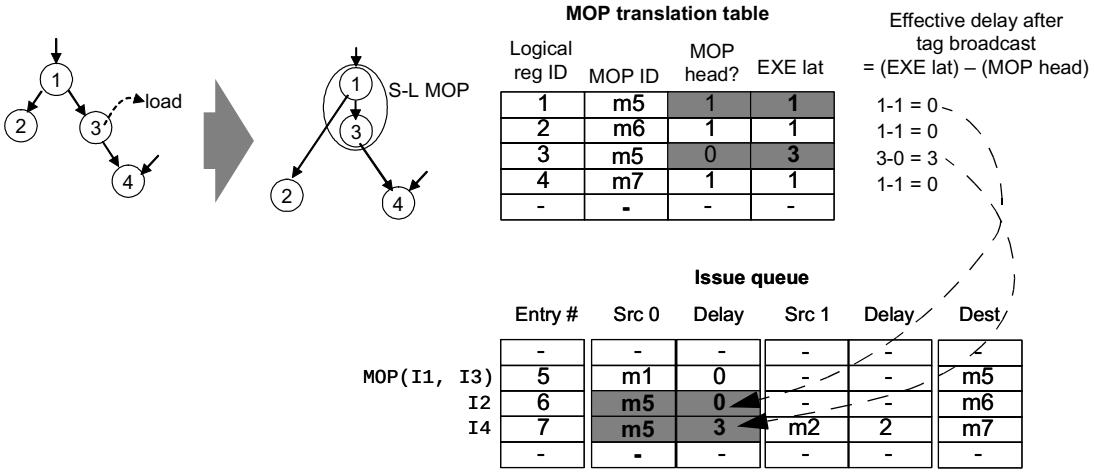


FIGURE 8-2. MOP offset tracking for macro-op execution.

lation table and acquire the same MOP ID of *m5* but two different execution latencies. Since the MOP scheduler performs pipelined 2-cycle scheduling, adjustments in the execution latencies are required to correctly track the readiness of source operands. The effective delay for a source operand grouped as a MOP head is one clock cycle less than its execution latency, since broadcasting a tag takes one clock cycle. However, if the source operand is an instruction grouped as a MOP tail, this adjustment is unnecessary because the effective issue timing of this instruction will be same as the clock cycle when its tag is broadcast. Therefore, the effective delay for source operand tracking becomes the difference between two values in *EXE lat* and *MOP head* fields. Note that this type of delay adjustment is already required for conventional pipelined instruction schedulers to reflect their scheduling latencies to the execution latencies.

After instructions are inserted into the issue queue, instructions *I2* and *I4* will observe the tag broadcast of *m5* from MOP (*I1*, *I3*). However, they will interpret the same tag differently due to the two different delay values (zero and three clock cycles for the head and tail, respectively), and correctly track the readiness of the corresponding source

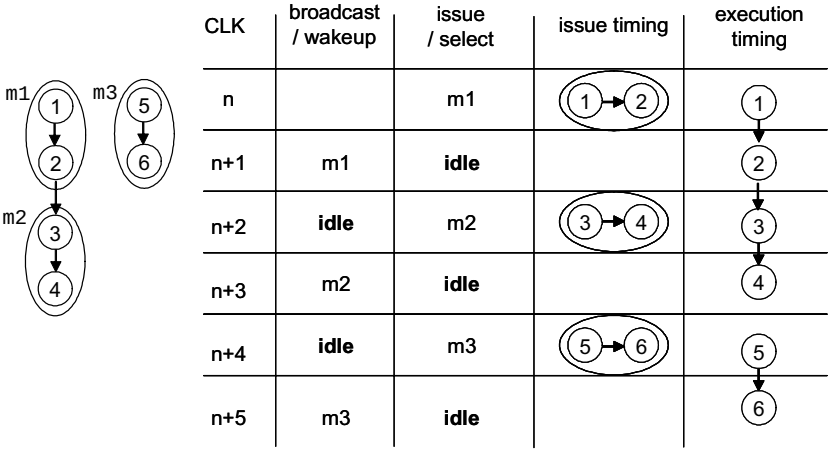
8.2.5 Instruction scheduling logic

In macro-op scheduling, two instructions are sequenced by the payload RAM that sends two original instructions within two consecutive clock cycles, and during which the issue slot is blocked in order to prevent other instructions from being sent to the same execution lane. To support this, a MOP is handled as a multi-cycle latency, non-pipelined operation from the scheduler's perspective. This implies that issue bandwidth is wasted because the select logic is idle during the sequencing operation. Macro-op execution exploits these unnecessary bubbles in the select logic to increase issue bandwidth by handling a MOP as a pipelined operation and moving the point of sequencing instructions further down to the EXE stage.

The scheduling and execution timings are illustrated in Figure 8-3. In this example, we assume that the scheduler has only one issue slot and therefore no other operations can be issued in parallel. In the base macro-op scheduling case (Figure 8-3a), the select logic is blocked after issuing a macro-op in order to provide a time window for the payload RAM to sequence the original instructions. In the scheduler for macro-op execution (Figure 8-3b), the select logic does not have idle cycles and can issue a MOP every clock cycle, which effectively doubles the issue bandwidth. We note that this scheduling logic still guarantees that MOP *m2* (instructions 3 and 4) is correctly scheduled two cycles after MOP *m1* (instructions 1 and 2) is issued since a MOP is handled as a pipelined multi-cycle operation from the scheduler's perspective.

One complication in the select logic is the handling of load instructions. If the MOP tail instruction is a load, the select logic should be able to avoid possible cache port

(a) base macro-op scheduling



(b) macro-op execution

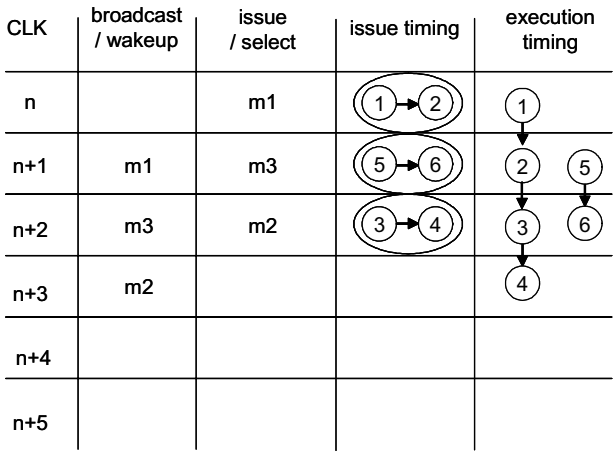


FIGURE 8-3. An example of scheduling timing for macro-op execution.

conflicts with other loads or MOPs issued in the following clock cycle. This simply needs the select logic to keep track of the load port uses in the previous clock cycle for correct resource management. Although this may complicate the select logic, I believe that its impact on the cycle time is not significant, especially considering that the scheduling logic is pipelined and the select logic can use the full clock cycle period.

8.2.6 Dispatch / Payload RAM Access

An issued MOP accesses the payload RAM and acquires the original register spec-

ifiers, opcodes, and other necessary information needed for execution. Each payload RAM line has two entries for two grouped instructions. Although this configuration will double the number of bits to be accessed by a single request, two instructions grouped in a macro-op can use only a single port, reducing the number of payload RAM read ports compared to a case with the same effective issue bandwidth. For example, a 2-wide machine with macro-op execution has two payload RAM read ports that support up to four original instructions accessed in parallel. Note that macro-op execution does not sequence the original instructions in this stage, while macro-op scheduling uses the dispatch stage to sequence instructions. The two original instructions grouped in a MOP are dispatched together to an appropriate execution lane.

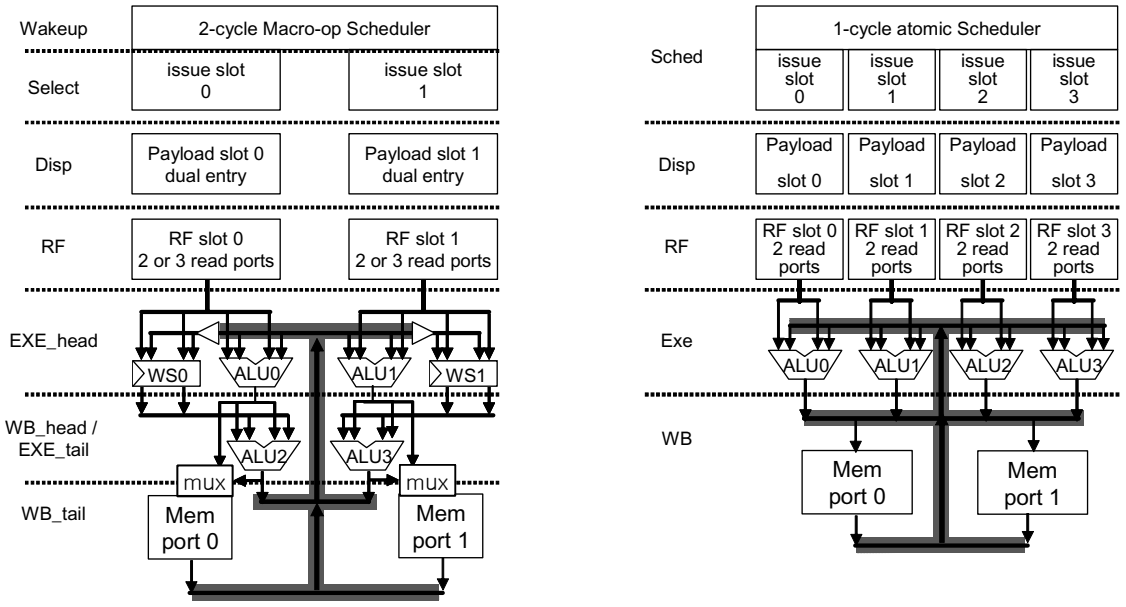
Payload RAM entries are allocated when instructions are inserted into the issue queue in the queue stage. Since the original instructions are inserted individually at the original instruction granularity while the execution core processes instructions at a MOP granularity, macro-op execution needs an equivalent number of write ports to the payload RAM. The number of write ports is determined by the insertion bandwidth (e.g. four instructions per cycle on the base machine) of the queue stage. The write ports can potentially be reduced if the coarse-granular instruction processing is fully extended to the front-end of the pipeline, i.e. fetch, decode and rename stages, but this optimization is not considered for this thesis research, since this may require a complex algorithms to reorder instructions and place groupable instruction pairs together while maintaining the sequential semantics of the original program. With support from a software-based dynamic translator that places *fused* instructions together [46], rename and insertion bandwidth can also be reduced, along with the write ports to the payload RAM.

8.2.7 Register File

A MOP accesses the register file for the source operands of the original instructions in parallel. The maximum number of source operands is three when two dependent instructions are grouped, assuming that an instruction has up to two source operands in this architecture. Although each MOP may need three register ports for source values, the MOP detection logic can prevent 3-source MOPs from being created and hence it does not necessarily increase the number of read ports needed for each execution lane. I showed in Section 7.3 that not many MOPs have three source operands and restricting them to only two sources does not significantly degrade the MOP coverage. This is also beneficial for CAM-style wakeup logic, which needs to support up to three source operands of 3-source MOPs. However, in either case, the total number of register read ports are reduced compared to the base case with the same effective execution bandwidth since fewer instructions (or MOPs) are processed simultaneously in this stage. Macro-op execution better utilizes register read ports by implicitly specifying that the result value of the MOP head will be read off the bypass path by the MOP tail.

Note that macro-op execution does not improve write port utilization and will require the same number of write ports as the base case, since this is a function of execution bandwidth and result values are written back to the register file individually by each instruction. Macro-op execution can potentially be extended to reduce write port requirements by performing liveness analysis of register communications or predicting and tracking the value degree of use [12], but these optimizations were not considered for this thesis because of the complexity required for implementing them in hardware.

8.2.8 Execution stages and bypass network



(a) 2-wide macro-op execution machine (b) conventional 4-wide machine

FIGURE 8-4. Execution pipeline and datapath for macro-op execution.

Figure 8-4a illustrates functional units and bypass paths in the execution stage for macro-op execution. When a MOP reaches the first execution stage (*EXE_head*), the MOP head starts execution. At same time, the *waiting station* (*WS0* and *WS1* in the figure) latches the MOP tail and its source values acquired from the register file. After the MOP head finishes execution, the result value is written back to the register file and also forwarded to the input ports of the stacked ALU in the *EXE_tail* stage. Since this forwarding occurs within a MOP (between the MOP head and tail) and no other instruction dependent on the MOP head (S-S and S-L MOP cases) is scheduled consecutively, it does not require full bypass paths across all functional units and therefore can be simply implemented by connecting two ALUs. Although instructions dependent on the MOP head of an L-L MOP may be scheduled consecutively, the output of the MOP head (a load instruction) becomes available at the cache ports, which have full bypass paths to the functional units ALU0 and ALU1 so this case is correctly handled. The waiting station sends the MOP tail to the

EXE_tail stage and the MOP tail starts its execution. After the MOP tail is executed, its result value is written back to the register file and is forwarded to two other functional units (ALU0 and ALU1) in the EXE_head stage, enabling consecutive execution of dependent instructions.

Compared to macro-op scheduling, macro-op execution does not degrade performance (ignoring other secondary effects incurred by narrowing the machine bandwidth or different grouping policies) due to the lost opportunities for bypassing or forwarding register values through these restrictive data paths. Rather, macro-op scheduling wastes bypass bandwidth even though the restrictive scheduling behavior created by MOPs that forces the deterministic execution prevents the bypass paths from being fully utilized. In a sense, the macro-op execution model can be interpreted as vertically clustering (or horizontally slicing across dependence chains) the datapaths and bypass network to improve their complexity, whereas other proposals for clustered microarchitecture [78][74][30] horizontally cluster the machine along with the chains of dependent instructions. In fact, the both horizontal and vertical clusterings are complementary and can potentially be implemented in combination without significantly interfering each other.

In Figure 8-4, the timing-critical bypass paths in the macro-op execution and the base 4-wide machine are highlighted. The complexity of the bypass paths for macro-op execution (Figure 8-4a) is similar to the conventional 2-wide machine case, in which the output ports of two ALUs are connected to the input ports of two ALUs. Note that the bypass paths in the actual circuit layout do not necessarily expand across two functional units vertically (as shown in the figure). Macro-op execution enables more routing flexibility due to the fewer number of input and output ports to be connected, and potentially

shortens the worst-case path. In contrast, the conventional 4-wide machine with full bypass network (Figure 8-4b) needs to connect them across all input and output ports. Also note that MOP tails may need to acquire source values from the critical bypass path, although the instruction scheduling guarantees that MOP tails and their parent instructions outside the MOPs are not scheduled consecutively. This is because MOP tails access the register file two cycles before execution, creating a two-cycle bypass window. Since MOP tails in the waiting station do not perform any useful operations, this forwarding path can be routed further using a full clock period without severely affecting the bypass logic delay (e.g. through bus repeaters).

8.2.9 Resources and Execution Timings

Table 8-1 presents resources and effective execution timings for different types of instructions and MOPs. “S” represents a single-cycle candidate instruction. “L” represents a load instruction that is composed of an address generation and a cache port access operation. All types of instructions and MOPs are assumed to execute in the left execution lane in the Figure 8-4a. When they are executed in the right execution lane in the figure, the corresponding resources are ALU1, ALU3, MEM1 and WS1.

Macro-op execution achieves maximum benefit when two dependent instructions are grouped because this case reduces the issue queue contention, increases issue bandwidth, and enables consecutive execution of two dependent instructions, all at the same time. In the case of independent MOPs in which two independent instructions are grouped, it may negatively affect performance by serializing execution of two instructions; these may have executed in parallel through two different issue slots if they are multi-cycle operations like loads. In contrast, independent MOPs of single-cycle instruc-

Table 8-1: Resources and timings in macro-op execution.

Instruction / Macro-op types	S	L	S-S MOP	S-L MOP	L-L MOP	Others
Resources & execution timing (for the left execution slot)	ALU0	ALU0 MEM0	ALU0 WS0 ALU2	ALU0 WS0 ALU2 MEM0	ALU0 WS0 MEM0 ALU2 MEM0	dedicated functional units

tions do not affect performance negatively compared with the 2-cycle scheduling case, in terms of execution timings of their dependent instructions, as described in Section 6.7.1. Aggressive grouping of independent instructions may degrade performance for this reason, if issue bandwidth is not a major performance bottleneck. However, in cases where limited issue bandwidth is a bottleneck, performance is improved as more instructions are grouped, since the benefits of increased machine bandwidth can compensate for their negative effects on data dependences.

Table 8-2 estimates and compares hardware complexity of conventional and macro-op execution machines. All machine configurations have the same machine bandwidth of four instructions in fetch, decode, rename, and instruction commit. The machine with macro-op execution is configured to have an effective execution bandwidth of four instructions per cycle when two MOPs are issued in parallel, followed by another two instructions in the following clock cycle. In the table, the complexity of macro-op execution (with 2-source MOPs) is similar to a conventional 2-wide machine case with 1-cycle scheduling, except that more functional units and register write ports are required. From the perspective of instruction scheduling logic complexity, macro-op execution is similar to the conventional 2-wide machine with 2-cycle scheduling, which has a scheduler pipelined over two pipeline stages.

.

Table 8-2: Comparison of hardware complexity in execution pipeline.

HW component	4-wide-1-cycle	2-wide-1-cycle	2-wide-2-cycle	2-wide-MOP
Fetch / decode / commit	4 instructions per cycle			
Scheduler	atomic 1-cycle scheduling, 4 issues / cycle	atomic 1-cycle scheduling, 2 issues / cycle	pipelined 2-cycle schedul-ing, 2 issues / cycle	pipelined 2-cycle schedul-ing, 2 issues / cycle
Dispatch / payload RAM	4 read ports, 4 write ports	2 read ports, 4 write ports	2 read ports, 4 write ports	2 read ports, 4 write ports, each entry has 2 fields
Register file (for 1-cycle ALUs and load ports)	8 read ports, 6 write ports (4 for ALUs, 2 for loads)	4 read ports, 4 write ports (2 for ALUs, 2 for loads)	4 read ports, 4 write ports (2 for ALUs, 2 for loads)	4 (2-src) or 6 (3-src) read ports, 6 write ports (4 for ALUs, 2 for loads)
Bypass network (for 1-cycle ALUs and load ports)	6 outputs (4 for ALUs, 2 for loads) con-nected to inputs of 4 ALUs	4 outputs (2 for ALUs, 2 for loads) con-nected to inputs of 2 ALUs	2 outputs (for loads) con-nected to inputs of 2 ALUs	4 outputs (2 for ALUs in EXE_tail stage, 2 for loads) connected to inputs of 2 ALUs (EXE_head stage), local bypass paths between EXE_head and EXE_tail stages

A machine with macro-op execution can potentially be configured to have wider execution bandwidth with the same number of execution lanes, i.e. 8-wide execution bandwidth using four execution lanes with two functional units each. In order to evaluate the potentials for macro-op execution later in Chapter 9, macro-op execution will be tested with various execution bandwidths.

8.2.10 Effective issue and execution bandwidth for macro-op execution

Macro-op execution increases issue and execution bandwidth by processing multi-ple instructions as a single unit. Theoretical issue and execution bandwidth for macro-op execution is based on can be calculated using the following equation, which is similar to *Amdahl's law* [1]:

$$\text{Effective BW} = \frac{(\text{issue BW})}{1 - (\text{MOP coverage}) + \frac{(\text{MOP coverage})}{(\text{MOP size})}}$$

MOP coverage is the fraction of instructions grouped into MOPs for macro-op execution. *Issue BW* is the issue bandwidth of the base machine. Since the current policy groups two instructions in a MOP, *MOP size* is two for our case.

Note that the machine can achieve this theoretical issue and execution bandwidth when there are plenty of instructions to fill all the issue slots every clock cycle. In fact, the actual issue and execution bandwidth may be lower, because a wider execution bandwidth is achieved only when MOPs are issued followed by other instructions in the next clock cycle. On the 2-wide macro-op execution machine, for example, when two MOPs are issued and no other instructions are issued in the following clock cycle, the effective issue bandwidth is four and zero (for each cycle) and the effective execution bandwidth is two and two, losing the benefits of macro-op execution. Therefore, this equation should be interpreted as the upper limit of the macro-op execution.

8.3 Summary and Conclusions

The principles of macro-op execution and the details of microarchitectural support for enabling macro-op execution are discussed in this chapter. Macro-op execution extends coarse-grained instruction processing to the entire execution pipeline including instruction scheduling, dispatch, payload RAM and register file accesses, and execution stages. A key difference between macro-op scheduling and execution is where instructions are sequenced. The proposed macro-op execution moves the instruction sequencing point down to the execution stage and enables the original instructions grouped in MOPs to be

naturally sequenced as they flow through the datapath. Combined with the benefits of relaxed atomicity and scalability constraints of conventional instruction scheduling achieved by macro-op scheduling, the proposed macro-op execution increases the effective machine bandwidth using similar or even lower complexity hardware. Macro-op execution shares many key components to create MOPs with macro-op scheduling, such as MOP detection and formation logic. Some modifications to support macro-op execution are also described. For grouping loads instructions, MOP detection logic needs to detect cycle conditions through memory dependences. MOP formation logic also needs some modifications for MOP offset tracking. Macro-op execution enables narrowing the processing bandwidth (especially for read accesses) of many structures and datapaths. The number of issue slots in the scheduler can be reduced by issuing multiple instructions as a single MOP without sequencing them, and exploiting idle cycles in select logic. The read ports to the payload RAM and register file can be better utilized by processing multiple requests in groups. Bypass logic can be simplified by exploiting the attribute of macro-ops that forces predetermined and restrictive execution of instructions grouped, compared to the conventional case with a full bypass network and the same effective execution bandwidth.

Chapter 9

Experimental Evaluation of Macro-op Execution

Several aspects of macro-op execution are examined. The description of the simulator along with the parameters used for the base machine was described in Section 3.2.2 and Section 3.2.4. The necessary microarchitectural modifications for macro-op execution were described in Section 8.2. Since macro-op execution builds on macro-op scheduling and shares many mechanisms, this chapter will focus on evaluating its performance potential, as opposed to its sensitivity to various MOP detection and formation configurations, which were extensively discussed in Chapter 7.

9.1 Machine Configurations

To measure the effectiveness of macro-op execution, I simulate several machine models. In order to simply the discussion, I will refer to a machine with N issue bandwidth and M-cycle scheduling logic as *N*-wide-*M-cycle*. *4-wide-1-cycle* is the base machine (detailed in Section 3.2.4) and all performance data are normalized to this case, unless specified otherwise. *2-wide-1-cycle* has the same configuration as the base machine except that the issue bandwidth is halved and the execution pipelines are configured accordingly. Although this machine has the same number of functional units and memory ports in order to keep all other configurations the same, only two instructions can start execution in parallel each clock cycle and hence it cannot fully utilize its execution resources. *2-wide-2-cycle* has the same configuration as the 2-wide-1-cycle case except that it has pipelined scheduling logic with two separate wakeup and select stages. Similarly, I also test *3-wide-*

1-cycle and *3-wide-2-cycle* machines with the issue bandwidth of three instructions. Macro-op execution is built on top of the N-wide-2-cycle machine: *N-wide-MOP-2src* and *N-wide-MOP-3src*, which support 2-source and 3-source MOPs, respectively. In the 2-wide-MOP-2src and 2-wide-MOP-3src cases, the number of functional units and memory ports are the same as the base case. Other macro-op execution configurations have more functional units than the base machine, which are configured accordingly based on Figure 8-4 (two simple ALUs per execution lane). Other complex integer, FP, and memory ports are the same as the base case. 2-wide macro-op execution can fully utilize 4-wide execution bandwidth only when two instructions or MOPs are issued following two MOPs issued in the previous clock cycle. 3- and 4-wide macro-op execution can execute up to six and eight instructions per cycle, as long as plenty of MOPs are issued. The detailed execution resource configuration and bypass paths were presented in Section 8.2.5 through Section 8.2.8. The base MOP detection and formation has a 2-cycle scope, which captures MOPs within up to eight instructions in program order on the 4-wide machine configuration. The details on their operation and structure were discussed in Section 8.2.2 through Section 8.2.4. MOP detection logic is fully pipelined, and has a latency of three clock cycles from examining register dependences to generating MOP pointers. In later sections, I will also evaluate other various configurations of macro-op execution. The IPCs of N-wide-M-cycle machines were presented in Table 3-4 and Table 3-5.

9.2 Microbenchmark Results

To ensure the correct behavior of macro-op execution implemented on the timing simulator and demonstrate its potential, several experiments similar to Section 7.2 are per-

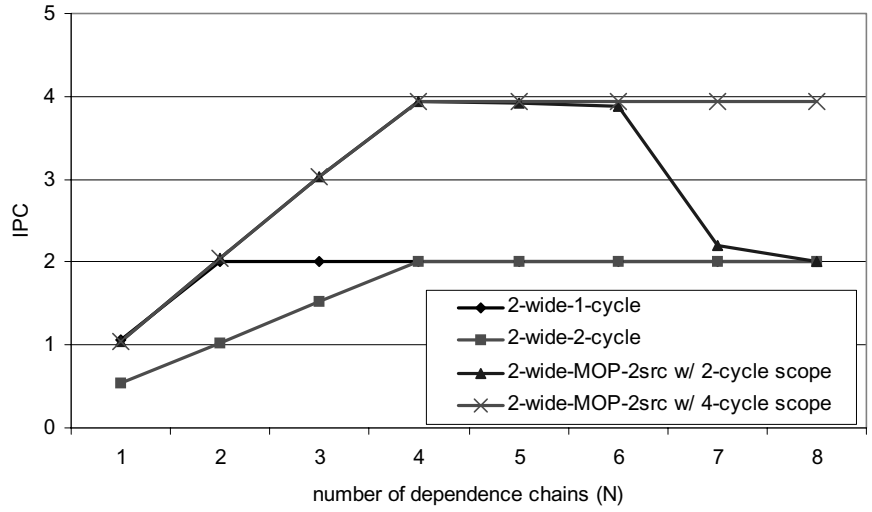


FIGURE 9-1. Performance of macro-op execution with microbenchmarks.

formed. The basic code structure of the microbenchmark was illustrated in Figure 7-1. The details on compiling microbenchmarks were explained in Section 3.2.3. For controlled experiments, the base machine is configured to have a perfect branch predictor and perfect memory. The base machine still fetches and commits four instructions per cycle but its issue bandwidth is narrowed down to two instructions or two MOPs per cycle.

Figure 9-1 presents the IPCs measured on 2-wide-1-cycle, 2-wide-2-cycle and 2-wide-MOP-2src machines when the number of dependence chains (N) varies from one to eight. The performance curves in both 1-cycle and 2-cycle scheduling cases saturate at an IPC of two because of the issue bandwidth of two instructions per cycle. The IPC for 2-cycle scheduling is lower than that of 1-cycle scheduling because scheduling bubbles are created and it cannot issue and execute dependent instructions consecutively. To achieve an equivalent performance to 1-cycle scheduling, 2-cycle scheduling needs twice many independent instructions to completely fill the scheduling bubbles.

Macro-op execution achieves the full execution bandwidth of four instructions per

cycle using a pipelined 2-cycle scheduler and a 2-wide issue bandwidth. This trend is sustained until N becomes six. The 2-cycle MOP scope case (*2-wide-MOP-2src w/ 2-cycle scope*) significantly loses performance when N is seven or eight, since the MOP detection process cannot detect long-distance dependence edges over seven or eight instructions and loses grouping opportunities. This result was also shown in Figure 7-3 in conjunction with macro-op scheduling. In contrast, the 4-cycle MOP scope case (*2-wide-MOP-2src w/ 4-cycle scope*) does not experience this difficulty and therefore fully achieve an IPC near four instructions per cycle when N is seven or eight. Compared with Figure 7-2, its performance curve (4-cycle scope) is almost identical to the 1-cycle scheduling performance with a full, 4-wide issue bandwidth, which is the expected behavior that the proposed microarchitecture aims to achieve.

It should be noted that a wider macro-op execution (i.e. *3-wide* or *4-wide-MOP-2src*) will achieve the same performance curve as presented here, and does not further improve performance. This is because the fetch and commit bandwidth is fixed to four instructions per cycle and wider issue width cannot be utilized. However, with real benchmark programs, wider macro-op execution machines may benefit from the increased issue bandwidth to some extents since the actual IPC is far lower than its maximum, i.e. IPC of four, although the degree of benefits will be still limited by the fetch and commit bottlenecks.

In the following sections, macro-op execution will be evaluated using SPEC2K benchmarks.

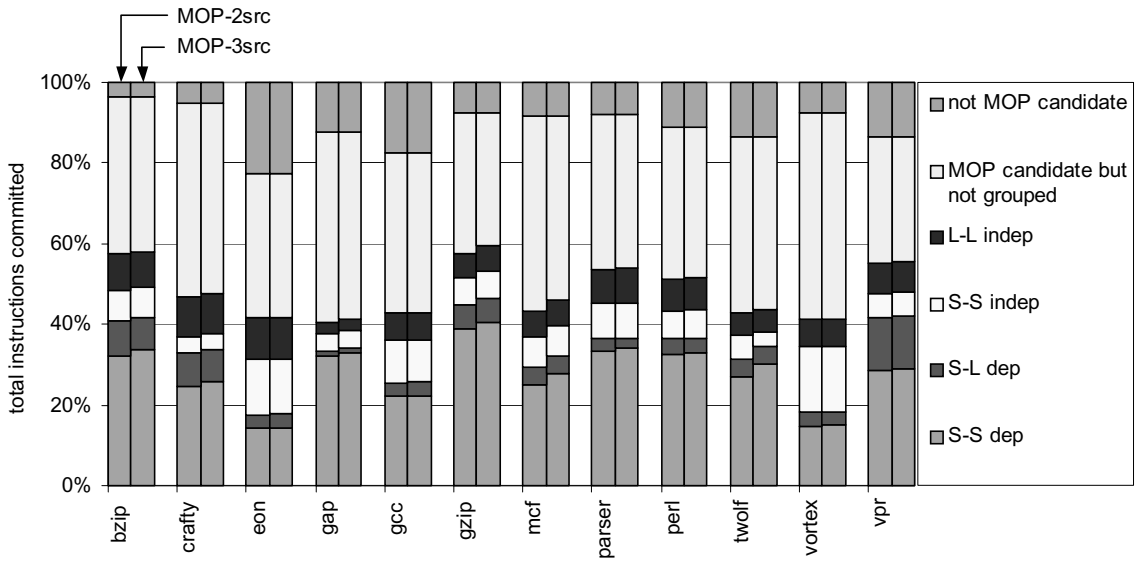


FIGURE 9-2. Instructions grouped in MOPs for macro-op execution.

Figure 9-2 shows the percentage of grouped instructions for macro-op execution. Each benchmark has two bars: MOP with two sources (*MOP-2src*) and three sources (*MOP-3src*) cases. The y-axis shows the total dynamic instructions committed. *Not MOP candidate* and *MOP candidate but not grouped* bars are self-explanatory. Comparing with the macro-op scheduling case shown in Figure 7-6, the percentage of candidate instructions is higher since macro-op execution includes loads as groupable candidates. If an instruction is not grouped and is a single-cycle operation, its dependent instructions will not be issued consecutively since the scheduling logic is pipelined. The other four stacked bars represent grouped instructions, categorized by instruction types and their dependences; *L-L indep* is two independent loads with the same base register; *S-S indep* is two independent single-cycle operations with identical source operands; *S-L dep* is a single-cycle operation followed by a dependent load; *S-S dep* is two dependent single-cycle oper-

ations. The data indicate that the MOP-2src case captures almost all grouping opportunities as MOP-3src, which allows three source operands in a MOP. This is consistent with the previous result in the macro-op scheduling in Figure 7-6. Comparing with the number of grouped instructions for macro-op scheduling shown in Figure 7-6 and Figure 7-9, S-S dep and S-S indep portions are almost identical. Combinations of a single-cycle ALU and a load (S-L dep), and two independent loads (L-L indep) account for a significant portion of the grouped instructions, which will help increasing the effective queue size and issue bandwidth. L-L indep MOPs are frequently generated from a sequence of instructions for stack manipulations. Note that these additional types of MOPs do not necessarily improve performance for macro-op scheduling and may degrade performance in some timing-critical cases, since issuing loads can be dynamically delayed by unresolved prior stores, memory port conflicts, store-to-load aliases or cache misses. These dynamic events create the same negative effect as the last-arriving operand in MOP tails, as shown in Figure 6.7.2. I will discuss the performance impacts of those MOPs shortly. Across the benchmarks, 41.2 ~ 59.5% of instructions are grouped, potentially increasing issue queue capacity by 24.1% on average.

9.4 Distribution of Effective Execution Bandwidth

Macro-op execution increases the machine bandwidth by issuing and executing multiple instructions in groups. Figure 9-3 and Figure 9-4 show the fraction of execution time categorized by the effective execution bandwidth in the base (4-wide-1-cycle) and macro-op execution (2-wide-MOP-3src) machines with 128-entry and 32-entry issue queues, respectively. The total execution time in each benchmark is normalized to that of

the 4-wide-1-cycle machine, so stacked bars in 2-cycle-MOP-3src may be plotted over or below the 100% line, depending on the relative performance. Note that these data were measured in terms of the number of instructions that start execution, as opposed to how many instructions are issued in a certain clock cycle. For example, if two MOPs are issued, and no other instructions are issued in the following clock cycle, the effective execution bandwidth in each cycle is measured to be two and two, as opposed to four and zero.

The results show that macro-op execution frequently achieves execution bandwidth greater than two instructions. This result clearly shows that the proposed approach to coarse-grained instruction processing is able to extract parallelism by exploiting serial portions of program. Compared to the base 4-wide-1-cycle machine, macro-op execution levels performance over time and utilizes execution resource more steadily, reducing 0-instruction execution cycles.

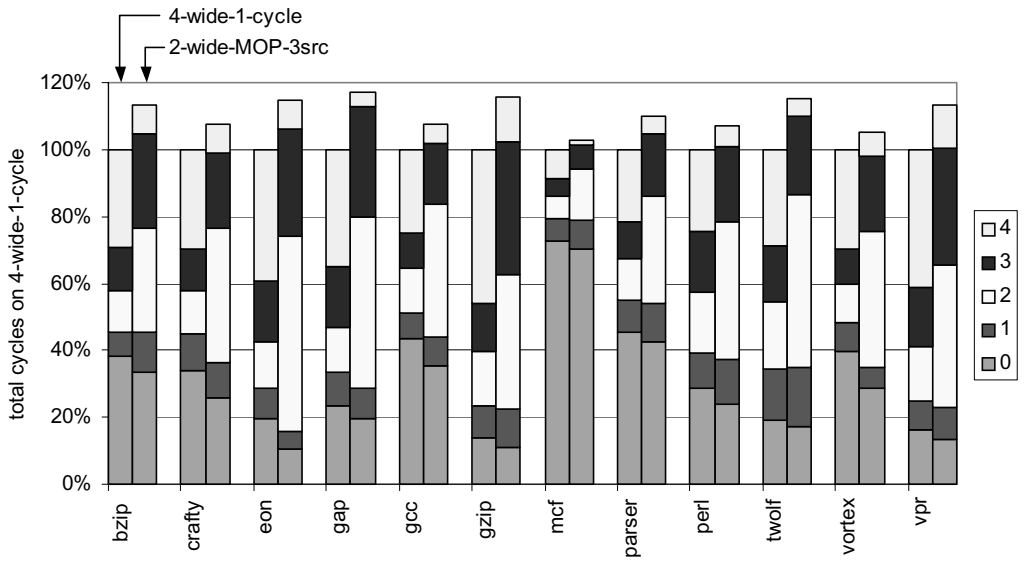


FIGURE 9-3. Effective machine bandwidth in macro-op execution (128-entry issue queue).

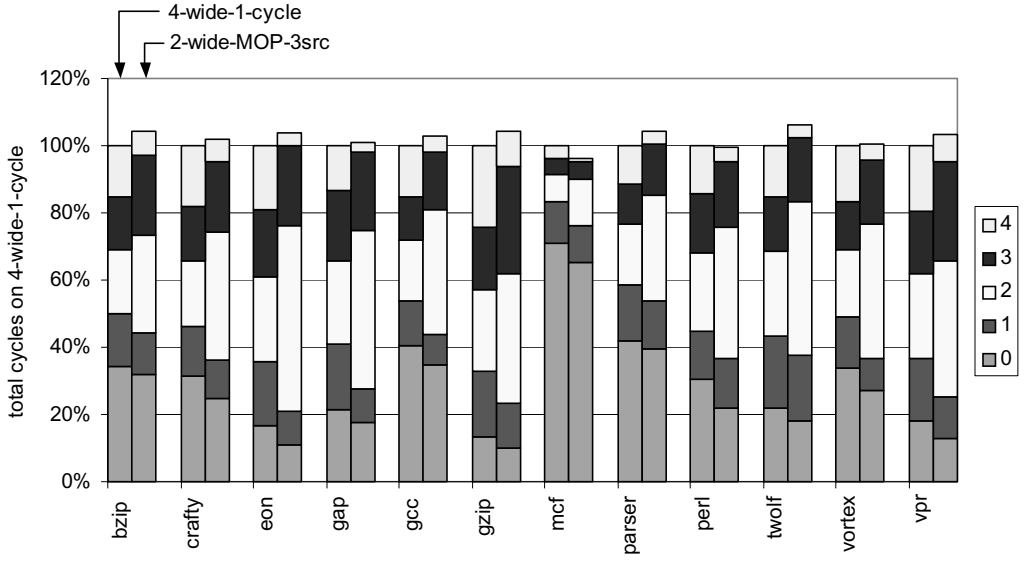


FIGURE 9-4. Effective machine bandwidth in macro-op execution (32-entry issue queue).

Figure 9-5 through Figure 9-10 present the IPCs of various machine configurations normalized to the base case (4-wide-1-cycle), as the issue queue size varies from 32 to 128. In the graphs, the first two bars from the left (2-wide-1-cycle and 2-wide-2-cycle) represent the 2-wide issue machines with 1-cycle and 2-cycle scheduling, respectively. They suffer significant performance drops due to the limited issue and execution bandwidth. *Mcf* shows the least performance degradation since its performance is mostly bounded by frequent cache misses. Compared to the performance data shown for macro-op scheduling in Figure 7-7, atomic instruction scheduling becomes less important in many benchmarks because the limited issue and execution bandwidth is the most significant performance bottleneck in this 2-wide issue configuration. It should be noted that the performance with the 32-entry issue queue is offset by 3~9% compared with other machine configurations with 48-entry or greater issue queues, exhibiting less performance degradations. This is because the 32-entry issue queue becomes the bottleneck of the base 4-wide-1-cycle machine and hence narrowing the machine bandwidth affects performance less significantly, as described in Section 3.3.

The third bar (*2-wide-MOPsched-3src*) represents the performance of macro-op scheduling, which does not increase the effective machine bandwidth. For this case, the same MOP grouping policies as those of current macro-op execution is used. By comparing performance data for macro-op execution with this bar, the additional benefits of increased machine bandwidth can be estimated. As briefly discussed in Section 9.3, the current grouping policies were not chosen for macro-op scheduling but focus on maximizing the benefits of increased machine bandwidth. So, macro-op scheduling in this chapter

does not perform as efficiently as in Chapter 7 with different MOP configurations, although it is still effective for recovering the performance degradation incurred by 2-cycle scheduling in many benchmarks.

The last two bars on the right (*2-wide-MOP-2src* and *2-wide-MOP-3src*) represent macro-op execution with two or three source operands. These two cases perform similarly since the groupability is not fundamentally affected by the number of source operands, as presented in Figure 9-2. Macro-op execution improves performance over the 2-wide-1-cycle and 2-wide-2-cycle cases, recovering significant portions of performance drops that come from narrower machine bandwidth and pipelined instruction scheduling. Compared to the base 4-wide-1-cycle case with the 32-entry issue queue, the worst case is measured to be 6.7% of IPC loss in *gzip* (2-wide-MOP-2src). In other configurations with a bigger queue, the same benchmark loses up to 16.3% of the base IPC (128-entry). However, this case still achieves a significant performance boost over the 2-wide-1-cycle (13.1% speedup) and 2-wide-2-cycle (27.8% speedup) cases. The average IPC loss of macro-op execution is measured to be less than 8.2% for all issue queue configurations.

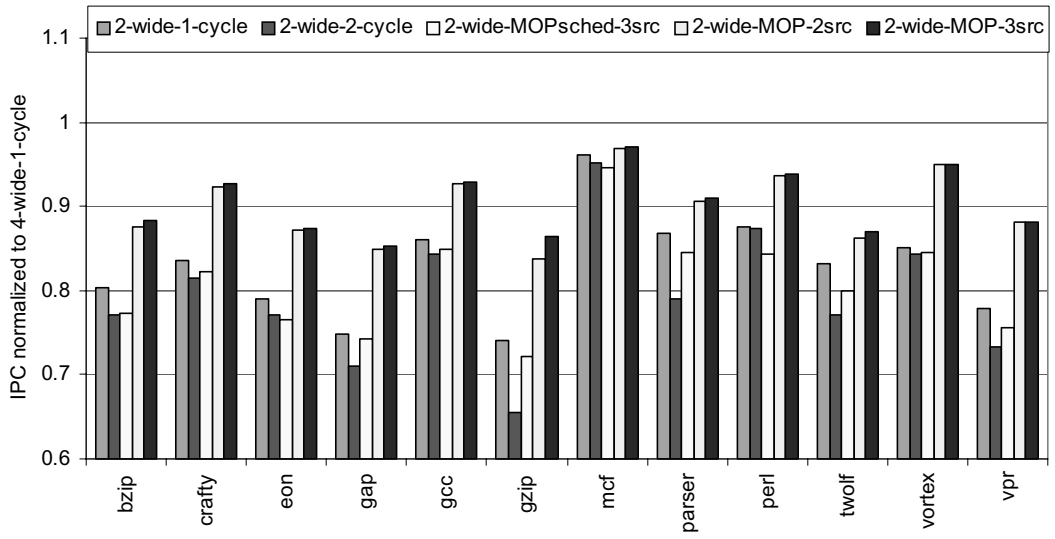


FIGURE 9-5. Performance of macro-op execution (128-entry issue queue).

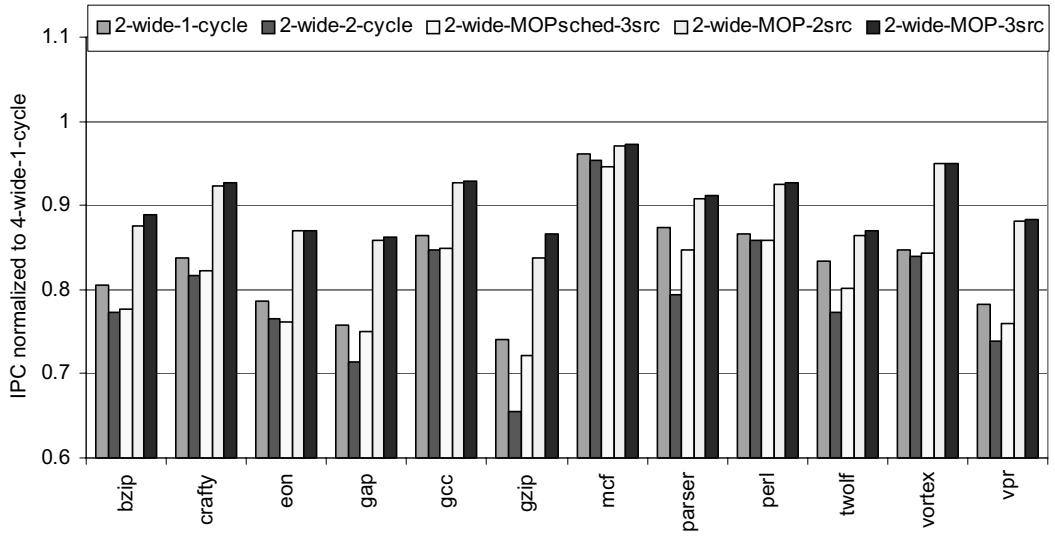


FIGURE 9-6. Performance of macro-op execution (64-entry issue queue).

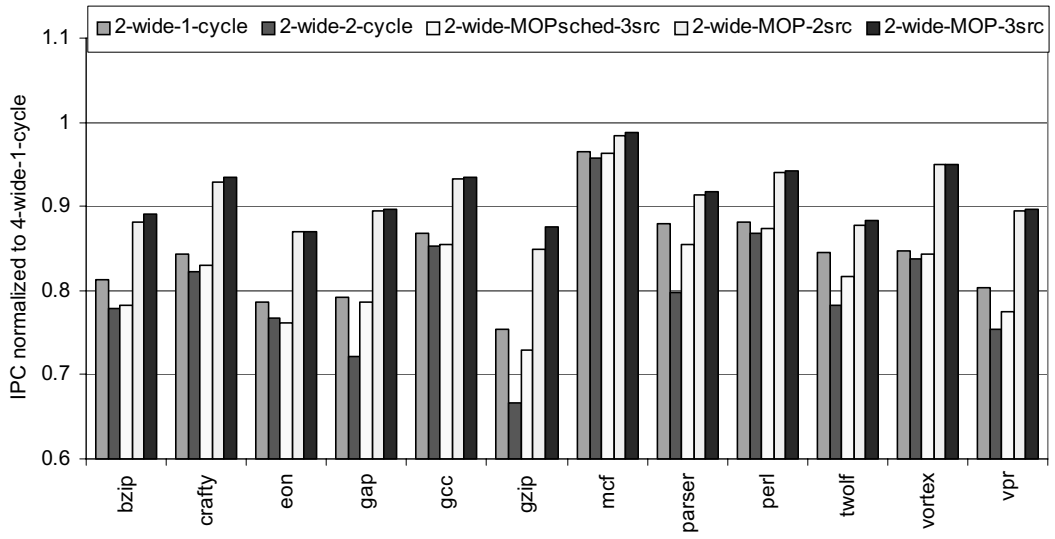


FIGURE 9-7. Performance of macro-op execution (48-entry issue queue).

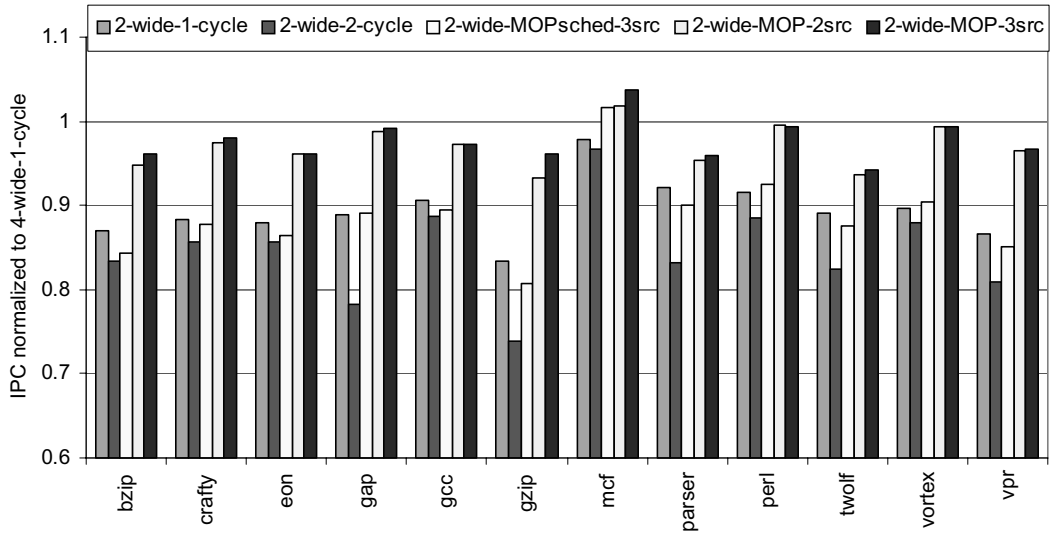


FIGURE 9-8. Performance of macro-op execution (32-entry issue queue).

Figure 9-9 and Figure 9-10 present the contributions to the speedup over the 2-wide-2-cycle machine, categorized by the benefits of either macro-op scheduling or the additional benefit of macro-op execution (3-src case). In the graphs, the sum of the two stacked bars in each benchmark represents the speedup of macro-op execution over the 2-wide-2-cycle machine. Since macro-op scheduling and execution in this chapter share the same grouping policies, macro-op scheduling should account for two benefits of macro-op execution: consecutive issue of dependent instructions and reduced issue queue contention. They are shown as the lower stacked bars (*MOP scheduling*), which are the speedup of macro-op scheduling (2-wide-MOPsched-3src cases in Figure 9-5 and Figure 9-8) over the 2-wide-2-cycle case.

In addition, macro-op execution increases the issue and execution bandwidth. This benefit is presented in a separate category of *increased BW*, which is the performance difference between macro-op execution and macro-op scheduling (2-wide-MOP-3src and 2-wide-MOPsched-3src in Figure 9-5 and Figure 9-8). The data indicate that increased issue and execution bandwidth accounts for a significant portion of performance improvement. With the 128-entry issue queue, some benchmarks suffer from macro-op scheduling and show a few percentage of performance drops, which are plotted in the negative direction. Again, this is because the current macro-op grouping policies are set to maximize the number of MOPs and increase the machine bandwidth, although they may unnecessarily serialize instructions; otherwise, instructions would have been issued in parallel through different slots. Potentially, MOP execution can employ adaptive grouping policies to dynamically disable certain MOPs that are likely to degrade performance based on issue bandwidth contention. The study of such optimizations is left to future work.

Figure 9-11 presents the average IPCs (harmonic mean) across the benchmarks on 4-wide-1-cycle, 3-wide-1-cycle, 2-wide-1/2-cycle and 2-wide macro-op execution machines with various issue queue sizes from 32 to 128. Macro-op execution with 2- and 3-source MOPs does not show a significant difference in performance, although the 3-source case slightly performs better. Its performance saturates around 48 or more issue queue entries, which is a similar trend compared to other 2-wide-1-cycle or 2-wide-2-cycle cases. Compared with 3-wide-1-cycle, 2-wide macro-op execution achieves 95 ~ 96% of its performance, with an exception of the case with the 32-entry issue queue in which the benefit from reduced queue contention enables macro-op execution to outperform the 3-wide-1-cycle case in many benchmarks. This performance trend is an expected behavior, since the MOP formation groups on average 48% of total instructions and this can be translated into, in theory, a 2.63-wide issue and execution bandwidth based on the equation presented in Section 8.2.10. Compared with 2-wide-1-cycle and 2-wide-2-cycle cases, macro-op execution achieves on average 7 ~ 8% and 11 ~ 13% of speedups over each case across all issue queue sizes, respectively.

In summary, macro-op execution increases the effective issue and execution bandwidth by processing multiple instructions as a single unit throughout the entire execution pipeline, with similar or even less hardware complexity in scheduler, dispatch, register file and bypass logic.

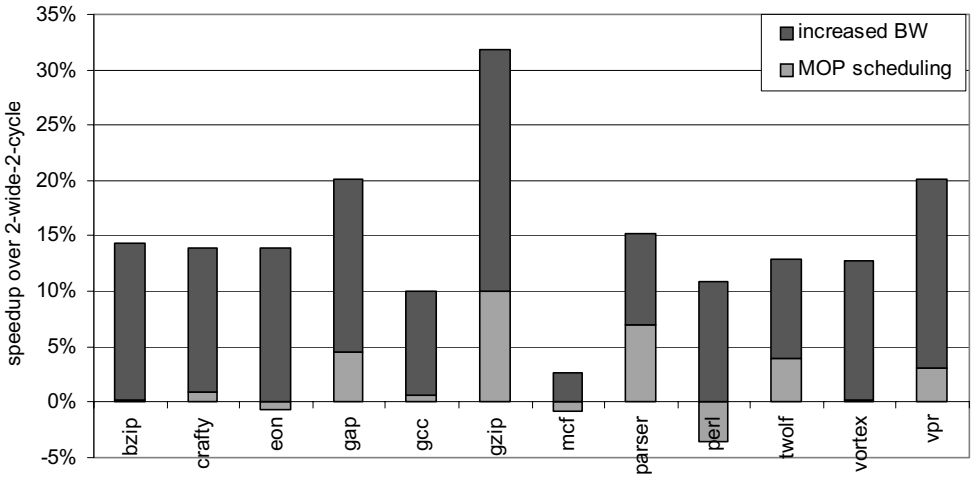


FIGURE 9-9. Contributions to the speedup (2-wide-MOP-3src, 128-entry issue queue).

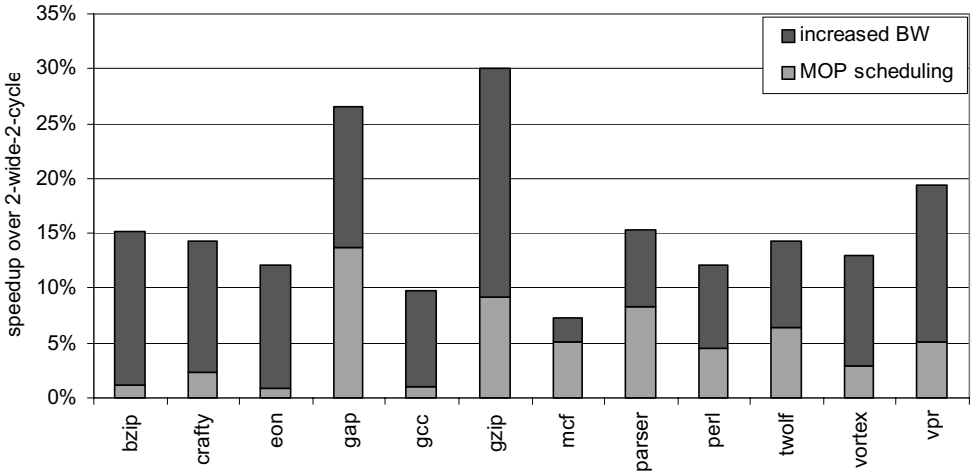


FIGURE 9-10. Contributions to the speedup (2-wide-MOP-3src, 32-entry issue queue).

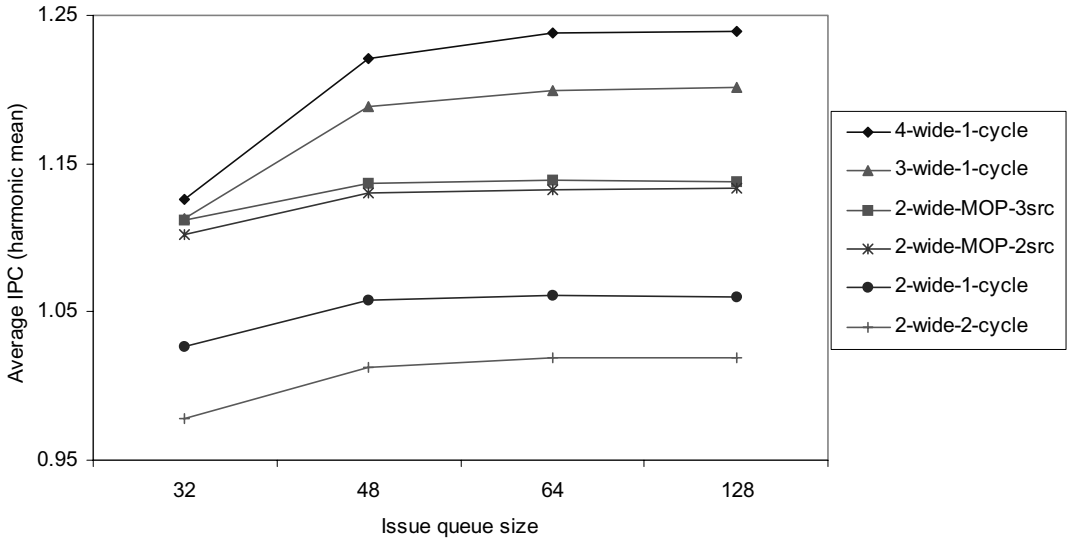


FIGURE 9-11. Performance of the base and macro-op execution machines.

9.6 Impact of MOP Scope

The current macro-op execution is configured to capture as many MOPs as possible to maximize the benefits of wider machine bandwidth. Base macro-op detection and formation logic captures MOPs with up to three sources within a 2-cycle scope (up to eight instructions). To measure the impact of MOP scope on macro-op execution performance, the MOP scope is increased from the base two cycles to three (up to 12 instructions) or four cycles (up to 16 instructions). In addition, the issue queue size changes from 32 to 128 entries so that its detailed performance impact can be observed.

Figure 9-12 shows the number of grouped instructions in the 2-wide-MOP-3src case with various MOP scopes. The graph does not categorize instructions into different types of MOPs since their differences are not significant. Each benchmark has three stacked bars that represent MOP scopes of two, three and four cycles. The base 2-cycle

scope achieves a compelling grouping efficiency compared to wider 3- or 4-cycle scopes, with a maximum difference of 6.6% in *bzip*.

Figure 9-13 through Figure 9-16 present the performance sensitivity of macro-op execution to various MOP scopes when the issue queue sizes change from 128 to 32. In each benchmark, two bars correspond to the IPCs of 3- and 4-cycle scopes normalized to the base 2-cycle scope. The average performance of 3- and 4-cycle scopes is slightly better than the 2-cycle scope case, although the difference is not significant. However, some benchmarks such as *bzip*, *gap* and *gzip* are measured to be more sensitive than other benchmarks. An interesting result is observed in *gap*, where the benefits of wider MOP scopes reduce as the issue queue becomes smaller. With the 32-entry issue queue, 3- or 4-cycle MOP scopes degrade performance compared with the 2-cycle MOP scope. This behavior is related to 2-cycle scheduling performance. In *gap*, a larger issue queue incurs more issue bandwidth contention, which hides the performance degradation due to 2-cycle scheduling. As the queue becomes smaller, the impact of 2-cycle scheduling becomes more significant, and therefore the performance gap between 1- and 2-cycle scheduling becomes bigger. The benefits of aggressively grouping instructions do not exceed the their negative impacts on scheduling performance, resulting in a slight performance degradation.

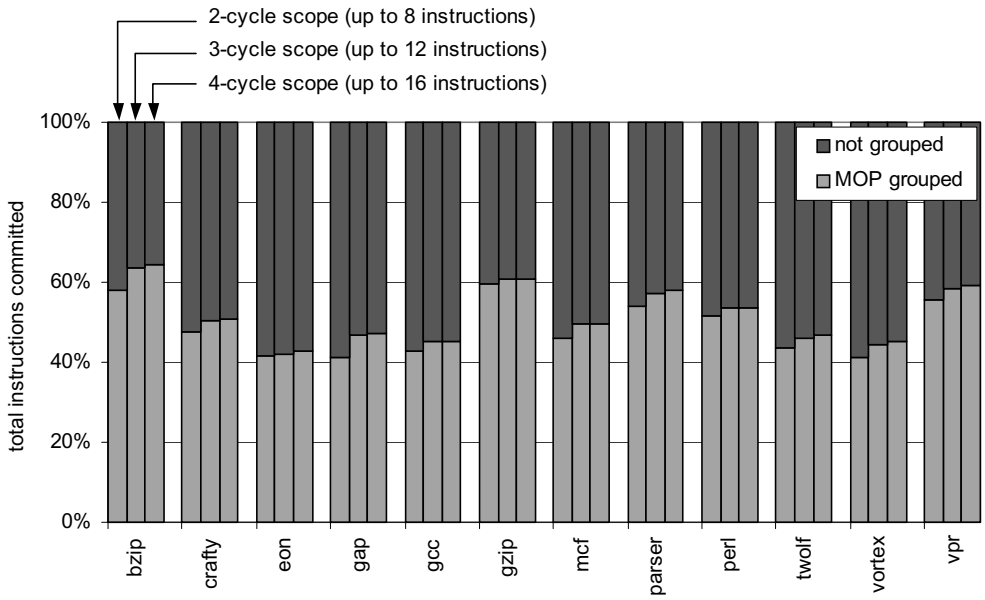


FIGURE 9-12. Impact of MOP scope on MOP coverage (2-wide-MOP-3src, 128-entry issue queue).

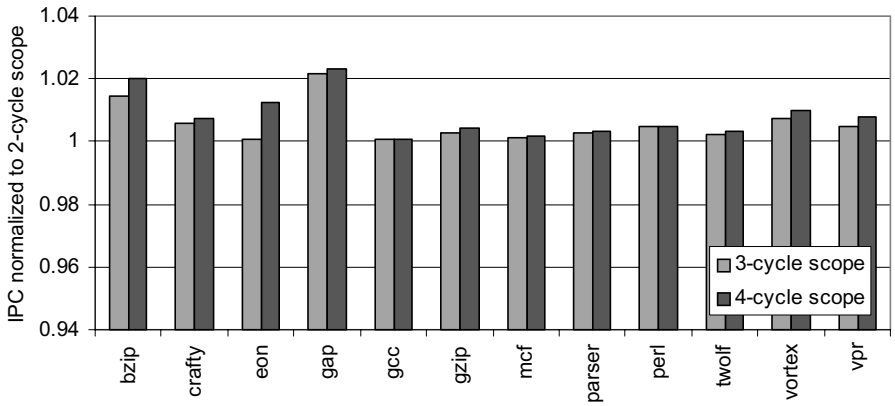


FIGURE 9-13. Performance impact of MOP scope (2-wide-MOP-3src, 128-entry issue queue).

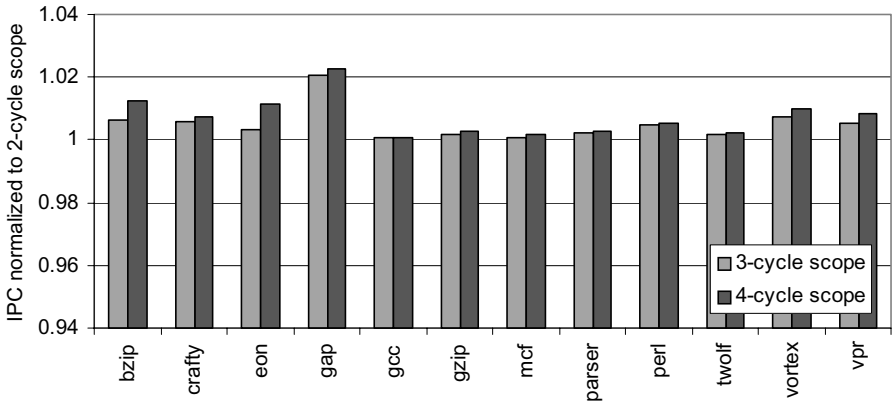


FIGURE 9-14. Performance impact of MOP scope (2-wide-MOP-3src, 64-entry issue queue).

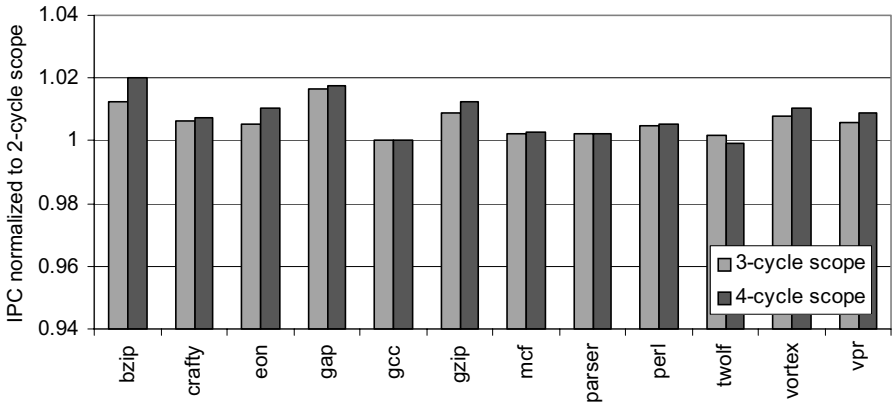


FIGURE 9-15. Performance impact of MOP scope (2-wide-MOP-3src, 48-entry issue queue).

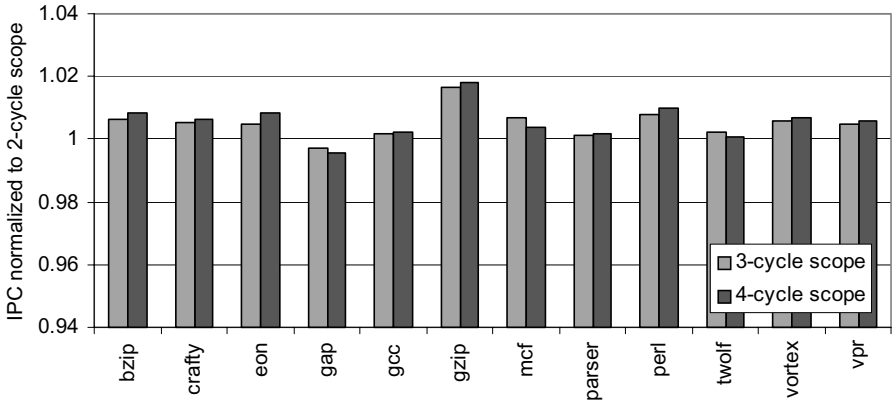


FIGURE 9-16. Performance impact of MOP scope (2-wide-MOP-3src, 32-entry issue queue).

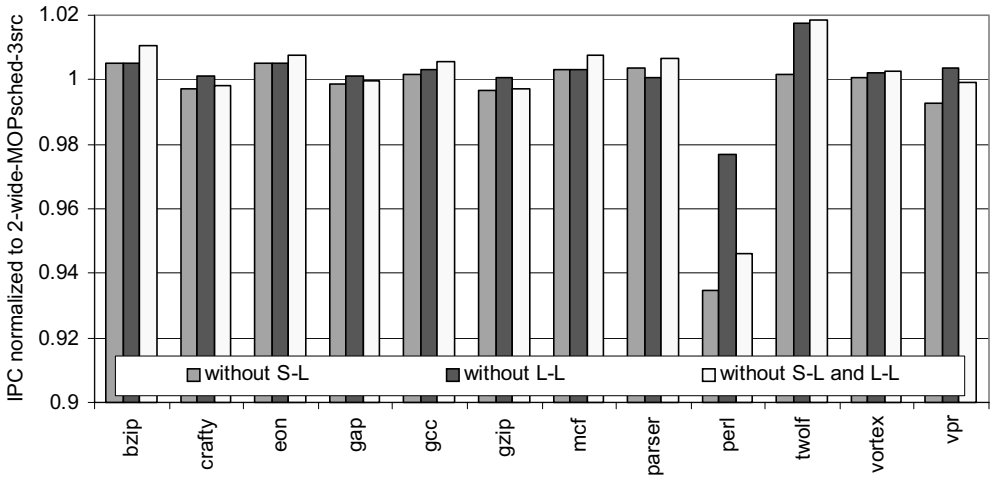


FIGURE 9-17. Performance sensitivity of macro-op scheduling to MOP types (2-wide-MOPsched-3src, 128-entry issue queue).

The grouping policy that determines the combinations of instructions in MOPs should be carefully considered since it may affect performance negatively. In addition to the MOPs of two single-cycle instructions (S-S dep and S-S indep, referred to as dependent and independent MOPs used for macro-op scheduling in Chapter 6), two additional types of MOPs are used for macro-op execution: S-L and L-L MOPs. To evaluate how these MOPs affect performance, I first measure the performance of macro-op scheduling without those additional MOP types. This experiment decouples the benefit of increased issue bandwidth so that it affects instruction scheduling only. Then, I measure the performance of macro-op execution without them to determine whether they are beneficial.

Figure 9-17 shows the result of the first experiment for measuring the performance sensitivity of macro-op scheduling. The base machine has the 128-entry issue queue. Each benchmark has three bars that represent the cases in which S-L, L-L or the both types are not generated. All performance is normalized to the base macro-op scheduling case with

the same grouping policy as the one for 2-wide-MOP-3src. The base macro-op scheduling performance was presented as 2-wide-MOPsched-3src categories in Figure 9-5. If the normalized performance exceeds one when a certain MOP type is removed, the result indicates that it negatively affects instruction scheduling for some reason, e.g. unnecessarily delaying MOP head instructions. Conversely, normalized performance below one implies that the MOP type further improves scheduling performance. Note that the base machine with a 2-wide issue bandwidth and the 128-entry issue queue is less sensitive to 2-cycle scheduling than the machine with a smaller issue queue, as the data in Figure 9-5 indicate. This is because a wider instruction window can easily find ready instructions to fill two issue slots. Although a smaller window makes the machine more sensitive to 2-cycle scheduling, the current issue queue size is chosen to decouple the benefit of reduced issue queue contention.

The results in the *without S-L* category indicate that S-L MOPs do not always improve macro-op scheduling, although they shorten 1-cycle dependence edges between a single-cycle instruction and a load, which implies that the memory disambiguation policy used in the current machine model delays many loads and turns them into not useful and harmful MOPs. In contrast, they positively affect some benchmarks; the IPC in *perl* drops by 6.5% without S-L MOPs.

L-L MOPs contain two independent load instructions and therefore do not additionally enable back-to-back instruction scheduling for macro-op scheduling. Rather, grouping independent instructions tends to degrade performance, as discussed in Section 6.7.1. *Twolf* loses 2% of the IPC due to this effect. In contrast, *perl* is positively affected by L-L MOPs, which is not an expected behavior. This result simply comes from a sec-

When neither MOP types are generated (*without S-L and L-L*), macro-op scheduling tend to perform slightly better in many cases with the exception of *perl*, although the negative effects are not significant.

Figure 9-18 to Figure 9-21 show how S-L and L-L MOPs affect macro-op execution. The performance impacts are measured and plotted in the same way as the previous experiment, except that 2-wide macro-op execution is used for this experiment. All performance is normalized to the 2-wide-MOP-3src case, which was presented in Figure 9-5.

With the 128-entry issue queue (Figure 9-18), the issue bandwidth contention is high enough for S-L and L-L types to compensate for the negative impacts on scheduling performance. As the issue queue size and the issue bandwidth contention is reduced (Figure 9-19 to Figure 9-21), the issue bandwidth benefits tend to decrease. However, the combined effects (*without S-L and L-L*) improve macro-op execution in most cases.

In summary, the additional types of S-L and L-L MOPs may degrade the performance of instruction scheduling but are still beneficial in macro-op execution because they increase machine bandwidth.

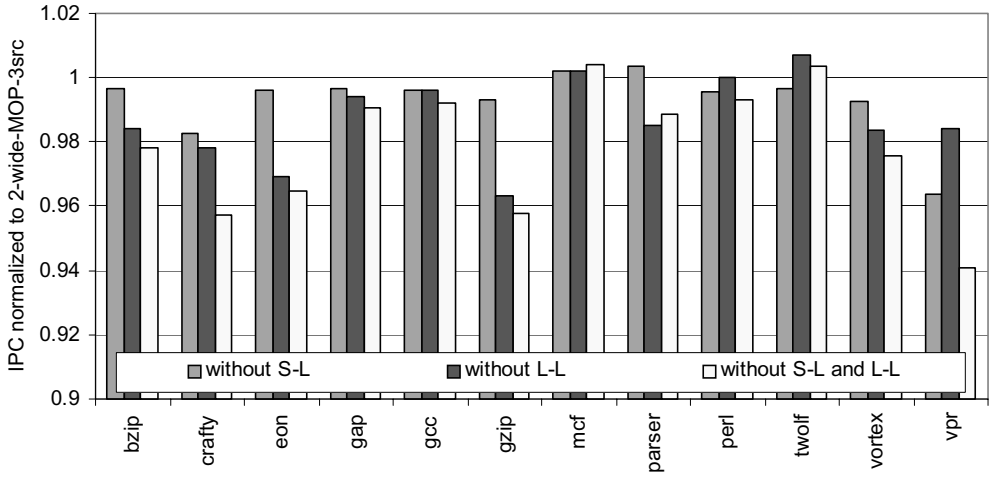


FIGURE 9-18. Performance sensitivity to MOP types (2-wide-MOP-3src, 128-entry issue queue).

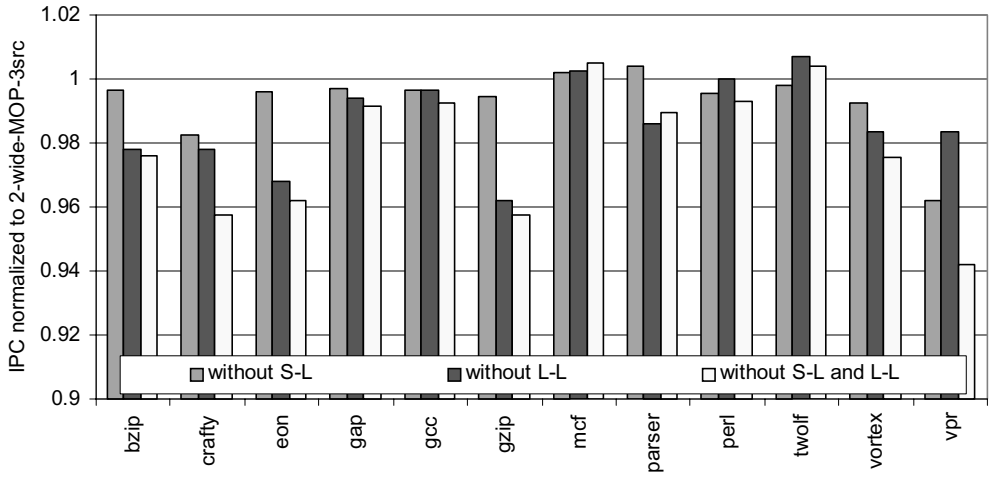


FIGURE 9-19. Performance sensitivity to MOP types (2-wide-MOP-3src, 64-entry issue queue).

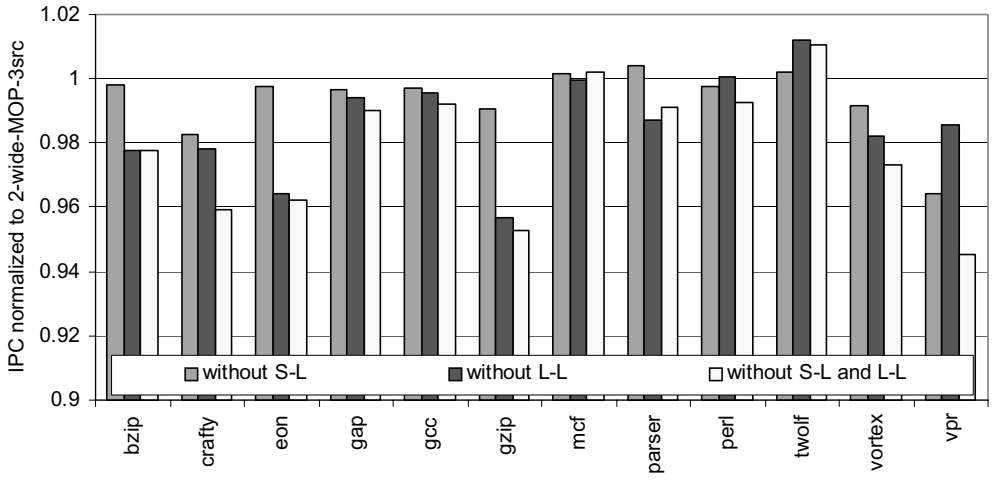


FIGURE 9-20. Performance sensitivity to MOP types (2-wide-MOP-3src, 48-entry issue queue).

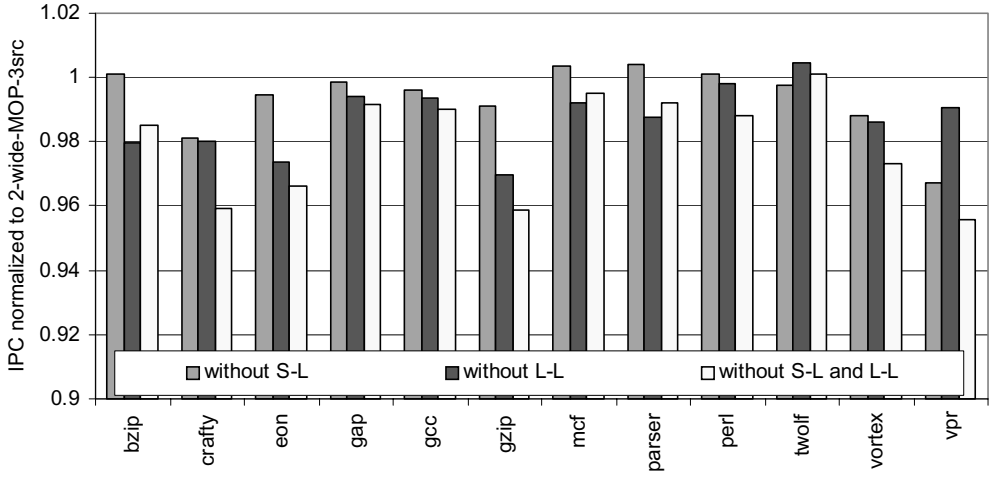


FIGURE 9-21. Performance sensitivity to MOP types (2-wide-MOP-3src, 32-entry issue queue).

9.8 Macro-op Execution for Wider Machine Bandwidth

So far, macro-op execution has been tested using a narrow 2-wide issue machine that achieves a peak bandwidth of four instructions per cycle. In reality, MOP detection and formation cannot group all instructions nor achieve 100% of MOP coverage, so the actual machine bandwidth that macro-op execution achieves is between two and three instructions (theoretical bandwidth of on average 2.63 instructions, ignoring other factors like pipelined instruction scheduling or reduced issue queue contention).

If we build macro-op execution on a 3-wide or wider machine, macro-op execution may achieve an even greater execution bandwidth than the base 4-wide machine, at the expense of more register write ports and read ports (for 3-source MOPs) and functional units. The theoretical effective execution bandwidth of 3-wide and 4-wide macro-op execution, given the MOP coverage of 48%, is 3.95 and 5.27 instructions, respectively. From the perspective of other hardware resources such as scheduling logic, issue ports and dispatch bandwidth, macro-op execution achieves similar hardware complexity to a conventional machine with the same issue bandwidth and 2-cycle scheduling logic. Bypass logic complexity is similar to that of a conventional machine with the same issue bandwidth and conventional 1-cycle scheduling.

The performance of 3-wide and 4-wide macro-op execution is presented in from Figure 9-22 to Figure 9-25 when the issue queue size varies from 128 to 32 entries. In each graph, all performance is normalized to the 4-wide-1-cycle case. Each benchmark has five bars: *3-wide-1-cycle*, *3-wide-2-cycle*, *3-wide-MOP-2src*, *3-wide-MOP-3src*, and *4-wide-MOP-3src*. The first two bars from the left (*3-wide-1-cycle* and *3-wide-2-cycle*) represent the performance when the issue bandwidth of the base machine is reduced to

three instructions, with 1-cycle and 2-cycle scheduling logic, respectively. The next two bars (*3-wide-MOP-2src* and *-3src*) represent the performance of macro-op execution built on the *3-wide-2-cycle* case, with 2- and 3-source MOPs. The last bar on the right presents an extreme case, in which macro-op execution is configured to achieve a peak bandwidth of eight instructions, built based on the base *4-wide-2-cycle* machine. Note that only the number of functional units for simple integer operations is increased in macro-op execution. Other functional units for complex operations or memory ports are the same with or without macro-op execution. Figure 9-26 presents the average performance of 3-wide and 4-wide macro-op execution with various issue queue sizes.

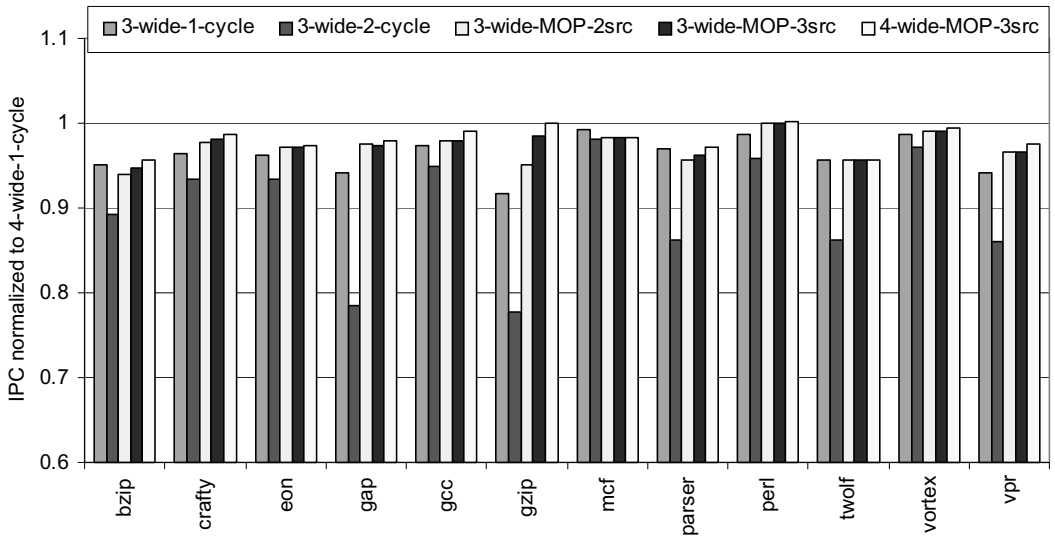


FIGURE 9-22. Performance of macro-op execution (3- and 4-wide-MOP, 128-entry issue queue).

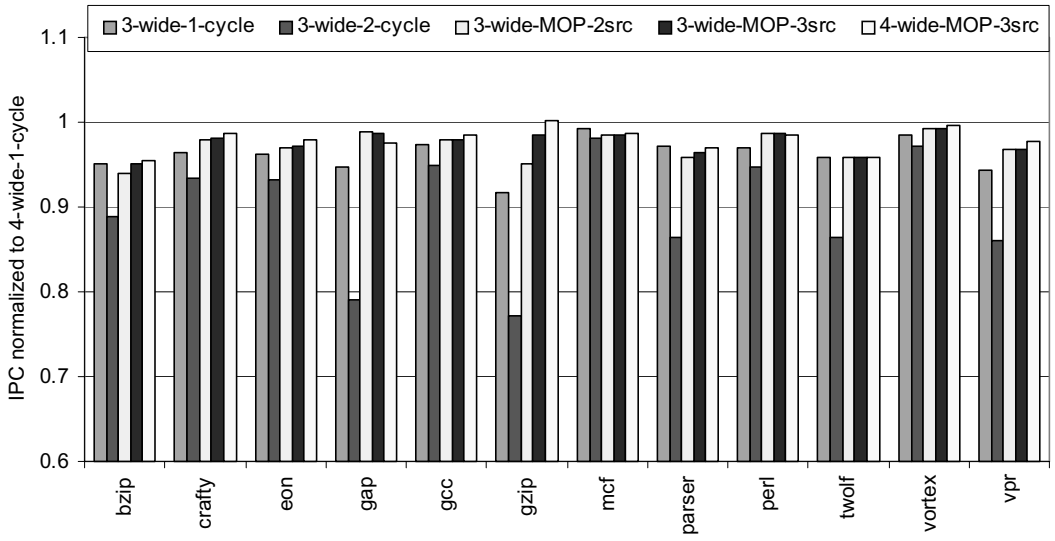


FIGURE 9-23. Performance of macro-op execution (3- and 4-wide-MOP, 64-entry issue queue).

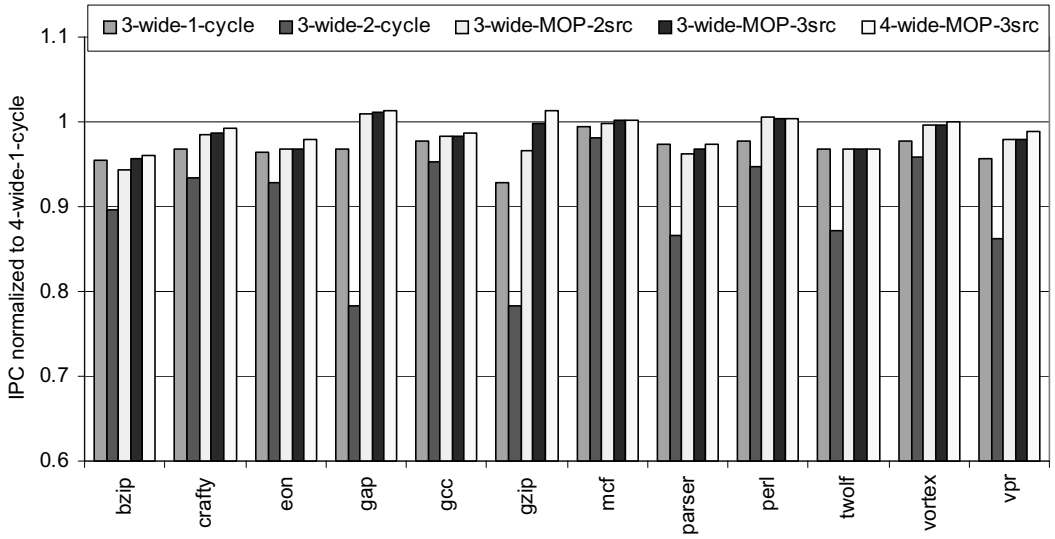


FIGURE 9-24. Performance of macro-op execution (3- and 4-wide-MOP, 48-entry issue queue).

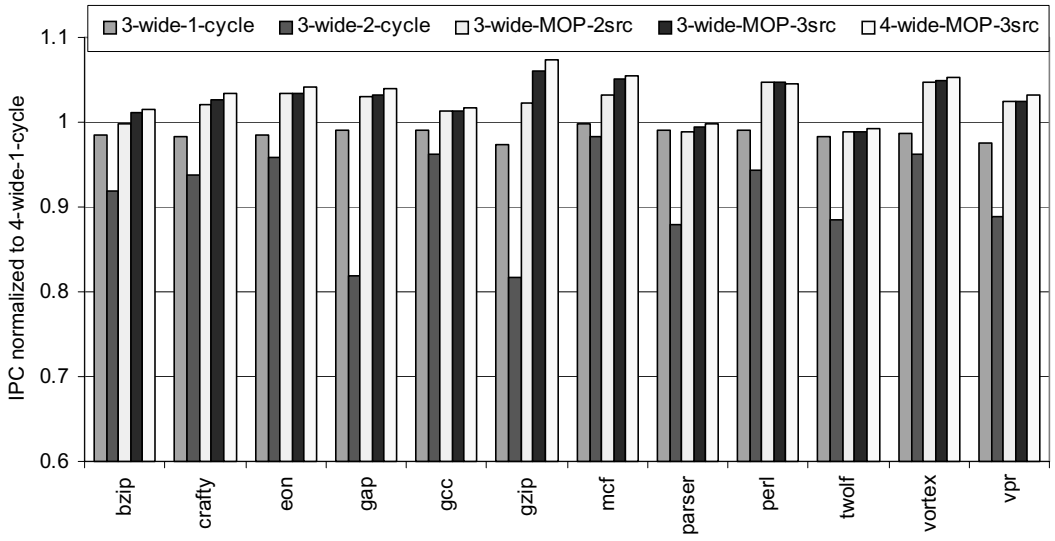


FIGURE 9-25. Performance of macro-op execution (3- and 4-wide-MOP, 32-entry issue queue).

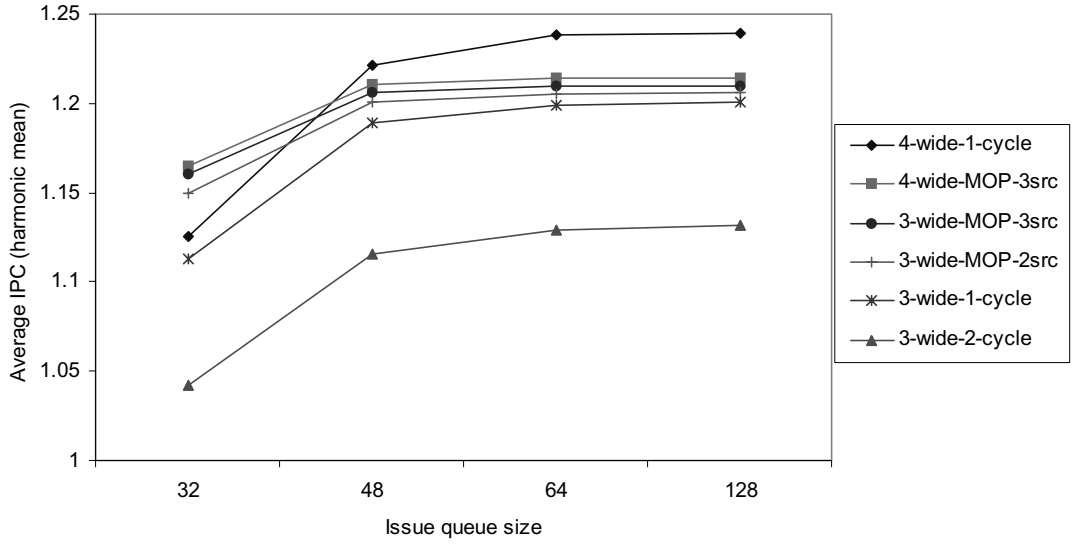


FIGURE 9-26. Performance of 3- and 4-wide macro-op execution with various issue queue sizes.

With the exception of the 32-entry and 48-entry issue queue cases in Figure 9-26 where macro-op execution greatly benefits from reduced issue queue contention, 3-wide macro-op execution does not significantly improve performance over 3-wide-1-cycle. Even 4-wide macro-op execution does not outperform the base performance (4-wide-1-cycle) in most cases. Considering the theoretical bandwidth of 3.95 or 5.27 instructions for each case, the actual results are somewhat disappointing.

The performance of macro-op execution is bounded by many factors. Among the three major benefits of macro-op execution--reduced issue queue contention, consecutive instruction issue and increased machine bandwidth, macro-op execution benefits least from the first benefit (reduced issue queue contention) on a machine with a large issue queue, because queue contention is not a major performance bottleneck. Since macro-op execution is built based on 2-cycle scheduling and macro-op scheduling cannot fully achieve the performance of 1-cycle scheduling although the second benefit (consecutive instruction issue) recovers a significant portion of the performance loss, the third benefit (increased machine bandwidth) should be greater than the performance loss from macro-op scheduling, in order for macro-op execution to outperform the base case. Unfortunately, the benefit from increased machine bandwidth is not significant in many benchmarks on the current base machine.

Table 9-1 presents the performance gain achieved by doubling the execution bandwidth from four to eight instructions (128-entry issue queue). The performance of the 8-wide-1-cycle configuration is the upper limit that the 4-wide macro-op execution can achieve. The maximum speedup is measured to be 2.71% of IPC improvement in *gzip*. Most benchmarks achieve less than 1% of performance gain (on average 0.98%). This

Table 9-1: Performance of the base machine when the execution bandwidth is doubled.

Benchmarks	4-wide fetch, 4-wide-1-cycle, 128-entry IQ	4-wide fetch, 8-wide-1-cycle, 128-entry IQ	Speedup
bzip	1.53	1.55	1.01%
crafty	1.55	1.57	0.94%
eon	2.13	2.14	0.71%
gap	2.10	2.13	1.39%
gcc	1.29	1.30	0.76%
gzip	1.99	2.04	2.71%
mcf	0.38	0.38	0.19%
parser	1.12	1.13	0.83%
perl	1.31	1.34	2.00%
twolf	1.50	1.52	1.89%
vortex	1.75	1.75	0.26%
vpr	1.64	1.67	1.95%

result implies wider issue and execution bandwidth is not fully utilized and there are few cases where instructions are delayed due to limited execution bandwidth.

To compare the degree of execution bandwidth utilization, Figure 9-27 presents the fraction of execution time categorized by the effective execution bandwidth in 4-wide-1-cycle, 8-wide-1-cycle and macro-op execution (4-wide-MOP-3src) cases. The total execution time in each benchmark is normalized to that of the 4-wide-1-cycle machine, so the total height of the stacked bars in the 8-wide-1-cycle and 8-cycle-MOP-3src cases may be lower or higher than the 100% line, depending on relative performance. The graph indicates that macro-op execution infrequently utilizes the wider execution bandwidth and performs slightly worse than the base 4-wide-1-cycle case. Even the 8-wide-1-cycle case spends only ~10% of the total cycles executing more than four instructions in many benchmarks.

The result comes primarily from the fact that instruction supply to the out-of-order window is limited by the finite fetch bandwidth (i.e. 4-wide fetch on the base machine). In

Section 5.3, the performance of the base machine (4-wide, 128-entry issue queue and ROB) was measured as its hardware constraints are relaxed. Table 9-2 presents the IPCs measured in that experiment. Comparing the fourth column (no constraints for branch prediction and memory) and the fifth column (no constraints for branch prediction, memory and execution bandwidth), providing unlimited execution bandwidth improves performance by only 12.2% on average across the benchmarks, whereas relaxing the fetch bandwidth enables a much greater improvement.

More importantly, the fundamental benefits of transitioning to a wider machine should be considered. The IPCs in the fourth column in Table 9-2 can be interpreted as performance of the base machine running at a full speed in a steady state when no branch misprediction or cache miss occurs. Even in this case, the IPC usually does not exceed three instructions per cycle. These results explain why 3- or 4-wide macro-op execution does not perform well, while 2-wide macro-op execution performs substantially better than the conventional case. Reducing the issue bandwidth below those numbers (i.e. three instructions) may create a significant amount of contention, and macro-op execution is efficient at resolving it. In contrast, an issue bandwidth of three or more instructions, given the hardware constraints in this pipeline, may be sufficient to run many programs and therefore the machine does not have enough issue bandwidth contention for macro-op execution to matter.

Due to these limitations, an 8-wide machine bandwidth across the entire pipeline stages including fetch, decode, execute and commit, does not provide twice or near twice the performance of a 4-wide machine. For example, doubling all of the machine bandwidth, queue sizes and resources achieves only on average 27% of IPC improvement

(ranging from 16.3% to 47.8%). If the queue sizes (including issue queue and ROB) and the number of memory ports do not change, it improves performance by only 2% ~ 16%. Consequently, when a machine with 8-wide fetch bandwidth is tested, macro-op execution with 4-wide issue bandwidth provides only a marginal benefit over a conventional machine with 4-wide issue bandwidth.

In summary, macro-op execution achieves only marginal performance benefits on a 3-wide or 4-wide issue machine when issue queue contention is not significant. This is partly because of the limited fetch bandwidth, and partly because of the characteristics of the benchmark programs, which have little demand for additional execution bandwidth.

A potential application of macro-op execution for wider machine bandwidth would be simultaneous multithreading (SMT) [94][95][20][100][49][65]. Since the shared instruction queue can extract more parallelism from multiple independent threads than from a single thread, many execution resources in the processor can be better utilized. Meanwhile, wire delay constraints are forcing hardware designers to divide pipeline resources into clusters to exploit physical communication locality [78][74][30][53][17]. When SMT is applied to such clustered designs, it is necessary to efficiently partition resources to different threads to improve the processor's ability to share execution resources. The macro-op execution model may provide a good approach to partitioning execution resources; a macro-op efficiently localizes value communications through chains of dependent instructions within a single thread with limited instruction-level parallelism, while multiple macro-ops enable increasing machine bandwidth by scheduling multiple independent threads in an interleaved fashion. The study of macro-op execution on SMT processors is left to future work.

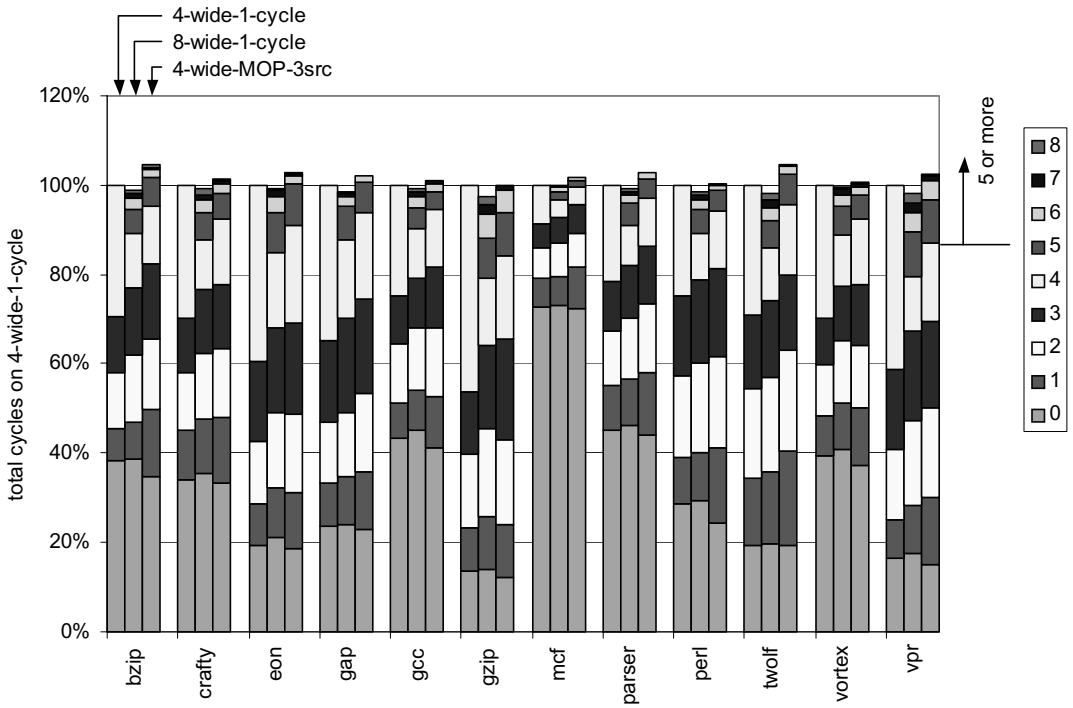


FIGURE 9-27. Comparison of effective machine bandwidth.

Table 9-2: IPCs of the base machine when hardware constraints are relaxed.

Benchmarks	Base	- Branch prediction	- Memory	- Execution BW	- Fetch / commit
bzip	1.57	1.76	2.95	3.24	4.72
crafty	1.71	1.88	3.33	3.56	6.24
eon	2.38	2.54	2.84	3.40	4.34
gap	2.24	2.35	2.99	3.30	3.60
gcc	1.38	1.57	2.84	3.51	6.45
gzip	2.04	2.66	2.97	3.26	3.95
mcf	0.38	0.40	2.77	3.12	5.50
parser	1.13	1.29	2.66	2.93	3.65
perl	1.34	2.17	2.48	2.66	2.68
twolf	1.55	2.18	2.60	2.89	3.11
vortex	1.97	1.82	3.06	3.56	5.14
vpr	1.70	2.73	3.13	3.47	3.98
Average (harmonic mean)	1.29	1.49	2.87	3.21	4.16

9.9 Summary and Conclusions

Macro-op execution is evaluated and several aspects are examined. The experimental results indicate that macro-op detection and formation captures a significant portion of instructions. This enables a 2-wide machine (with 4-wide fetch bandwidth) to work as a wider machine with a peak bandwidth of four instructions, with similar or less complexity in many pipeline stages.

Macro-op execution is effective when issue bandwidth is limited and issue queue contention is high. On a machine with 2-wide issue bandwidth, macro-op execution recovers a significant portion of 4-wide machine performance. With a 32-entry issue queue, macro-op execution can nearly achieve the performance of the base 4-wide machine with 1-cycle scheduling. I also study the performance impact of MOP scope. Wider MOP scopes generally improve macro-op execution, but the 2-wide scope achieves most of the benefits, and the performance difference compared with wider scopes is not significant. The performance impact of additional macro-op types is also studied. Grouping loads as either macro-op heads or tails may not improve instruction scheduling, but it is beneficial for macro-op execution since it further increases machine bandwidth as well as reduces issue queue contention.

Macro-op execution exhibits only a marginal performance benefit when implemented on a 3- or 4-wide issue machine because performance benefits for wider machine bandwidth are constrained by fetch bandwidth as well as the limited parallelism of the benchmark programs tested.

Chapter 10

Conclusions

In this thesis, I make three main contributions. First, I present the concept of coarse-grained instruction processing, which reduces the hardware overhead involved in coordinating all of the concurrent actions in a modern out-of-order superscalar processor. Second, I quantify the degree of abstraction that can be applied to a program for coarse-grained instruction processing, and find that a significant portion of the instructions can be processed together in groups without requiring fine-grained, instruction-level controls for scheduling and execution. Third, I propose and study two microarchitectural techniques called *macro-op scheduling* and *macro-op execution*, for exploiting the benefits of coarse-grained instruction processing in the scheduling and execution pipeline stages in an out-of-order superscalar processor.

Coarse-grained instruction processing is enabled by grouping multiple instructions into macro-ops. A macro-op is an atomic schedulable unit that contains multiple instructions with a sequential execution order. Grouping instructions into macro-ops has two major benefits. First, it reduces the number of schedulable units, among which fewer dependences and boundaries are tracked and preserved. Second, it restricts scheduling and executing instructions to a subset of the possible outcomes in a foreseeable manner. These properties can be applied towards reducing hardware complexity involved in instruction scheduling and execution.

Macro-op scheduling performs coarse-grained instruction scheduling. To ensure back-to-back execution of dependent instructions, the conventional instruction scheduling

performs a set of wakeup and select operations atomically every clock cycle. This in turn prevents the out-of-order window from growing beyond a certain size, restricting its scalability. To overcome these limitations, macro-op scheduling groups multiple instructions and forces scheduling decisions to occur at coarser macro-op boundaries. Macro-op scheduling performs pipelined scheduling of multi-cycle macro-ops while the processor core still executes dependent instructions consecutively. Combined with the relaxed scalability constraint due to fewer schedulable units, it achieves comparable or better performance than conventional atomic scheduling.

Macro-op execution extends the range of coarse-grained instruction processing to the entire pipeline for out-of-order execution. Multiple original instructions in a macro-op are managed and processed together in the pipeline as a single unit until they reach the execution stage in which the functional units and bypass paths are configured in such a way that the original instructions are naturally sequenced. In addition to the benefits of macro-op scheduling, i.e. larger instruction window and pipelined scheduling logic, macro-op execution increases the machine bandwidth in instruction scheduling, dispatch, register access and execution with similar or less hardware complexity.

10.1 Thesis Summary

The following sections summarize the work performed for this thesis study on macro-op scheduling and execution.

10.1.1 Groupability of instructions for coarse-grained instruction processing

The benefits from grouping instructions can be summarized as dependence abstraction and increased scheduling determinism. Several issues in grouping instructions

into macro-ops are discussed. Grouping chains of dependent instructions is more beneficial than independent instructions since a macro-op forces sequential execution of the instructions. Macro-op grouping may reduce instruction-level parallelism by unnecessarily serializing instructions. Therefore, macro-op grouping requires a judicious policy to maximize its benefits.

To evaluate the degree of grouping instructions into macro-ops and measure its potential, several experiments are performed. The dependence edge distances are first characterized to determine the scope of searching for groupable candidate instructions. The characterization result indicates that many dependent macro-op candidates are placed near each other, and the vast majority is captured within eight instructions in program order. The groupability of instructions are quantified in terms of the fraction of instructions that can be grouped, the number of instructions in each macro-op, and the impact of macro-op grouping on the instruction-level parallelism. For a set of SPEC2K integer benchmarks, a significant number of instructions can be processed in groups -- 27 ~ 65% of total instructions, depending on benchmarks and grouping policies. The data also indicate that carelessly grouping instructions significantly degrades instruction-level parallelism. To determine the size of macro-ops, the average number of candidate instructions in a dependence chain is measured. Dependence chains tend to be short and macro-ops with two instructions account for the vast majority of grouping opportunities, although three or more instructions can be grouped in many cases.

10.1.2 Macro-op scheduling

This work is divided into three categories. First, I analyze the impact of pipelining instruction scheduling logic on performance, and find the reasons for different perfor-

mance sensitivities to it across benchmarks. Second, the basic concept of macro-op scheduling and its microarchitecture is detailed. Finally, several aspects of macro-op scheduling and its performance benefits are studied and evaluated.

10.1.2.1 Analysis of pipelined instruction scheduling logic

When conventional atomic instruction scheduling logic (referred to as 1-cycle scheduling) is pipelined over two separate stages (referred to as 2-cycle scheduling), it loses the capability to issue instructions dependent on a single-cycle instruction in the consecutive clock cycle. The average IPC loss due to 2-cycle scheduling is measured to be 6.5% on a 4-wide machine that this study is based on. The performance loss due to 2-cycle scheduling is not easily recovered by a wider instruction window potentially enabled by pipelining the logic. An experimental result shows that 2-cycle scheduling with a 256-entry issue queue achieves only 83.5% of the performance of 1-cycle scheduling with a 48-entry issue queue in *gzip*.

Not all benchmarks are sensitive to 2-cycle scheduling. On our base machine model, some benchmarks lose more than 10% of performance (up to 21%) while some others show virtually no performance degradation. This performance insensitivity can be caused by many hardware constraints that hide the extra delays incurred by 2-cycle scheduling. Among those hardware constraints, the machine bandwidth that determines how fast instructions are delivered to the out-of-order window greatly affects the sensitivity. Some program characteristics strengthen or weaken the trend of insensitivity given the hardware constraints. The results of several experiments show that the performance sensitivity is correlated to dependence edge distances. A program with long dependence edges (i.e. a value-generating instruction and its dependent instructions are placed across many

intervening instructions in program order) tends to be insensitive to 2-cycle scheduling. This is because given a finite fetch bandwidth, many instructions are issued before their dependent instructions enter the instruction window and therefore the extra delays in 2-cycle scheduling do not directly affect performance. Conversely, a program with short dependence edges tends to lose performance significantly for the opposite reason.

Experimental results show that a significant percentage (around 90%) of the wakeup activities in the scheduler are incurred by short-distance dependence edges within eight instructions. This is primarily because dependent instructions pairs are placed near each other. Also, long-distance dependence edges may not even be observed by the scheduler, or do not directly awaken dependent instructions. Therefore, a technique to focus on short-distance dependent pairs is likely to achieve most benefits.

10.1.2.2 A microarchitecture for macro-op scheduling

Ensuring back-to-back execution of dependent instructions requires scheduling logic to perform at the same rate as they are executed. Macro-op scheduling systematically removes instructions with single-cycle latency from the machine by grouping multiple instructions into macro-ops, and performs non-speculative pipelined scheduling of multi-cycle operations.

A microarchitecture for macro-op scheduling has MOP detection logic located outside the processor's critical path, which 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. MOP formation logic reads MOP pointers and groups the original instructions into macro-ops. During this process, MOP dependence translation is performed for the scheduler to track dependences at a coarser MOP

level. The instruction scheduler performs pipelined scheduling of multi-cycle macro-ops. An issued macro-op accesses the payload RAM, which sequences the original instructions in the original instruction-grained execution pipelines.

There are several issues in determining groupable candidate instructions in the MOP detection process. Macro-op grouping abstracts the original dependences and potentially induce cycles in the data dependence chain. To prevent this, MOP detection logic uses a simple heuristic that detects cycle conditions conservatively. The complexity of the detection logic is also considered. The logic can be pipelined by adopting multiple detection queues that examine instructions independently at multiple points.

Several performance considerations are made. A macro-op of dependent instructions provides the most benefit since it relaxes both scheduling atomicity and scalability constraints simultaneously. A macro-op of independent instructions does not relax scheduling atomicity but can be beneficial since issue queue contention is reduced. A macro-op may degrade performance by unnecessarily delaying instructions due to last-arriving operands in macro-op tail instructions. To prevent this, macro-op scheduling requires a filtering mechanism in MOP detection logic that depreciates the harmful macro-ops.

10.1.2.3 Results

Given the policy of grouping two single-cycle instructions within a 2-cycle scope that captures up to eight instructions, macro-op scheduling groups on average 32.4% of total instructions, which enables 26% ~ 63% of single-cycle instructions to behave as if 1-cycle scheduling is performed. Without issue queue contention, macro-op scheduling achieves on average 97.3% of 1-cycle scheduling performance. Macro-op scheduling tends to be more effective in recovering the performance of the benchmarks sensitive to 2-

cycle scheduling, such as *gap* and *gzip*. With issue queue contention, macro-op scheduling outperforms the base 1-cycle scheduling. The performance gain over 1-cycle scheduling is measured to be on average 1.5% of IPC improvement, which is an 8.5% improvement over the conventional 2-cycle scheduling case.

Several aspects of macro-op scheduling are examined. Limiting the number of source operands of a macro-op to two does not significantly reduce the effectiveness of macro-op scheduling. Regarding MOP detection and formation scope, a 2-cycle scope that captures up to eight instructions achieves most benefits compared with other wider scopes. Pipelining the detection process with 8- or 16-instruction intervals gives most benefits of fully-pipelined detection logic. Also, the detection latency is not critical since MOP pointers are stored in the instruction cache and reused repeatedly. Without filtering out harmful MOPs, macro-op scheduling may lose significant amount of performance benefits. The proposed filtering mechanism is effective in avoiding them.

10.1.3 Macro-op execution

This work is divided into two categories. First, the basic concept of macro-op execution and its microarchitecture is detailed. Then, several aspects of macro-op execution and its performance benefits are studied and evaluated.

10.1.3.1 A microarchitecture for macro-op execution

The range of coarse-grained instructions processing is extended from the instruction scheduling logic to the entire execution pipeline in macro-op execution. Unlike macro-op scheduling that sequences the original instructions in the dispatch stage, macro-op execution moves it down to the execution stage and enables the original instructions grouped in macro-ops to be naturally sequenced as they flow through the datapaths. To

support this, the execution stage has stacked functional units that execute two original instructions sequentially in two consecutive cycles. When multiple macro-ops that contain two instructions each are consecutively issued, the effective execution bandwidth is doubled by fully utilizing two functional units in each execution lane.

For grouping load instructions, MOP detection logic needs to detect cycle conditions through memory dependences. The number of issue slots in the scheduler can be reduced by issuing multiple instructions as a single macro-op without sequencing them, and utilizing idle cycles in select logic. The read ports to the payload RAM and register file can be better utilized by processing multiple requests in groups. The bypass logic can be simplified by exploiting the attribute of macro-ops that force predetermined and restrictive execution of instructions grouped, compared to the conventional case with full forwarding paths and the same effective execution bandwidth.

10.1.3.2 Results

Given the policy of grouping two candidate instructions (single-cycle ALU, control, store address and load instructions) within a 2-cycle scope that captures up to eight instructions in program order, macro-op execution groups on average 48% of total instructions, which potentially enables a 2-wide machine (with 4-wide fetch bandwidth) to work as a 2.63-wide machine with a peak bandwidth of four instructions. Without issue queue contention, narrowing the issue bandwidth down to two instructions reduces performance by an average of 14.5% and 17.8% with 1-cycle and 2-cycle scheduling logic, respectively. A 2-wide issue machine with macro-op execution, which has pipelined 2-cycle scheduling logic, loses 8.2% of the performance of a full, 4-wide issue bandwidth. When issue queue contention is high (32-entry issue queue), 2-wide macro-op execution reaps

the most of the performance (98.8%) of the base 4-wide machine, whereas an equivalent 2-wide machine with 2-cycle scheduling loses 13.2% of the base performance.

Several aspects of macro-op execution are examined. Limiting the number of source operands in a macro-op does not significantly reduce the effectiveness of macro-op execution. This enables an execution lane to achieve increased execution bandwidth without additional read ports to the register file. Regarding MOP detection and formation scope, a wider scope tends to further benefit macro-op execution, although the difference is not significant. The performance difference between 2-cycle and 4-cycle formation scopes is on average 0.7% across the benchmarks. The benefits of additional macro-op types used for macro-op execution are also evaluated. Grouping loads instructions into macro-ops tends to degrade the performance of instruction scheduling, but the benefits from increasing machine bandwidth overcompensate the degradation and hence they tend to positively affect performance.

Macro-op execution exhibits only a marginal performance benefit when implemented on a 3- or 4-wide issue machine because performance benefits for wider machine bandwidth are constrained by fetch bandwidth as well as limited parallelism of programs.

10.2 Future Research

Macro-op scheduling and execution has many aspects and this thesis does not cover the entire range of its design space completely. Also, the basic concept of coarse-grained instructions processing can be extended to wide range of areas besides the core microarchitecture. I describe several future avenues of research.

10.2.1 Macro-op detection and dynamic binary translation

The current macro-op detection is based on a greedy algorithm that searches for groupable instruction pairs without considering performance impact or other factors. Although the filtering mechanism successfully removes not useful or harmful macro-op, it is a dynamic approach and may not be suitable when the information on the dynamic behavior is limited or when macro-ops are cached in a decoded instruction cache or trace cache [38][50].

The complexity of macro-op detection can be significantly reduced by a software approach such as dynamic binary translation on a co-designed virtual machine [53][54][46] because finding dependent operations that can be grouped into macro-ops and checking their suitability is a relatively complex task that requires analysis of data dependences. In this approach, a more judicious and complex detection algorithm can be used to generate macro-ops. This approach may require a co-designed internal ISA specially designed for macro-op scheduling and execution, which improves its grouping efficiency and also facilitates the macro-op formation process to further reduce hardware complexity. Hu and Smith presented an initial work on the co-designed ISA that supports macro-op scheduling and execution [46].

10.2.2 Extending coarse-grained instruction processing to the entire pipeline

Extending coarse-grained instruction to the entire processor pipeline provides additional advantages. First, the instruction fetch bandwidth can be potentially increased, if the ISA efficiently encodes macro-ops into instruction words with fewer bits. Second, the rename bandwidth can be increased because 1) the dependence between macro-op head and tail instructions does not require intra-rename-group dependence checking nor map table access, and 2) the maximum number of source operands in a macro-op is lower

than the number in individual instructions. Therefore, the additional complexity incurred by renaming two instructions in a macro-op should be much lower than naively doubling the rename bandwidth. Third, the dispatch bandwidth can also be increased since two instructions in a macro-op require only a single access to the issue queue, register map table and the payload RAM in the queue stage. Finally, the entries in the reorder buffer can be shared by multiple instructions in macro-ops. To enable this, it is essential to place groupable instructions pairs together in the instruction cache (or some other structures like trace cache [38][50][16]) and to reorder instructions. A downside of this approach is that the original program order is no longer maintained and therefore other mechanisms are required to preserve precise machine state. An initial proposal for enabling coarse-grained instruction processing in the entire processor pipeline, based on a co-designed virtual machine, is presented in [46].

10.2.3 Analyzing the degree of register use

Although macro-op execution is efficient in reducing the hardware complexity involved in handling source operands, it does not reduce the register write ports since the results values should be written back individually at the original instruction level. If the macro-op tail is the only consumer of the result value generated by the macro-op head instruction, register write ports can be also reduced since the head's result is available through the private bypass path that connects the two functional units. To enable this, MOP formation logic should ensure that no other instruction is dependent on the value, which requires the knowledge of the value degree of use [12].

There are several difficulties in this approach. First, it requires global analysis of data dependences. Second, statically analyzing register live-out or the degree of register

use may not be feasible since dynamic control flow changes the degree of register use. Finally, there should be some mechanisms to preserve the precise architectural state for branch misprediction recovery or exception handling, since the register value should be architecturally visible when those conditions occur between the two grouped instructions.

10.2.4 Larger macro-ops

Although processing more instructions in larger macro-ops provides more benefits of further relaxing scheduling atomicity and increasing the size of the instruction window as well as machine bandwidth, dependence chains in the SPEC2K integer benchmarks we tested tend to be short and do not provide many opportunities for grouping more than two instructions. However, compilers may efficiently support larger macro-ops by placing multiple instructions together in a manner suitable for larger macro-ops or generating instructions sequences to maximize the utilization of instruction slots for macro-ops.

10.2.5 Vertically long instruction word

Macro-op scheduling and execution shares some philosophy with native execution of CISC instructions adopted in some x86 implementations such as the Cyrix M1 processor [66], although macro-ops focus on reducing hardware complexity and improving performance rather than high code density and ease of programming that many CISC instruction sets try to achieve. Our proposed approach motivates revisiting the philosophy behind instruction set architecture designs. Does the RISC approach really enable faster and more efficient hardware implementations than the CISC approach? How will CISC instruction sets look if the regularity of operations is maintained? Is it possible to reap the benefits of both worlds?

New instruction set architectures may be designed based on the coarse-grained, macro-op execution model. A possible approach would be similar to the conventional VLIW approach that moves burden of complex instruction scheduling from hardware to compiler. An instruction word or bundle contains multiple instructions selected by the complier. However, an instruction word in this ISA contains a series of instructions (potentially dependent chains) that are executed sequentially, while packing instructions in the VLIW approach requires independent instructions that are executed in parallel. This ISA, *vertically long instruction word*, can incorporate the possible benefits of macro-op scheduling and execution.

10.2.6 Macro-op execution for simultaneous multithreading

A potential application of macro-op execution for wider machine bandwidth would be simultaneous multithreading (SMT) [94][95][20][100], which is a technique that permits multiple threads to execute in parallel within a single processor. The IBM POWER5 and the Hyper-threaded Pentium 4 [49][65] are the examples of SMT processors. An SMT processor usually uses shared instruction queues to collect instructions from the different threads so that the processor core can find both fine-grained parallelism within a thread and coarse-grained parallelism across different threads. When multiple independent threads are running, it is likely that issue bandwidth becomes more critical than when only a single thread (with limited parallelism) is scheduled because the instruction scheduler may be able to find more instructions to issue in each clock cycle; this is the situation where macro-op execution benefits most, since macro-op execution model issues multiple independent strands in an interleaved fashion to increase machine bandwidth while value communication through chains of dependent instructions within a single

thread can be efficiently localized. 264

References

[1] G. M. Amdahl, Validity of the Single Processor Approach to Achieving Large Scale Computing capabilities, in *Proc. of AFIPS Spring Joint Computer Conference*, April 1967.

[2] S. Balakrishnan and G. S. Sohi, Exploiting Value Locality in Physical Register File, in *Proc. of 36th International Symposium on Microarchitecture*, San Diego, December 2003.

[3] R. Balasubramonian et at., Reducing the Complexity of the Register File in Dynamic Superscalar Processors, in *Proc. of 34th International Symposium on Microarchitecture*, 2001.

[4] B. Black and J. P. Shen, Scalable Register Renaming via the Quack Register File, Tech report, *CMuART-2000-01*, Carnegie Mellon University, Pittsburgh, 2000.

[5] M. Bluhm and R. A. Garibay jr., Going Native!, *Microprocessor Report*, vol. 9, no. 12, pp. 17-20, September 1995.

[6] E. Borch, E. Tune, S. Manne and J. Emer, Loose Loops Sink Chips, in *Proc. of 8th International Symposium on High Performance Computer Architecture*, Boston, 2002.

[7] S. Breach, *Design and Evaluation of a Multiscalar Processor*, PhD Thesis, University of Wisconsin-Madison, February 1999.

[8] E. Brekelbaum, J. Rupley II, C. Wilkerson and B. Black, Hierarchical Scheduling Windows, in *Proc. of 35th International Symposium on Microarchitecture*, 2002.

[9] M. Brown, J. Stark and Y. Patt, Select-free Instruction Scheduling Logic, in *Proc. of 34th International Symposium on Microarchitecture*, 2001.

[10] W. Buchholz, The IBM System/370 Vector Architecture, *IBM Systems Journal*, vol 25, No. 1, 1986

[11] D. C. Burger and T. M. Austin. The Simplescalar Tool Set, Version 2.0, *Technical report, University of Wisconsin Computer Sciences*, 1997.

[12] J. A. Butts and G. S. Sohi, Characterizing and Predicting Value Degree of Use, in *Proc. of 35th International Symposium on Microarchitecture*, December 2002.

[13] R. Canal and A. Gonzalez, A Low-Complexity Issue Logic, in *Proc. of 14th International Conference on Supercomputing*, 2000.

[14] B. Catanzaro, *Multiprocessor system architectures: A technical survey of multiprocessor/multithreaded systems using PSPARC, multi-level bus architectures and Solaris (sunos)*. Sun Microsystems, 1997.

[15] *CDC Cyber 200 Model 205 Computer System Hardware Reference Manual*, Arden Hills, MN: Control Data Corporation, 1981.

[16] Y. Chou and J. P. Shen, Instruction Path Coprocessors, in *Proc. of 27th International Symposium on Computer Architecture*, June 2000.

[17] Compaq Computer Corporation. *Alpha 21264 Hardware Reference Manual*, 2000.

[18] *CRAY-2 Central Processor*, http://www.ece.wisc.edu/~jes/papers/cray2a.pdf, unpublished document, circa 1979.

[19] J. L. Cruz, A. Gonzalez, M. Valero and N. P. Topham, Multiple Banked Register File Architecture, in *Proc. of 27th International Symposium on Computer Architecture*, 2000.

[20] G. E. Daddis Jr. and H. C. Torng, The Concurrent Execution of Multiple Instruction Streams on Superscalar Processors, in *Proc. of International Conference of Parallel Processing*, August 1991.

[21] K. Diefendorff, K7 Challenges Intel, *Microprocessor Report*, vol. 12, No. 14, October 26, 1998.

[22] K. Diefendorff, P. K. Dubey, R. Hochsprung and H. Scales, AltiVec Extension to PowerPC Accelerates Media Processing, *IEEE Micro*, pp 85-95, 2000

[23] D. R. Ditzel and D. A. Patterson, Retrospective on High-level Language Computer Architecture, in *Proc. of the 7th International Symposium on Computer Architecture*, 1980.

[24] K. Ebcioglu, E. R. Altman, DAISY: Dynamic Compilation for 100% Architectural Compatibility, in *Proc. of International Symposium on Computer Architecture*, 1997.

[25] K. Ebcioglu et al. Dynamic Binary Translation and Optimization, *IEEE Transactions on Computers*, vol. 50, no 6, pp. 529-548, June 2001.

[26] M. Eng, H. Wang, P. Wang, A. Ramirez, J. Fung and J. Shen, Mesocode: Optimizations for Improving Fetch Bandwidth of Future Itanium Processors, *Workshop on Complexity-effective Design* in conjunction with *29th International Symposium on Computer Architecture*, May 2002.

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

[28] D. Ernst, A. Hamel and T. Austin, Cyclone: A Broadcast-Free Dynamic Instruction Scheduler with Selective Replay, in *Proc. of 30th International Symposium on Computer Architecture*, San Diego, June 2003.

[29] K. I. Farkas, N. P. Jouppi and P. Chow, Register File Design Considerations in Dynamically Scheduled Processors*, WRL Research Report* 95/10, 1995.

[30] K. I. Farkas, P. Chow, N. Jouppi and Z. Vranesic, The Multicluster Architecture: Reducing Cycle Time through Partitioning, in *Proc. of 30th International Symposium on Microarchitecture*, December 1997.

[31] B. Fields, S. Rubin and R. Bodik, Focusing Processor Policies via Critical-path Prediction, in *Proc. of 28th International Symposium on Microarchitecture*, 2001.

[32] B. Fields, R. Bodik and M. D. Hill, Slack: Maximizing Performance under Technological Constraints, in *Proc. of 29th International Symposium on Computer Architecture*, May 2002.

[33] J. A. Fisher, Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Transaction on Computers*, vol. C-30, pp. 478-490, July 1981.

[34] J. A. Fisher, Very Long Instruction Word Architectures and the ELI-512, in *Proc. of 10th Annual Symposium on Computer Architecture*, June 1983.

[35] M. Franklin and G. S. Sohi, The Expandable Split Window Paradigm for Exploiting Fine-grain Parallelism, in *Proc. of 19th Annual International Symposium on Computer Architecture*, Gold Coast, June 1992.

[36] M. Franklin and G. S. Sohi, Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-grain Parallel Processors, in *Proc. of 25th International Symposium on Microarchitecture*, November 1992.

[37] M. Franklin, *The Multiscalar Architecture*, PhD Thesis, University of Wisconsin-Madison, November 1993.

[38] 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.

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

[40] A. Gonzalez, J. Gonzalez and M. Valero, Virtual-Physical Registers, in *Proc. of the 4th International Symposium on High Performance Computer Architecture*, 1999.

[41] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 2nd Ed.*, p 394, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1996.

[42] G. Hinton et al., The Microarchitecture of the Pentium 4 Processor, *Intel Technology Journal*, Q1, 2001.

[43] 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.

[44] 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.

[45] J. S. Hu, N. Vijaykrishnan and M. J. Irwin, Exploring Wakeup-free Instruction Scheduling, in *Proc. of 10th International Symposium on High Performance Computer Architecture*, February 2004.

[46] S. Hu and J. E. Smith, Using Dynamic Binary Translation to Fuse Dependent Instructions, in *Proc. of the 2nd International Symposium on Code Generation and Optimization*, March 2004.

[47] IBM Microelectronics Division, *PowerPC604 RISC Microprocessor User's Manual*, 1994.

[48] Intel corporation, *Pentium Pro Family Developers Manual*, December 1995.

[49] Intel Corporation, *Intel Pentium 4 and Intel Xeon Processor Optimization: Reference Manual*, October 2002.

[50] Q. Jacobson and J. E. Smith, Instruction Pre-processing in Trace Processors, in *Proc. of 25th International Symposium on Computer Architecture*, 2000.

[51] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar and A. Yoaz, A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification, in *Proc. of the 31st International Symposium on Microarchitecture*, 1998.

[52] M. Kagan, S. Gochman, D. Orenstien and D. Lin, *M*MX Microarchitecture of Pentium Processors with MMX technology and Pentium II Microprocessors, *Intel Technology Journal*, 3rd quarter, 1997.

[53] H. Kim and J. E. Smith, An Instruction Set and Microarchitecture for Instruction Level Distributed Processing, in *Proc. of 29th International Symposium on Computer Architecture*, May 2002.

[54] H. Kim and J. E. Smith, Dynamic Binary Translation for Accumulator-Oriented Architectures, in *Proc. of the 1st International Symposium on Code Generation and Optimization*, March 2003.

[55] I. Kim and M. H. Lipasti, Implementing Optimizations at Decode Time, in *Proc. of 29th International Symposium on Computer Architecture*, May 2002.

[56] I. Kim and M. H. Lipasti, Half-price Architecture, in *Proc. of the 30th International*

[57] I. Kim and M. H. Lipasti, Understanding Scheduling Replay Schemes, in *Proc. of 10th International Symposium on High Performance Computer Architecture*, February 2004.

[58] I. Kim and M. H. Lipasti, Macro-op Scheduling: Relaxing Scheduling Loop Constraints, in *Proc. of International Symposium on Microarchitecture*, December 2003.

[59] A. Klaiber, The Technology Behind Crusoe Processors, *Transmeta Technical Brief,* 2000.

[60] 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.

[61] A. R. Lebeck et al, A Large, Fast Instruction Window for Tolerating Cache Misses, in *Proc. of 29th International Symposium on Computer Architecture*, 2002.

[62] M. H. Lipasti, B. R. Mestan and E. Gunadi, Physical Register Inlining*,* in *Proc. of 31st International Symposium on Computer Architecture*, June 2004.

[63] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein and R. Nix, The Multiflow Trace Scheduling Compiler, *The Journal of Supercomputing*, July 1993.

[64] N. Malik, R. Eickemeyer and S. Vassiliadis, Interlock Collapsing ALU for Increased Instruction-level Parallelism, in *Proc. of 25th International Symposium on Microarchitecture*, 1992.

[65] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, Hyper-threading Technology Architecture and Microarchitecture, *Intel Technology Journal*, 6(1), February 2002.

[66] S. C. Mcmahan, M. Bluhm and R. A. Garibay jr., 6x86: The Cyrix Solution to Excuting x86 Binaries on a High Performance Microprocessor, *Proceedings of the IEEE*, vol. 83, no. 12, December 1995.

[67] S. Melvin, M. Shebanow and Y. Patt, Hardware Support for Large Atomic Units in Dynamically Scheduled Machines, in *Proc. of 21st Annual Workshop on Microprogramming and Microarchitecture*, San Diego, California, November 1988.

[68] S. Melvin and Y. Patt, Enhancing Instruction Scheduling with a Block-Structured ISA, in *International Journal of Parallel Programming*, vol. 23, no. 3, 1995.

[69] 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.

[70] T. Monreal, A. Gonzales, M. Valero, J. Gonzalez and V. Vinals, Delaying Physical Register Allocation through Virtual-Physical Registers, in *Proc. of 32nd International Symposium on Microarchitecture*, November 1999.

[71] C. Moore. POWER4 System Microarchitecture. In *Proc. of the Microprocessor Forum*, October 2000.

[72] R. Nagarajan, K. Sankaralingam, D. Burger and S. Keckler, *A Design Space Evaluation of Grid Processor Architectures*, in *Proc. of International Symposium on Microarchitecture*, 2001.

[73] A. Pajuelo, A. Gonzalez and M. Valero, Speculative Dynamic Vectorization, in *Proc. of International Symposium on Computer Architecture*, May 2002.

[74] S. Palacharla, N. P. Jouppi and J. E. Smith, Complexity-Effective Superscalar Processors, in *Proc. of 29th International Symposium on Computer Architecture*, 2002.

[75] G. Papadopoulos and D. Culler, *Monsoon: An Explicit Token-Store Architecture*, in *Proc. of International Symposium on Computer Architecture*, 1990.

[76] I. Park, M. Powell and T. Vijaykumar, Reducing Register Ports for Higher Speed and Lower Energy, in *Proc. of 35th International Symposium on Microarchitecture*, 2002.

[77] 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.

[78] N. Ranganathan and M. Franklin, Complexity-effective PEWs Microarchitecture, *Microprocessors and Microsystems*, 1998.

[79] B. R. Rau, D. Yen, W. Yen, and R. A. Towle, The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions and Trade-offs, *IEEE Computer,* 22, 1989.

[80] R. M. Russell, The CRAY-1 computer system, *Communications of the ACM 21*, No. 1, 63-72, January 1978.

[81] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. Kecker and C. Moore, *Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture*, in *Proc. of International Symposium on Computer Architecture*, 2003.

[82] 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.

[83] H. Sharangpani, K. Arora, Itanium Processor Microarchitecture, *IEEE MICRO*, vol. 20, No. 5, 2000.

[84] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, beta edition, pp. 400-402, McGraw-Hill, 2002.

[85] A. J. Smith, Line (block) Size Choice for CPU Cache Memories, *IEEE Transactions on Computers*, vol. 36, 9, pp. 1063-1075, September 1987.

[86] G. S. Sohi, S. Breach, and T. N. Vijaykumar, Multiscalar Processors, in *Proc. of 22nd International Symposium on Computer Architecture,* pp. 414-425, June 1995.

[87] Standard Performance Evaluation Corporation, http://www.specbench.org.

[88] J. Stark, M. Brown and Y. Patt, On Pipelining Dynamic Instruction Scheduling Logic, in *Proc. of 33th International Symposium on Microarchitecture*, 2000,

[89] Texas Instruments, *TMS320C6000 CPU and Instruction Set Reference Guide*, SPRU189F, October 2000.

[90] D. J. Theis, Special Tutorial: Vector Supercomputers, *IEEE Computer*, pp. 52-61, April 1974.

[91] R. M. Tomasulo, An Efficient Algorithm for Exploiting Multiple Arithmetic Units, *IBM Journal*, vol. 11, pp. 25-33, January 1967.

[92] M. Tremblay, B. Joy and K. Shin, A Three Dimensional Register File for Superscalar Processors, in *Proc. of 28th Hawaii International Conference on System Sciences*, pp. 191-201, 1995.

[93] J. Tseng and K. Asanovic, Banked Multiported Register Files for High-Frequency Superscalar Microprocessors, in *Proc. of 30th International Symposium on Computer Architecture*, San Diego, June 2003.

[94] D. Tullsen, S. Eggers and H. Levy, Simultaneous Multithreading: Maximizing On-chip Parallelism, in *Proc. of 22nd International Symposium on Computer Architecture*, June 1995.

[95] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor, in *Proc. of 23rd International Symposium on Computer Architecture*, May 1996.

[96] United States Patent #6,212,626, *Computer Processor Having a Checker*, A. A. Merchant and D. J. Sager, assigned to Intel Corporation, issued April 3, 2001.

[97] S. Vajapeyam, J. P. Joseph and T. Mitra, Dynamic Vectorization: A Mechanism for Exploiting Far-flung ILP in Ordinary Programs, in *Proc. of the 26th International Symposium on Computer Architecture*, May 1999.

[98] L. Wu, C. Weaver and T. Austin, Cryptomaniac: A Fast Flexible Architecture for Se-

cure Communication, in *Proc. of the 28th International Symposium on Computer Architecture*, 2001.

[99] W. A. Wulf, Compilers and Computer Architecture, *IEEE Computer*, vol 14(8), pp. 41-47, 1981.

[100] W. Yamamoto and M. Nemirovsky, Increasing Superscalar Performance through Multistreaming, in *Proc. of Conference on Parallel Architecture and Compilation Techniques*, June 1995.

[101] A. Yoaz, M. Erez, R. Ronen and S. Jourdan, Speculation Techniques for Improving Load Related Instruction Scheduling, in *Proc. of 26th International Symposium on Computer Architecture*, 1999.