This is to certify that the
dissertation entitled

OPTIMIZATION OF TECHNOLOGY-SCALABLE WIDE-ISSUE SUPERSCALAR MICROPROCESSORS

presented by

JUNWEI ZHOU

has been accepted towards fulfillment
of the requirements for the

Ph.D degree in Electrical & Computer Engineering

Major Professor's Signature

Aug 18, 2006

Date

OPTIMIZATION OF TECHNOLOGY-SCALABLE WIDE-ISSUE SUPERSCALAR MICROPROCESSORS

By

Junwei Zhou

A DISSERTATION

Submitted to
Michigan State University
In partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Electrical and Computer Engineering

2006

ABSTRACT

OPTIMIZATION OF TECHNOLOGY-SCALABLE WIDE-ISSUE SUPERSCALAR MICROPROCESSORS

By

Junwei Zhou

Advances in VLSI fabrication technology have enabled performance improvements in superscalar microprocessors that can exploit the instruction-level parallelism (ILP) and thread-level parallelism (TLP) in programs to achieve high instruction throughput. Overall instruction throughput is determined by the summation across all threads of the product of the clock rate and the instructions per clock (IPC). As submicron process features continue to scale down, however, wire delays compromise the clock rates in centralized processors, and further deepening the processor pipeline could significantly degrade the IPC performance. Coupled with the inconsistency in ILP and TLP across applications, it is increasingly difficult to continue improving overall instruction throughput to meet industry demands.

Multi-core approaches dividing a centralized superscalar processor into multiple separate cores to achieve overall wide issue width while enabling high clock rates. With a fixed hardware budget, the hardware resources of an individual core are reduced, thus minimizing wire delays. However, in multi-core designs, the IPC of the individual thread is compromised, hurting the overall throughput. To maximize instruction throughput a new approach is needed that can simultaneously optimize both the IPC and the clock rate.

This thesis explores the performance bottlenecks in the superscalar microarchitecture and aims to identify process-scalable methodologies for maximizing instruction throughput over a wide range of applications. First, circuit-level techniques are applied to

optimize the critical instruction queue stage of a centralized microarchitecture, significantly reducing the required hardware and associated delays and providing up to 36% improvement in overall instruction throughput. Next, the distribution of hardware resources is explored to identify a performance-optimal compromise between centralized and multi-core approaches. We show that distribution of the processor back-end can reduce the required hardware resources significantly with relatively small IPC degradation. Finally, a new adaptive clustered multithreaded (ACMT) microarchitecture is proposed to enable high performance on both single- and multi-threaded workloads. ACMT allows the allocation of hardware resources to be dynamically modified and provides an overall higher instruction throughput than simultaneous multithreading (SMT) and multi-core processor (CMP) implementations.

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Andrew Mason, for his support to this work during this four years period. His help, advice, and encouragement have been very important for me.

I am grateful to my parents for their concern, patience and understanding throughout my graduate study. I am indebted to my wife, Guoyu Zhu, for her support and encouragement during this un-easy but the most memorable time of my life.

I thank faculty Anthony Wojcik, Peixin Zhong and Nihar Mahapatra for serving on my committee. I thank them for providing directions, sharing their ideas, and answering my questions.

I am privileged to know Jichun Zhang, Prasanna Balasundaram, William Kun, Jian Li, Chao Yang, and Yue Huang during the past four years. I thank them for helping me with the circuit design in this work.

Finally, I want to thank Wireless Integrated Microsystems at University of Michigan for supporting this work.

TABLE of CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 High Performance Processors

Modern high performance microprocessors achieve high instruction throughput by exploiting parallelisms in the applications. The overall processor throughput is proportional to the number of simultaneously running threads and the instruction throughput of individual threads. Microprocessor throughput is ultimately a function of the processor clock rate and the instructions per clock (IPC) for a specific thread.

$$processor\_throughput = \sum_{t} instruction\_throughput_i = \sum_{t} IPC_i \times clock\_rate$$

where t is the number of threads

High throughput microprocessor architectures fall into two primary categories based on how they take advantage of parallelisms in the set of instructions to be executed, which is referred to as instruction level parallelism (ILP). The very large instruction word (VLIW) microarchitecture [3-5] relies on compiler techniques to exploit the ILP in a program. In contrast, the superscalar microarchitecture [1,2] utilizes hardware to make the most of ILP. Since the 1990s, superscalar performance has continued to improve through architectural advancements and significant increases in hardware resources. Currently, superscalar is a standard for high performance microprocessors in a wide range of applications [27,28,32,33,34,86].

Today's superscalar processors overcome control flow and data flow constraints that are inherent in a program using intensive speculation techniques [6-15]. Speculative execution effectively improves the instruction throughput of single thread by predicting

program events and fetching and executing instructions before the events occur. In addition, Simultaneous Multithreading (SMT) [16-20] has been used in superscalar processors to achieve high performance on multiple thread/program workloads. In SMT, several execution threads are multiplexed onto a common set of hardware resources to increases hardware utilization.

Superscalar is a hardware intensive microarchitecture that has benefited significantly from improvements in silicon fabrication technology. Following the trend predicted by Moore's Law, each generation of process technology has approximately doubled the number of available transistors. In combination with architectural improvements, the dramatically increasing hardware budget has allowed superscalar performance to scale along with fabrication technology, maximizing overall processor throughput by putting more and more hardware resources on a microprocessor chip.

1.2 Challenges

The superscalar microarchitecture is deeply pipelined to support high clock rate. Until recently, the increasingly available hardware resources have been harnessed to increase the depth of processor pipelines, i.e., the number of pipeline stages and the size of individual pipeline stage. This increases the number of in-flight instructions to exploit program parallelism. Conventional superscalar processors implement centralized hardware in each pipeline stage, and multiple instructions access the common hardware simultaneously. The centralized design typical has high hardware utilization and high efficiency when the hardware resources are relatively low. However, as more and more hardware resources have been utilized to scale performance, hardware utilization of the centralized resources has decreased with successive generations of fabrication

technology. A plot of delivered processor performance versus the number of implemented transistors demonstrates that the efficiency with which the available hardware resources are utilized has declined over time. As a result, many recent efforts have turned to enhancing the width of the pipeline [26-28] to boost performance by increasing the number of instructions that are fetched and executed concurrently.

Wire delays present another challenge to scaling superscalar performance with process technology. In new technologies, wire delays increase relative to the number of transistors [79,80] and have become a dominate delay source. Coupled with the increased size of centralized hardware units, the pipeline critical path delay has increased, resulting in slower clock rates and lower instruction throughput. Combating this problem by further deepening the pipeline would introduce pipeline bubbles that significantly compromise the IPC performance and therefore the overall instruction throughput [31]. On the other hand, multi-core processor approaches, including Chip Multi-processors (CMP) [29,91,92] and Chip Multi-threading (CMT) [26,28], implement multiple small cores to achieve overall wide issue width while enabling high clock rates. A multi-core processor can be viewed as dividing a centralized superscalar processor into separate cores that share part of memory hierarchy. With the same hardware budget, the hardware resources of an individual core are reduced, thus minimizing wire delays. However, in multi-core designs, the *IPC* of the individual thread is compromised, hurting the overall throughput. To maximize instruction throughput with the available transistor budget, a new approach is needed that can simultaneously optimize both the IPC and the clock rate.

1.3 Related Work

A high performance superscalar processor has many pipeline stages. Their hardware implementations have different level of circuit complexity and scale differently with technologies. Many researches optimized critical pipeline stages to remove the system bottleneck, achieving high performance. Optimization of individual stage has minimal affect on other stages, and therefore can retain most advantages of a centralized design.

The critical pipeline stages includes the instruction queue [44-60], the register file [35-42], the memory disambiguation logic [73-76], the instruction cache [49,87], and functional units [85,86]. Some work improved the delay by improving hardware utilization and reducing the overall hardware resources [36,37,38,44,57]. Other work reduced the hardware resources on the critical path by dividing a large centralized structure into multiple small segments in hierarchy [37,47,52,60], in pipeline [51,54], or in parallel [32,39,45,74,76]. In addition, new circuit techniques [40,41,59] were used to counter the increasing delay with technologies.

Instead of optimizing individual pipeline stages, some techniques distribute part of a centralized processor to support technology scaling. Multiscalar architecture [22] and Trace Window architecture [23] contain multiple processing elements which have their own instruction flow control, instruction queue, register file and functional units. The two architectures differ in the organization and communications of the processing elements. Both use compiler techniques to divide programs into tasks which are assigned to processing elements. A task is a sequence of code that is executed speculatively. When control or data prediction prove to be wrong, the whole task and its results are discarded or re-executed.

Multicluster architecture [24] only distributes the register file, instruction window and functional units. Each cluster is assigned a subset of the architecture registers. The register file of one cluster can be accessed by instructions executed in another cluster. Compiler support is used to reduce the portion of these instructions. Zyuban [25] proposed a version of the multicluster architecture, tailored to achieving high energy efficiency. Each cluster is provided with a local instruction queue, a local physical register file, a set of execution units, local memory disambiguation units, and one bank of the interleaved data cache. The instructions are dispatched to clusters dynamically based on data dependency.

Chip Multiple Processor (CMP) [29] and Chip Multi-Threaded (CMT) processor [26,28] implements multiple cores to exploit thread-level parallelism. A CMP core is assigned to a thread and is not available to other threads. Multiple cores can share some resources, such as the memory controller, off-chip bandwidth, and the L2 cache, to improve the utilization of these resources. Stanford Hydra processor has four MIPS-based processors on a single chip [91]. AMD, Intel and Fujitsu released dual-core processors [92, 27]. CMT support many simultaneous thread execution via a combination of support for CMP and SMT. It is similar to CMP, but each core run multiple threads simultaneously. In 2001, IBM introduces the dual-core POWER-4 processor and recently releases their second generation CMT processor, the POWER-5 [28], in which each core is a two-issue SMT. In 2003, Sun Microsystems released CMT processor Jaguar [93] that implements two SPARC on the chip. In 2004, eight cores Niagara processor [26] was introduced with an overall 32 issue width.

1.4 Projet Objectives

Optimization of the critical pipeline stage can improve processor performance by removing the performance bottleneck with minimal effect on other pipeline stages. In addition, targeting individual stages allows leveraging the unique characteristics of the stages to find the optimal design under various design constraints. One objective of the thesis aims to improve the design of a critical pipeline stage of a wide-issue superscalar processor and show the effectiveness of the resulting design strategy.

Optimization on each of many critical stages can be applied to keep improving performance, but it requires a significant amount of design effort. More importantly, the design space of an individual stage is greatly limited by other pipeline stages. Processor distribution provides a more scalable solution. It horizontally divides a wide centralized pipeline into parallel distributed pipelines, which can be viewed as applying a common optimization approach on each pipeline stage. This strategy requires less design effort than exploring the optimal solution of each pipeline stage. The hardware of additional pipeline stages can be easily added as a module to the available design to provide scalable processing resources. However, distribution of some pipeline stages may have very limited improvement on the hardware complexity but could cause significant performance degradation.

This thesis aims to create a link between two design strategies – optimization of critical pipeline stages and the distribution of a centralized processor – to explore the optimal solution by thorough analysis at both the circuit level and the architecture level. An effective and scalable approach will be applied to distribute the hardware on a critical stage, and then extend the hardware distribution scheme to other stages, eventually

resulting in a distributed processor. To show the design tradeoff and the impact on overall performance (instruction throughput), the affect of hardware distribution on circuit delays and the IPC performance are analyzed simultaneously. Based on the analysis, an optimal hardware distribution to maximize the overall performance is identified.

1.5 Thesis Organization

Chapter 2 describes the pipeline and hardware of a speculative execution superscalar processor. Two key circuits, multi-port SRAM and CAM, that are critical to exploit ILP but refuse to scale with technologies are analyzed. Chapter 3 presents the optimization of the instruction queue. Design at both the architecture and the circuit level is described, and the overall performance improvement is discussed. Another optimization approach that potentially leads to new instruction queue designs is also described. Chapter 4 explores hardware distribution of other pipeline stages. Based on obtained results, an Adaptive Clustered Multithreaded (ACMT) microarchitecture is proposed to achieve higher performance than both SMT and CMP processors. The conclusions and thesis contributions are summarized in Chapter 5.

2 POTENTIALS AND LIMITATIONS OF WIDE-ISSUE SUPERSCALAR PROCESSORS

The advances in VLSI technology support wider-issue superscalar processors to achieve high instruction throughput. More silicon area and smaller feature size allow storing a larger number of in-flight instructions in a processor pipeline to exploit the Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP) in applications. This chapter describes the pipelines of a speculative superscalar processor and their hardware implementations.

2.1 Microarchitecture Background

The pipeline of a speculative superscalar processor is shown in Figure 2.1. Instructions are fetched from the instruction cache and a trace cache that provides high bandwidth for the backend of the pipeline.

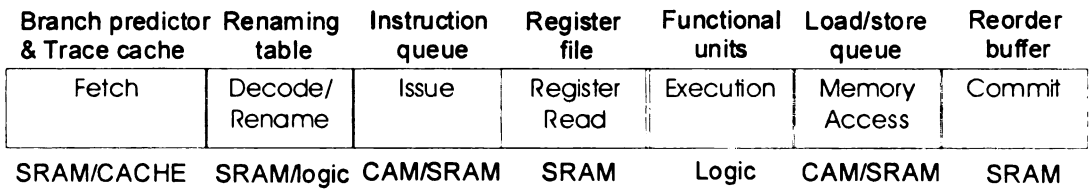


Figure 2.1. A basic superscalar pipeline.

The fetched instructions are sent to the pipeline where they are executed speculatively before the branch outcomes are available. First, the operands of fetched instructions are renamed from architectural registers to physical registers to remove false data

dependencies. The physical registers are used in the rest of the pipeline to track the data dependence between instructions. A register renaming table is used to keep the register mappings.

The renamed instructions go to a buffer where they will be selected for issuing depending on their operand availability. The instruction will request issuing with no latency if both operands are available; otherwise the instruction will stay at the queue waiting for the required operands. In a deep pipelined processor, the issue logic schedules instructions speculatively. The scheduler does not wait until the required data is available in the register file before issuing the instructions, otherwise there would be pipeline bubbles that cause a severe performance penalty. Instead, the issue logic assumes the data will be available in a fixed amount of cycles depending on the type of operations. The issue logic sends instructions for execution based on this delay, assuming the issued instruction can get the data directly from functional units through the bypass network immediately after the results are generated.

The two types of data dependencies between instructions are register and memory dependencies. Memory dependencies occur when the addresses referenced by a load instruction are the same as the preceding store instructions. While register dependencies are known at the instruction scheduling stage, memory dependencies cannot be resolved until the address computation is finished in the execution stage. Speculative execution can be applied to resolve memory dependence and expose the parallelism hindered by this ambiguous dependence [73,74]. Execution of a load does not wait until all the previous store addresses are calculated. Instead, estimation on whether the load has any true dependence is performed. The load may be allowed to execute and obtain memory

data speculatively before an ambiguously dependent store executes. Studies have shown that the speculation has a very high success rate [73]; therefore, performance is improved as loads are executed much earlier than they would if they had to wait for ambiguous dependences to be solved.

After instructions are executed, their results are saved in a physical register. Due to branch misprediction, the speculatively executed instructions might be at the wrong path and discarded later when the misprediction is discovered. The commit stage is to implement the sequential execution model when the actual execution is non-sequential.

Each register in the register file has four states [30]: (a) architectural register; (b) renamed register with invalid value, (c) renamed register with valid value, (d) free. When a decoded instruction includes a destination register, a free register from the register file is allocated for the destination register. The state of the allocated register is then set to be a renamed register with an invalid value. After the instruction finishes execution, the generated value is written into the register whose state is changed to be a renamed register with a valid value. Because this instruction could be in the mispredicted branch, its associated register remains in the renamed register state. After all of its previous instructions have been finished and no branch prediction occurs, the register is set to be the architectural register at the commit stage. The old architectural register is reclaimed for future register renaming.

2.2 Performance Potentials of Wide-Issue Width Processors

Prevalent speculative execution techniques throughout the pipeline greatly increase the hardware utilization, so that the performance can benefit from a wide issue width. The IPC vs. processor issue width was simulated on five SPEC integer benchmarks and six

floating point benchmarks. The processor configuration described in Chapter 3 is used in the simulation and the simulated results are shown in Figure 2.2. The issue window is assumed to be unlimited for all issue widths to show the exploitable parallelism in applications. A four-issue processor is 70% faster on average than a two-issue processor. The IPC can be increased by additional 38% with an 8-issue processor. Due to the limited ILP in a single program, further increasing the issue width to 16 only improves the IPC by an average of 10% over an 8-issue processor. However, the mesa benchmark still sees a 25% IPC improvement with a 16-issue process.

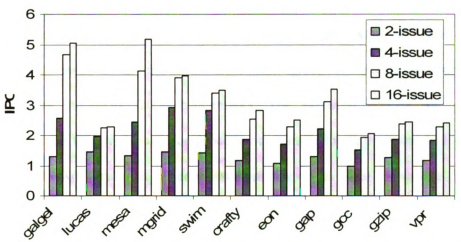


Figure 2.2. IPC performance vs. issue width on SPEC 2000 benchmarks (five floating point benchmarks on the left side; integer benchmarks on the right side).

A superscalar processor with large issue width has a high probability of leaving many of the pipeline stages idle due to control and data dependencies. Therefore, the benefit of implementing wide-issue superscalar is diminishing for single thread/program. With minimal hardware modification, a wide-issue superscalar can process several independent instruction streams simultaneously. In such processors, all the pipeline backends are

shared by all threads/programs, substantially improving the hardware utilization. All decoded instructions are saved in the same instruction queue which dynamically schedules instructions from any threads/programs to the processor execution units. If some threads stall due to long memory latency or branch misprediction, the instructions in un-stalled threads will be selected and executed. By exposing instruction-level and thread-level parallelism, processor performance is closely related to the issue width and the amount of hardware units when there is sufficient parallelism available. Superscalar processors that support such multithreading are called Simultaneous Multithreading (SMT). The wide-issue processors in this thesis are SMT unless otherwise specified.

The overall IPC vs. the processor issue width for multi-program workloads are simulated. The results are shown in Figure 2.3. Multiple groups are simulated for each combination of issue width and workloads. Each group contains programs selected randomly from the SPEC 2000 benchmarks. The averaged IPC of all groups is normalized by the IPC of a four-issue processor running a single program. In Figure 2.3, increasing the workloads of a four-issue processor to four programs improves performance only slightly because the performance is limited by the processor's issue width. An eight-issue processor has 88% higher performance than a four-issue processor on four-program workloads. A 167% improvement can be achieved with a 16-issue processor.

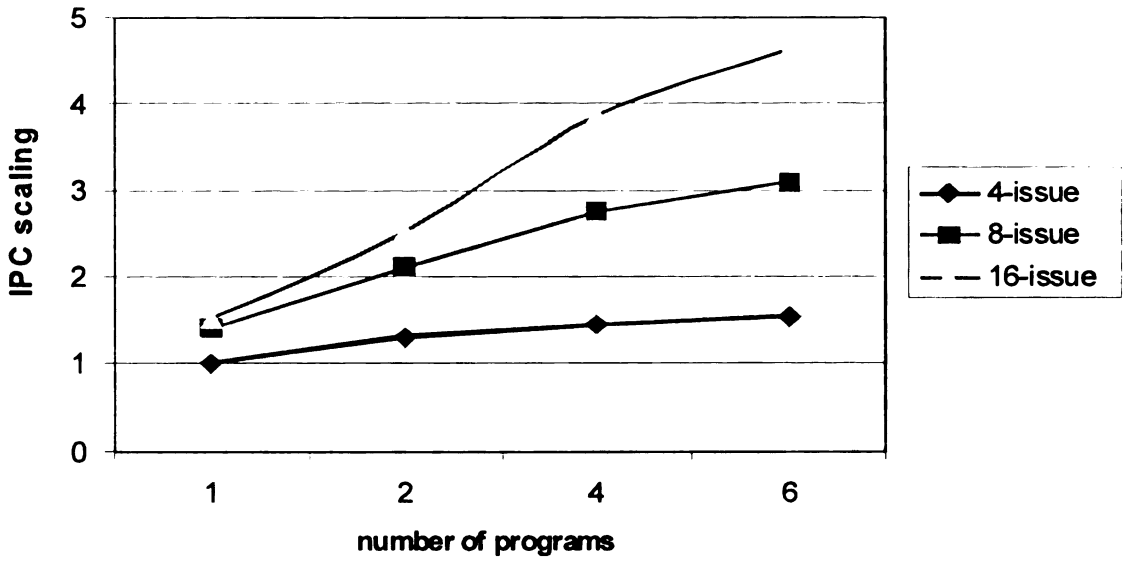


Figure 2.3. IPC speedup of wide-issue processor for multi-program workloads.

2.3 *Basic Hardware in Superscalar Processors*

A wide issue processor requires a large instruction window to expose parallel instructions and efficiently utilize the wide issue bandwidth. Hundreds of in-flight instructions are searched, executed and remain in the pipeline before committed. A high performance processor typically employs many memory circuits (SRAM, CAM) in most of the pipeline stages shown in Figure 2.1. The SRAM stores the history behavior and results of programs, which are used for speculative issue/execution. The CAM provides fast search ability to locate the candidates among hundreds of in-flight instructions in the pipeline.

The SRAM/CAM in wide-issue processors typically has multiple ports that support simultaneous access by multiple instructions. The number of ports in some key units is proportional to the issue width of the processor to avoid a performance penalty due to port conflictions. An eight-issue processor, for example, requires a 24 port register file (2? read ports and 1? write ports) so that up to eight instructions can fetch operands

and write results to the register file without access conflictions. Considering the bypass networks from memory and some special function units to the register file, even more ports are required [86].

2.3.1 SRAM

In a deep pipelined speculative execution processor, SRAM stores not only the current processor state, but also the history of instructions behavior. They are accessed by addresses, such as the architecture register name, the physical register name and the instruction address.

2.3.1.1 Delay Analysis

A typical multi-port SRAM cell is shown in Figure 2.4. The cell has dedicated read and write ports that support write and read operations in the same cycle. The read port uses dynamic NAND to avoid data corruption during the read operation. The SRAM is assumed to have one word each row, so there is no column decoder or column MUX on the bitline. A typical application of such SRAM is the register file in which bitlines are corresponding to the data bits in the data path. The total delay of the SRAM consists of the delay of the wordline decoder, the bitline, the sense amplifier, and the output buffer.

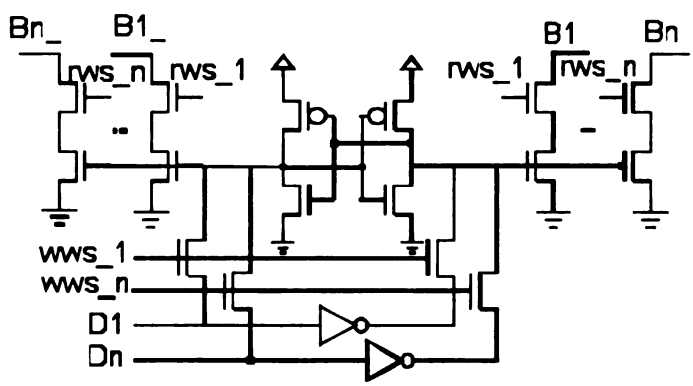


Figure 2.4. A multi-port SRAM cell.

Because the signal on the bitline is a small analog signal that cannot be reduced by pipelining, we assume an entire pipeline stage for the bitline read/write operation. The memory access can be pipelined in two stages with wordline decoding in one stage and bitline read/write in the second stage. Therefore, the bitline delay is a critical path delay that increases as technology scales.

Read access determines the critical timing of the SRAM. The selected cell discharges one of the differential bitlines that are both precharged to VDD. The small signal on the bitline is amplified by the sense amplifier that converts the small swing input signal to a full swing CMOS signal. The latch output is buffered by an inverter chain that drives on the data bus. To provide a good margin to process variation and noise, the sense amplifier is enabled after a certain voltage swing at the input of the sense amplifier has been established. Therefore, the read delay is the sum of the delay to establish the required voltage swing on the bitline, the delay of the sense amplifier, and the buffer delay of driving the data line.

Reading of a very large array is very slow due to a large bitline load and wire delay. The voltage swing at the sense amplifier input is determined by the threshold variations in the crossed coupled inverter. The dominant source of threshold variations in closed space transistors in deep submicrometer geometries is the random fluctuation of the channel dopant concentrations [78]. This portion of the threshold mismatch (about 50mv) remains constant with process scaling.

The layout size of the SRAM is limited by wires. A possible layout of a four-port SRAM cell is shown in Figure 2.5. The bitlines are shielded by the power supply and ground to reduce coupling noise on the bitlines. The cell width is proportional to the

number of bitlines (twice the number of ports for a differential bitline design); the height is proportional to the number of word select lines (equal to port number). Therefore, as the number of ports increase, the size of the memory array grows quadratically and the bitline length grows linearly.

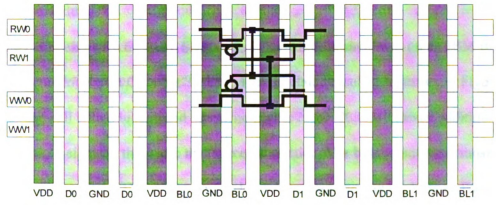


Figure 2.5. A four-port SRAM cell layout.

2.3.1.2 Simulation Results

A large portion of the read access delay establishes the voltage swing before it is amplified. This delay increases linearly with the load of the bitline. Therefore, given a fixed SRAM capacity, read access delay increases linearly with the port number. Furthermore, the demands on SRAM capacity keeps increasing to utilize the available issue bandwidth, therefore, the access delay grows more than linearly with the issue width of a processor. As a result, the performance improvement by high issue bandwidth is offset by larger access delay that hurts the clock rate.

The relationship between the port numbers and the issue width varies with the specific units. For example, the register renaming table needs to rename *2*IW* source

operands and *IW* destination operands each cycle, therefore it requires a total of *3*IW* ports to support *IW* issue rates. For the instruction queue, a SRAM stores the destination operands. It needs *IW* write ports to receive the destination operands of new instructions, and *IW* read ports to send out the destination operands of issued instructions. In the following delay simulation, an SRAM entry has the same number of read and write ports as the issue width.

The read delay of a 64-entry SRAM vs. issue width is simulated on an IBM 0.13μ CMOS technology with Cadence Spectre. The simulated results are shown in Figure 2.6. The storage cell with one read and one write port is 2.3μ X 1.9μ . Each additional port increases the height by the width and space of metal-1. The bitline isolation transistor is turned on after a 10% VDD voltage swing has been established on the bitline. The impact of non-ideal clock generation is ignored, that is, the clock timing of the sense amplifier and isolation transistors is perfectly controlled.

The delay of the sense amplifier remains constant due to the isolation transistor. The delay on the bitline increases significantly with the issue width. With a fixed SRAM size (the number of entries), the read delay of a 16-issue processor is more than twice that of a four-issue processor. Considering the effect of wire delay which grows quadratically with wire length, the delay grows faster for a larger SRAM structure (Figure 2.7). For a 256-entry SRAM structure, increasing the issue width from 4 to 8 slows down the SRAM by 90% due to the extra ports; this ratio grows to 118% for a 512-entry structure. It is worthy to note that the situation gets worse as technologies scale.

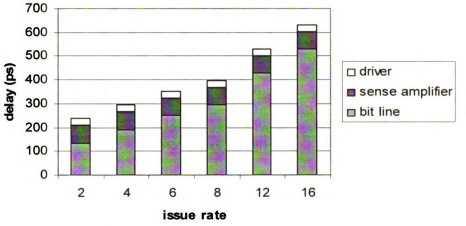


Figure 2.6. SRAM delay vs. the issue width (64 entries).

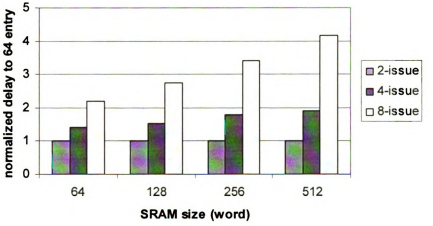


Figure 2.7. SRAM delay vs. SRAM size (number of entries).

With a fixed SRAM size, reducing the SRAM ports improves the SRAM speed. Some techniques have been designed to reduce the SRAM ports. A large SRAM, for example, is divided into a number of address interleaved small arrays. For each access, only the addressed array is activated. The addressed array has shorter bitlines, and can be accessed faster than the whole SRAM. In addition, the address interleaved memory has fewer ports but still maintains the same peak bandwidth as the original SRAM.

A multilevel bitline hierarchy partitions a single long bitline into small segments using additional metal layers [77]. A large SRAM is divided into a number of identically sized small arrays (shown in Figure 2.8). Each array is framed by a local bitline and the local sense amplifier. Each small array has a local bitline whose output is connected to the global bitline by nMOS drivers. Global bitlines are precharged in the same way as the local bitline in each cycle. If any of the outputs of the local bitline is high, the nMOS drivers turn on and discharge the global bitline. Additional metal layers are used to implement the global bitline on the top of the bitline, but there is an area penalty for the global bitline.

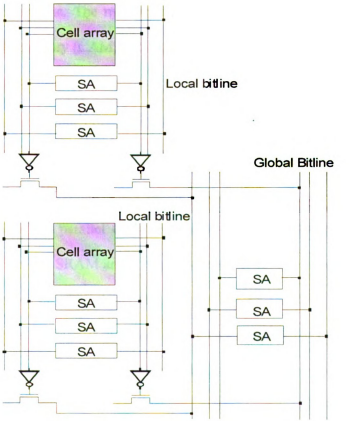


Figure 2.8 Hierarchical bitlines.

2.3.2 *Content-Addressable Memory*

The SRAM is used for data storage and lookup based on a known address. Inside a high speed processor, some data to be sought is associated with a known keyword rather than a known address. The known keyword is compared against previous stored keywords to reference other data or trigger actions. For example, the original operands of newly fetched instructions are renamed to reference the physical register. A physical register designator contains a keyword that is associated with a particular instruction. The exact instruction locations in each hardware unit are normally not known. The keywords, which are called tags, are used to search for the instructions among hundreds of in-flight instructions in the pipeline. The memory that performs this type of function is called content-addressable memory (CAM) [82-84].

2.3.2.1 *Delay Analysis*

The CAM uses the storage circuits of the SRAM but allow the access of data through a matching mechanism. There are three types of CAM architectures [82] varied by the matching mechanism: bit-serial, word-serial, and fully parallel. High performance processors use only fully parallel CAM to achieve minimum latency. A fully parallel CAM array is similar to a SRAM array, each row containing n-bit tags that are compared to the incoming search tag. The stored tags in each row are generally different but some rows could have the same tags. If there is a match, then a match line of the corresponding row sends a match signal to the encoder that generates the physical address of the matched row in the array. In most cases, a CAM is used with a SRAM, the match line is used directly as the word select line of the SRAM, and the encoder is not needed.

A multi-port CAM is shown in Figure 2.9. It contains a SRAM and extra search ports. A search port corresponds to a comparator that compares the data stored in the SRAM cell and the data on the tag line. The match lines are precharged to high. If the stored data is different from the tag, the corresponding match line is discharged to indicate a mismatch. Otherwise the match line remains high. Each search port associates with a match line. All of the match lines of a CAM cell are combined with an OR gate; the output of the OR gate is high if there is a match in any of the search ports.

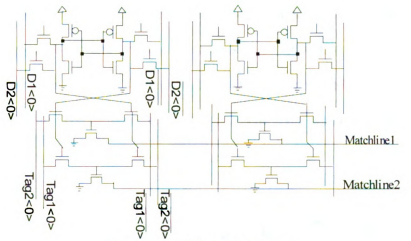


Figure 2.9. A CAM cell schematic

CAM has three types of operations: read, write, and search. The read and write operations are the same as the SRAM. The CAM in Figure 2.10 has shared read/write ports. The search operation consists of three sequential actions as follows:

- Searched tag drive: The searched tag is driven to the CAM tag line.
- Tag match: match lines are discharged when there is any mismatch bit between the searched tag and the stored tag. Match line results are evaluated after worst discharge time.

- Match-line OR: match results are combined to indicate if there is a match in any of the match lines.

Because tag drive and tag match delays are mainly wire delays, they do not scale with technologies. In addition, the search results are usually used to access the SRAM. Therefore, search is the critical timing operation in a CAM.

A layout of a two-port CAM cell is illustrated in Figure 2.10. The read/write port is the same as the SRAM write ports. The match line is in parallel with the word select lines. The height of a CAM is determined by the number of word select and match lines. Notice that match lines impede the implementation of high density CAM. In a SRAM, multiple words can be in one row and the access to an individual word is controlled by column decoders. In a CAM, the match lines can not be shared between words. The match lines have to take different rows or a different layer, making it difficult to trade array width for height like a SRAM array.

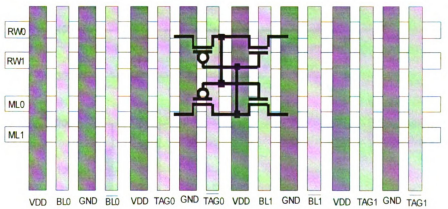


Figure 2.10. A CAM cell layout.

2.3.2.2 *Simulation Results*

The delay of a 64-entry CAM array was simulated on an IBM 0.13μ technology. The results are shown in Figure 2.11. The number of read/write and search ports assumes scaling with the issue width. The basic 1-port CAM cell is 2.3μ X 3μ. The size of the comparator in the CAM cell is 4X the minimum transistor size to reduce the match line delay. Increasing the port number raises the length of the tag lines, width of the match lines, and the number of OR gate inputs. Multiple smaller gates are used to implement the OR gates when it has more than four inputs.

The tag line delay is largely caused by the driver delay and only increases slightly with longer tag lines. However, there is significant increase in the match line delay. The match line speed is limited by the discharge transistors of the mismatch bits. In the worst case, only one bit is different, and the match line is discharged through only two serial transistors. Large OR gates are required for wide issue width to combine the search results.

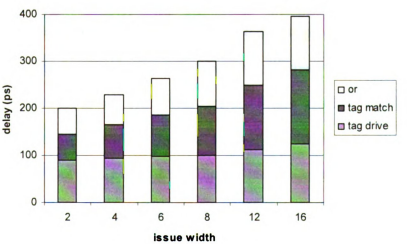


Figure 2.11. CAM delay vs. issue width

Notice that CAM is an energy intensive structure. The comparator in Figure 2.9 is a dynamic NOR gate that pulls down the match line when any mismatch bits are active. This comparator achieves a small delay. However, most comparisons in a CAM result in a mismatch; a significant amount of energy is wasted by driving searched words to these entries and charging/discharging match lines every cycle. A NAND type comparator discharges the match line only when there is a match, consuming less power. Unfortunately the delay of a NAND comparator is proportional to the width of the word, and is not used in high speed processors.

2.3.3 Delay Impact on Performance

Section 2.2 showed that many applications benefit significantly from wide-issue processors. However a speculative wide-issue superscalar requires large SRAM/CAM capacity and ports to exploit the available issue bandwidth. This significantly increases the delay of a processor. If we keep the clock rate scaling with technology, more pipeline stages are needed, which would offset the performance improvement achieved with a higher issue width.

The revised IPC performance was simulated with the same processor configuration as in section 2.2. It assumes the four-issue processor issues dependent instruction consecutively, while there is one clock delay for eight-issue processors and two clock delay for 16-issue processors between the executions of two dependent instructions. The simulated results are shown in Figure 2.12. The revised IPC of a 16-issue processor is even lower than a four-issue or eight-issue processor for a single program workload. The 16-issue processor has noticeable performance improvements only for more than two threads/programs workloads. In fact, the results do not take into account the increased

mis-speculation penalty which further degrades the performance of wide-issue processors.

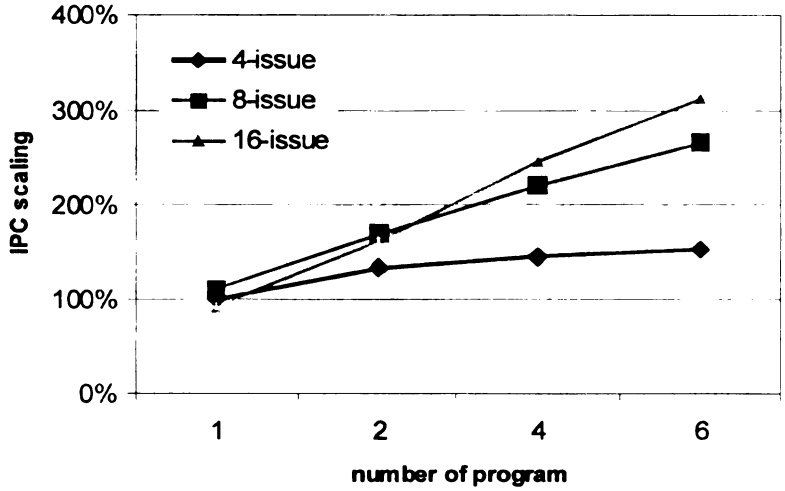


Figure 2.12 Revised IPC performances for multi-program workload.

Therefore, a fundamental issue in wide-issue processor design is to mitigate the scaling of delay to the issue width in heavy-ported SRAM/CAM structures. Approaches can be generalized into two dimensions: 1. reduce memory size. 2. reduce port number. The design space based on these two dimensions can be described in a tree shown in Figure 2.13.

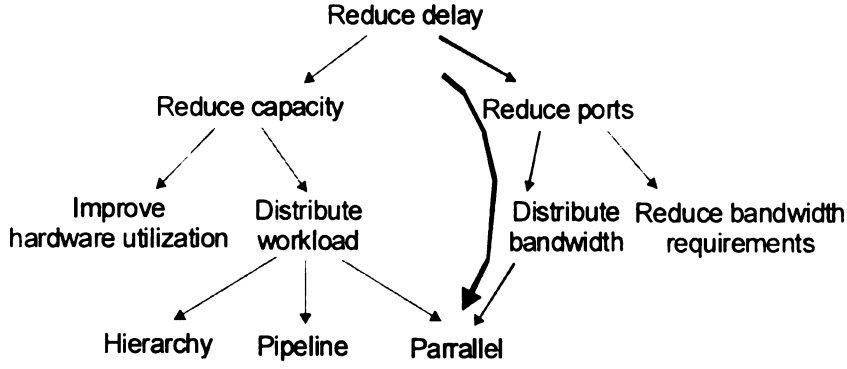


Figure 2.13. Optimization space to reduce delay of the SRAM/CAM structure in a wide-issue processor.

We can improve the hardware utilization to reduce capacity with a minimal performance penalty due to less hardware resources. We can also distribute the whole workload into multiple segments. The segments could operate in a hierarchy which has the smaller segment on the critical path, so that the delay depends only on the size of the smaller segment. Multiple segments can run in a pipeline; only the segment in the first stage exists on the critical path. Or each segment operates independently and can be accessed faster than putting them together. In the second dimension, we can reduce the amount of port accesses to decrease the data bandwidth requirement. Similar to the idea of distributing workloads, the total bandwidth is distributed to multiple smaller units, which operate in parallel to provide higher bandwidth.

Many researches reduce the size/port of the multi-port SRAM/CAM on the critical pipeline stages, including the instruction queue, the register file, and the memory disambiguation logic, to remove the system bottleneck. A W-issue N-core processor distributes a centralized W-issue core into N parallel cores, and each core has W/N issue rate. The size and ports number of the SRAM and CAM circuit is reduced by a factor of N accordingly.

3 INSTRUCTION QUEUE DESIGN & OPTIMIZATIONS

The instruction queue (also called the issue buffer in some literatures) is a critical component of out-of-order issue logic that determines the degree of ILP which can be exploited within a superscalar processor [43,47]. After new instructions are fetched from the instruction cache, instruction operands are renamed to remove false data dependencies. The renamed instructions are inserted into the instruction queue where they wait until their operands become ready. The ready instructions compete for issue slots. The issued instructions broadcast their status to all entries of the queue to update the operand status of dependent instructions. When both operand values are available, the waiting instructions wake up and request issuing at the next clock. To achieve high performance, the instruction queue is required to wake up and issue instructions with minimal latency.

This chapter focuses on the optimization of the instruction queue for wide-issue superscalar processors. A conventional instruction queue is described in section 3.1. Our new banked instruction queue is introduced in section 3.2. Simulation results and summary are provided in section 3.3 and 3.4. Another new optimization approach on the instruction queue is presented in section 3.5. The chapter summary is in section 3.6.

3.1 A Centralized Instruction Queue

A conventional instruction queue consists of two CAMs, a SRAM, and select logic, as shown in Figure 3.1. The CAMs store the source operand tags of waiting instructions and the SRAM stores destination operand tags. For every issued instruction, its destination tag is read from the SRAM and sent to the comparator input of the CAM. The

CAM associatively searches for source tags that match the destination tag and sets the ready bit when there is a match. If both source operands of an entry are ready, it sends a request to the select logic for issuing. When the entry wins an issue slot, its destination tag is read from the SRAM and broadcast to the CAM to wakeup dependent instructions in the next cycle.

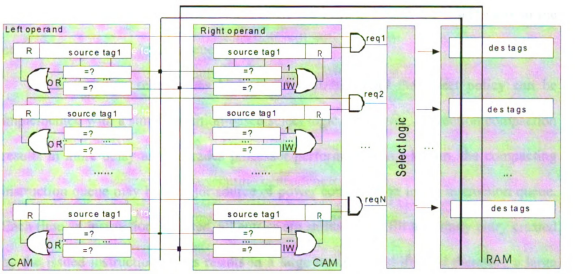


Figure 3.1. A conventional instruction queue.

Select logic selects instructions for issue from a pool of ready instructions in the instruction queue. The oldest-first selection policy is mostly implemented in the select logic. When there are more ready instructions than the available issue slots, the oldest instructions are prioritized over the younger instructions. The instruction wakeup and selection process exist in the critical path of the pipeline. They constitute a closed loop in which delay limits the speed of executing dependent instructions.

There are two types of instruction queues implementing the oldest-first policy: compacting and non-compacting. In the compacting instruction queue, the top entries in

the queue have the highest priority. New instructions are inserted from the bottom of the queue. In every cycle, the empty entries created by the issued instructions are filled by the following instructions. Instructions that occur earlier in the program order occupy the top entries in the queue and have higher priority than the later instructions. In the non-compacting queue, the entries that are created by the issued instructions are not filled immediately. Instead, a head and tail pointer is used to indicate the start and the end of the queue. When the instruction pointed to by the head is issued, the head points to the next non-empty entry in the queue.

The compacting scheme has high queue utilization and the select policy can be implemented with a simple position-based select logic. However, instruction compaction results in more delay and degrades processor performance. In addition, the compacting instruction queue may be a major source of power consumption in the instruction queue. Each time an instruction is issued, all entries are shifted up to fill the empty entry created by the issued instructions. This results in a large number of shifts and therefore a large amount of power dissipation.

The non-compacting queue doesn't need to shift each entry every cycle, and therefore has much lower power dissipation. The head always points to the oldest instruction in the queue. However, because the head moves along the queue, more complicated select logic has to be used. Bradley [94] et. al. proposed Cyclic segmented prefix (CSP) circuits to implement the oldest-first selection policy for the non-compacting instruction queue. However, it needs sequential add operations which have more than linear increases in delay with the issue width. Because power has become the major design limitation, we only discuss non-compacting instruction queue in this work.

As the processor issue width continues to increase to achieve higher performance, more memory ports are required in the CAM and SRAM. As discussed in Chapter 2, the delay and area of SRAM and CAM is proportional to their port number; it is getting more difficult to keep the delay and power consumption under design constraints.

3.2 A Banked Instruction Queue Design

A banked instruction queue has been designed to reduce the CAM/SRAM port number. The centralized CAM and SRAM is divided into N banks shown in Figure 3.2. Each CAM bank can issue up to M ($M<IW$) instructions in parallel but a total of IW instructions are granted for issue in one cycle. Because a CAM bank issues only M instructions simultaneously, M instead of IW write ports are needed to fill the entries emptied by the issued instructions. Similarly, the SRAM needs M write ports to accept new instructions and M read ports to read the destination tags of issued instructions.

The decentralized instruction queue needs communication between banks because some instructions may have its operands located in different banks. The CAM of each bank has M search ports, among which IW/N ports are used for the issued instructions at the same bank, i.e. the tags of issued instructions from the same bank are sent to these ports to search for the dependent instructions. The other $NC=M-IW/N$ ports are used as the communication ports to receive destination operands from other banks. One communication port requires a pair of global tag lines that are shared by all banks. The global tag lines are active only when the dependent instructions are in different bank from the producer.

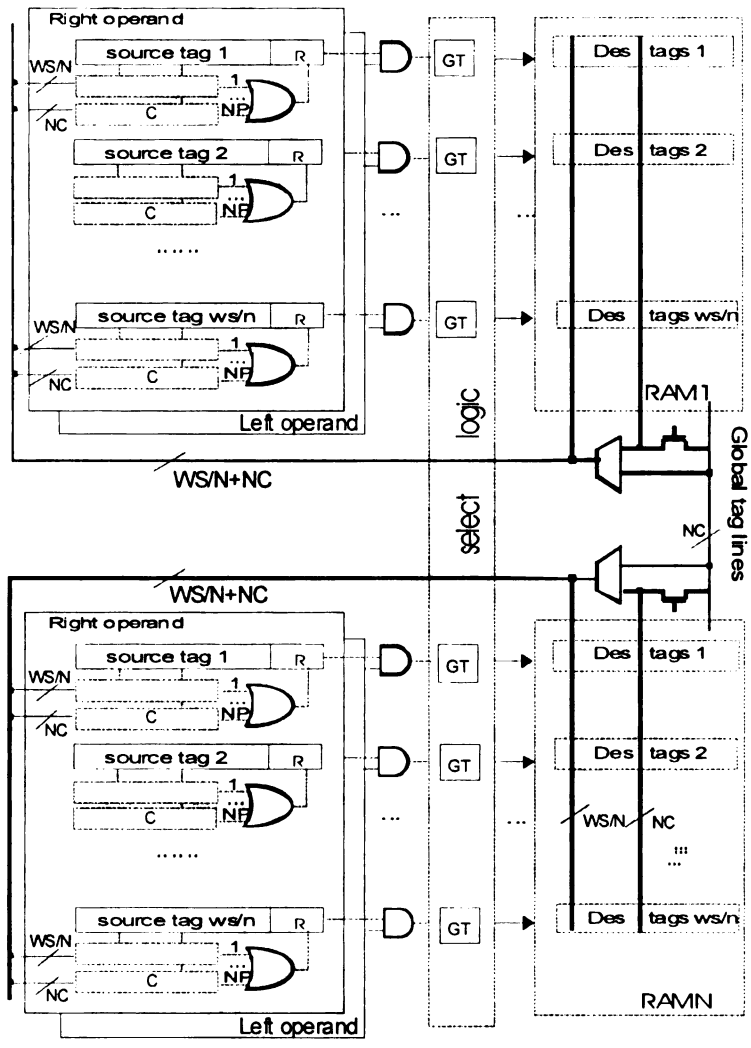


Figure 3.2. A banked instruction queue.

To improve resource utilization, the communication ports are reused for local instruction wakeup on the cycles when global tag lines are idle. Up to *M* instructions can be selected for issue in each bank. In the following section, the operand tags that have to be sent to other banks are called global tags, they otherwise called local tags. The issue requests generating global tags are called global requests and are called local requests otherwise. A *GT* bit associated with a CAM entry is added in the instruction queue to indicate the type of issue request. It is used by the select logic for issue arbitration. The

select logic chooses at most *NC* entries that have a set *GT*. If the *GT* bit of an issued instruction is set, its destination operand will be put on the global tag lines through the pass transistors in Figure 3.2.

With a fixed *M*, reducing the amount of bank communications would increase the ports available for instruction issue and wakeup. Therefore, newly decoded instructions need to be assigned to banks in a manner that minimize the amount of communication. The steering logic, which implements the steering policy, assigns a decoded instruction to a bank, and notifies the producer instruction by setting its *GT* if dependent instructions are sent to different banks

3.2.1 Instruction Steering

Steering logic is needed to assign renamed instructions to instruction queue banks based on the steering policy. There is a tradeoff between IPC performance and the hardware complexity of the steering logic. A complicated algorithm that achieves less communication may have larger delay thus hurting the clock rate.

Based on data dependencies between instructions, there are three cases when steering instructions.

1) both operands are ready: the instruction is inserted into the assigned bank without setting bit *GT*.

2) only one operand is ready: If the bank of the producer instruction is the same as that of the new instruction, then no further action is taken. Otherwise, the producer instruction having been steered to the instruction queue needs to be notified that its dependent instructions are in another bank.

3) both operands are not ready: it is similar to case 2). The *GT* bit of two entries corresponding to the two producer instructions might be set.

An approach similar to [50] is used to set up this bit (shown in Figure 3.3). Two new fields *Ei* and *Pi* are added to the rename alias table (RAT). Field *Ei* points to the instruction in the bank that will produce the corresponding register. *Pi* indicates the bank where the producer instruction is assigned. Considering an instruction newly assigned to a bank by the steering logic, if both operands are ready then the instruction is inserted into the assigned bank. Otherwise, the entry corresponding to the producer instruction is accessed through *Ei* and its *GT* bit is set.

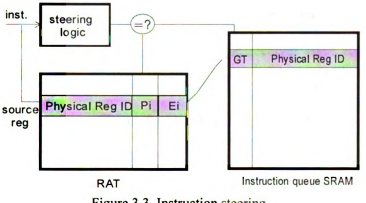


Figure 3.3. Instruction steering.

There is another implementation option to set up bit *GT*. Instead of using a pointer *Ei* to point to the producer in the instruction queue, the *Physical Reg ID* in the RAT can be sent to the instruction queue and compares with the *Physical Reg ID* field in the SRAM. The *GT* of matched entry are set. This method doesn't need the extra storage cell for *Ei* in the RAT, and therefore reduces the RAT complexity. However, extra comparators are needed for associative searching, turning the SRAM into CAM structure. This work adopted the first option.

3.2.2 *Select Logic*

In this banked instruction queue design, the select logic should meet the following requirements when selecting instructions for issue.

- Total number of selected instructions from all banks are no more than *IW*.
- Total number of selected instruction accessing to the global tag lines are no more than *NC*.
- Assuming the number of active global tag lines is N_a, each bank can issue up to M-N_a local instructions simultaneously.

The global and local issue requests are processed separately. The issue requests of ready entries are sent to either the global requests block or the local requests block shown in Figure 3.4. Each block implements a two-level select logic [95]. Take the local requests block for example, a first-level select logic receives issue requests from one CAM bank and asserts up to *M* requests with all unselected requests reset to logic zero. The total $N*M$ requests selected in the first level are then sent to the second level for final arbitration of up to *IW* instructions. The global request block operates in a similar manner except that each first level logic selects *NC* requests, and the second level also selects *NC* requests from $N*NC$ candidates.

The global requests are set to have higher priority than local requests. Each bank can issue up to *M* instructions, but the exact amount of issue slots depends on the dynamic number of global requests. Assuming there are *X* global requests, where $0 \leq X \leq NC$, then each bank can issue up to M-X local requests. Therefore, the results from the global block are used to gate the outputs of the local block.

The final grand signals from the global or local requests block are used as the SRAM's word select lines. The selected SRAM entries put the destination operands on *IW/N* local tag buses or *NC* global tag buses shown in Figure 3.2.

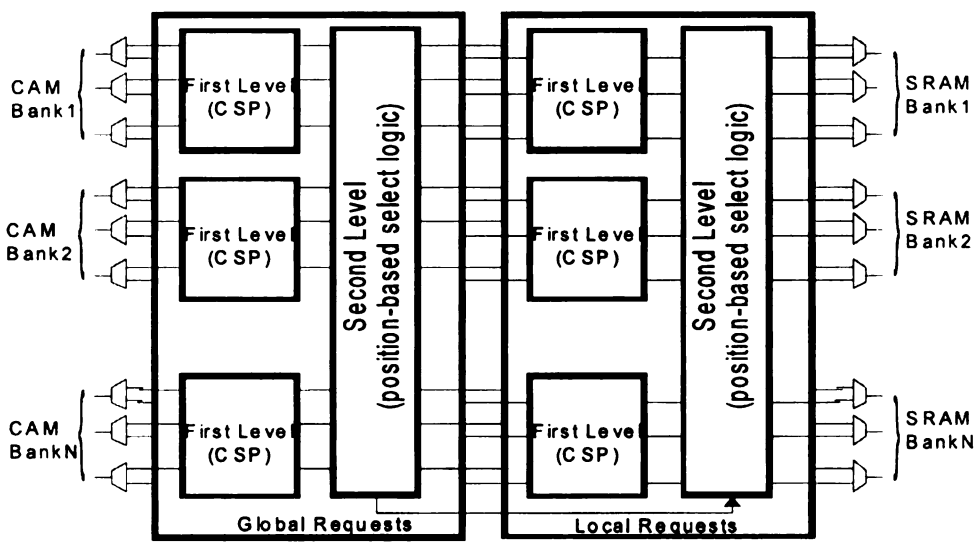


Figure 3.4. Two-level select logic.

3.2.2.1 First Level Select Logic

The Segmented prefix (CSP) circuits [94] is used in the first level that receives issue requests from one CAM bank and asserts up to *M or NC* requests with all unselected requests reset to logic low.

A modified linear CSP circuit is shown in Figure 3.5. It consists of a ring CSP circuit. Each CSP circuit has an adder and multiplexer. The ring is attached to the wrap-around CAM. The ready signal of a CAM entry is set high when the corresponding instruction is ready for issue. The head signal of each CAM entry indicates the head of the ring, which corresponding to the oldest instruction. The CSP circuits apply add operations to

successive inputs starting from the head entry, and accumulate the number of ready instructions. The sum circulation stops at the head entry by setting the head bit high.

The overflow bit (Ov) indicates if the accumulated sum is larger than the issue width. Only the ready entries without sum overflow access the SRAM to read destination operands. Once the overflow bit is set, it propagates to the rest of the ring with OR gates. The head bit chooses either to pass the OR results or set the initial overflow bit to zero. The delay of the OR gate is smaller than adders, and can bypass the slower add operations to reduce the delay. All requests after the first entry where overflow is detected are reset to low.

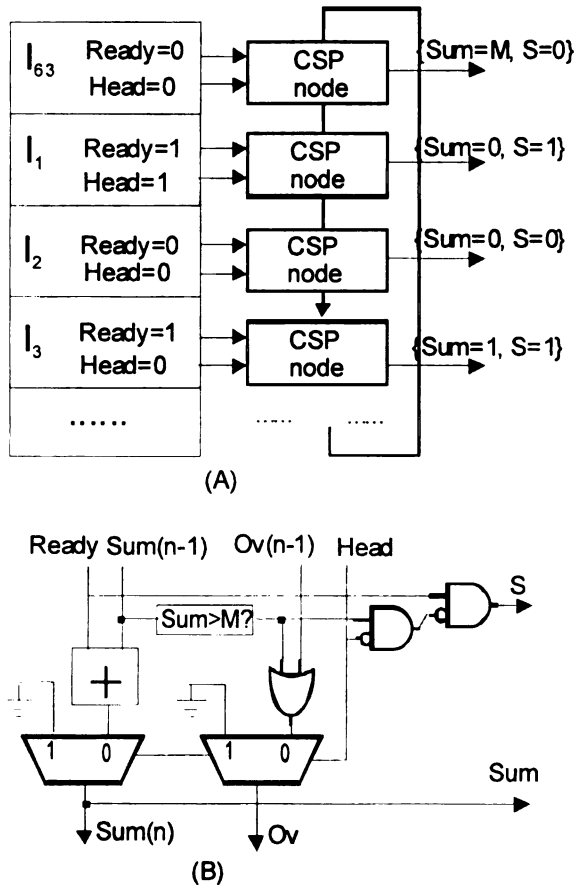


Figure 3.5. (A) Linear CSP circuits for select logic. (B) A CSP node

The CSP can be implemented in tree structures to achieve a logarithmic delay in the window size [94]. While the linear CSP circuits apply the add operator in-order to successive inputs, the logarithmic CSP circuits apply the operator in parallel to contiguous subsets of inputs. A binary tree CSP circuit has been implemented as shown in Figure 3.6. Every two contiguous inputs are grouped together in each stage and processed in parallel. The accumulated sum at the root is fed back to the previous stages to implement cyclic segmented prefix. The critical delay through the binary tree consists of 2lg*n*-1 CSP circuit nodes. One of the critical paths is shown in thick lines. The CSP node circuit shown in Figure 3.6(b) implements the accumulation function and passes the sum and overflow bit to the next node.

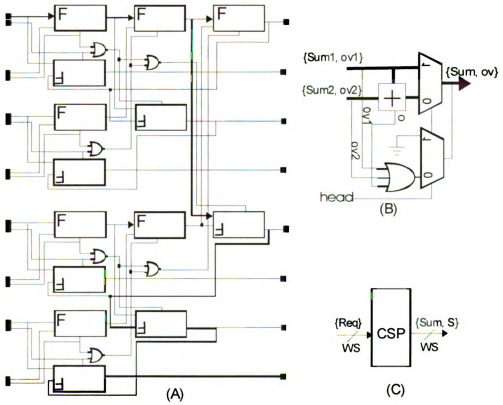


Figure 3.6. (a) Binary CSP circuits for an 8-entry CAM. (b) A CSP circuit node. (c) CSP circuits symbol.

Each CSP circuit receives up to *WS* issue requests and generates up to *WS* corresponding groups of outputs containing an accumulated *Sum* and a grant signal *S*. The grant signal indicates whether the request has been selected in the first level.

The adder width in the CSP has a logarithmic relationship with the issue width. Since each bank has a smaller issue width ($M<IW$), smaller adders can be used in the CSP circuit compared to the adder in a large instruction queue. Therefore, using small size CSP circuits can potentially lead to power and delay improvements.

3.2.2.2 *Second Level Select Logic*

The second-level select logic consists of switch networks, priority encoders and filters shown in Figure 3.7. The priority encoders select up to *IW* instructions from $N*M$ candidates. In a priority encoder, the priority is associated with the physical location of the request and cannot be changed dynamically. Because the Sum output from the first-level CSP circuits indicates the relative age of the requesting entries (relative distance from the requesting entry to the head of the queue), this information is used at the second level to implement the oldest-first select policy.

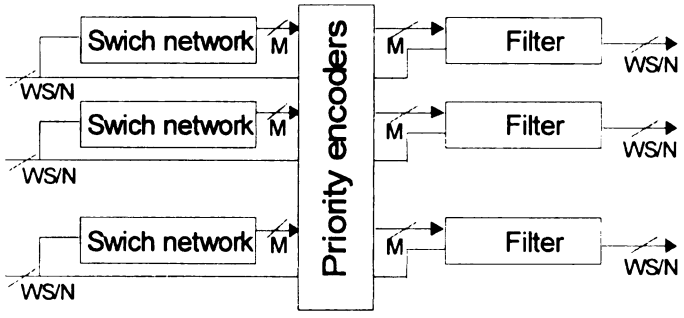


Figure 3.7. Second level select logic.

The Switch network block in Figure 3.7 passes the selected requests to *M* channels based on the sum from CSP circuits' outputs. The issue request with *sum=0* represents the

head of the waiting instructions, and are sent to the first channel. The issue request with *sum=1* goes to the second channel, and so forth. There are only *IW/N+NC* and *NC* output channels from the local and global block respectively. If there is no issue request, the corresponding channel output keeps low. These channel outputs are sent to priority encoders at the second level for final issue decision. Notice that the original CSP circuits' outputs are also sent to the second stage where they combine with the second level result to generate the final grant signals.

A simple switch network implementation is shown in Figure 3.8. Each channel consists of a dynamic OR and multiple comparators. First, the comparator in Figure 3.8(a) compares sum *A* with the channel number *C* (assuming M≤4*)*. When *A* matches *C* and the associated select signal is valid, then output CR turns high. The comparator's outputs connect to the dynamic OR gate of the channel. If one comparator detects a match, then the OR gate output turns high indicating there is an issue request from channel C. Because the channel number bits are static, the circuits in Figure 3.8(a) can be simplified. Take channel one for example, the channel number C<2>=0 C<1>=1; M6, M3, M4 and M2 can be removed because their branches are always open. In addition, M3 and M7 can be short since they are always on. The simplified circuits at channel one and two are shown in Figure 3.8(b).

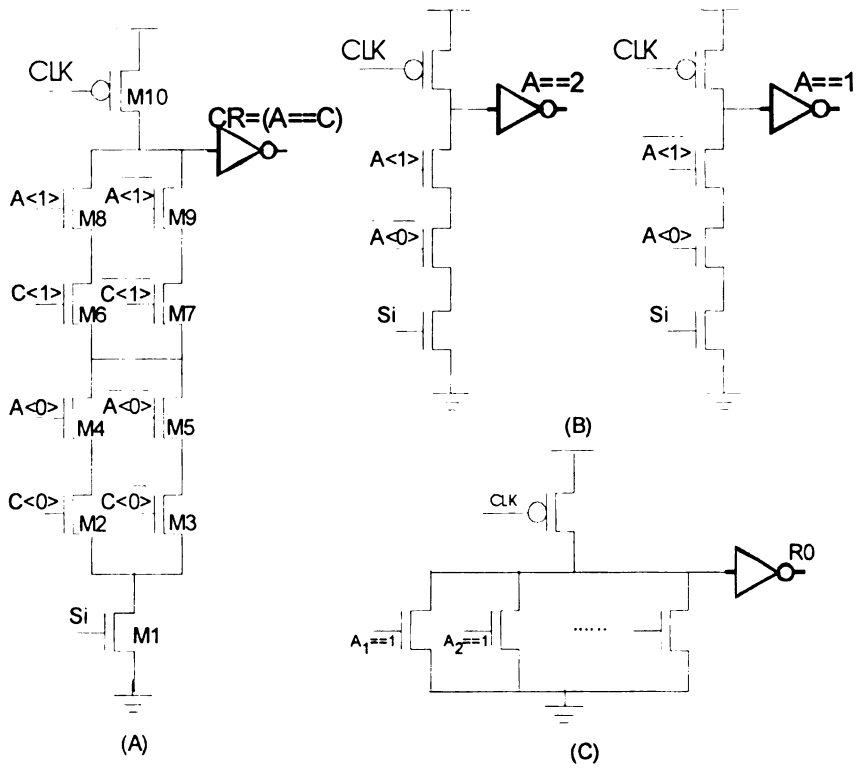


Figure 3.8. Circuits in the switch network. (a) A generic dynamic 2-bit comparator. (b) the comparators used in channel 1 and 2. (c) Dynamic OR.

The filter circuits in Figure 3.7 receive outputs from both CSP circuits and priority encoders. Only the issue requests selected by both blocks are granted for issue. Each grant signal is used as the SRAM word select signal for a corresponding SRAM bank. The circuits in Figure 3.9 can be used as the filter circuits. A grant signal from the priority encoder output is passed through a MUX to an AND gate based on the sum value of each entry. If this signal and the corresponding output of the first level are both high, then the issue request is granted.

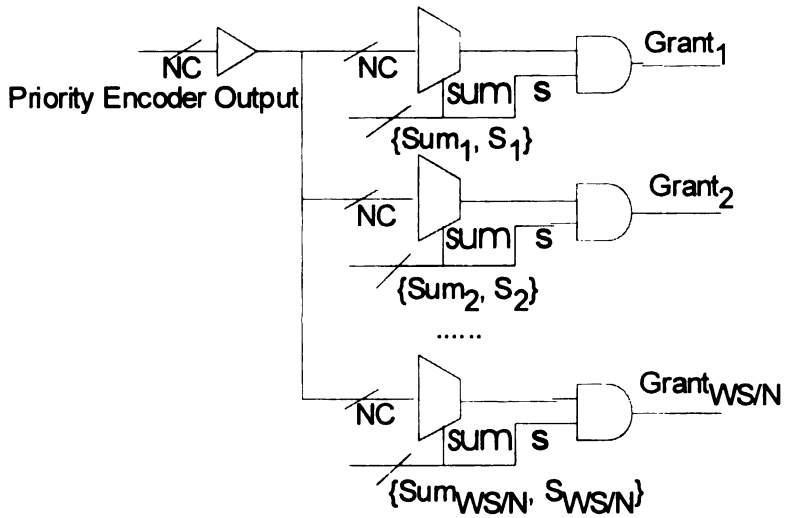


Figure 3.9. Filter circuits used in Figure 3.7.

IW priority encoders are employed to select *IW* instructions in parallel. Because there are total *N*M* requests, each priority encoder receives *N*M/IW* requests and grants one. To ensure that the oldest instruction of each bank always wins an issue slot over younger instructions, the issue requests to the priority encoder are interleaved as shown in Figure 3.10. The *N*M* requests from all first-level banks are grouped and ordered by the channel number. This is assuming that each priority encoder receives two signals, A and B, and grants one with priority A>B. The ordered *N*M* requests are connected to input A of each priority encoder in sequential order and then to the input B, so that lower channel numbers have higher priority. Figure 3.10 shows an example of the interleaved requests for *N*=4, *M*=4 and *IW*=8.

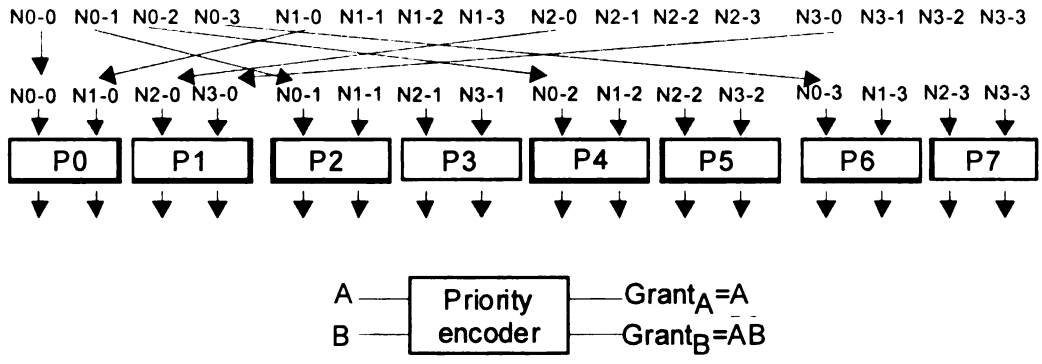


Figure 3.10. Interleaved inputs to priority encoders. *NX-Y* represents a request from channel *Y* of bank *X*. .

The global requests have higher priority than local requests and are always granted to be issued. The outcomes of the global priority encoder are used to reset the last *NC* grant signals from the local priority encoder because they share the same search port. The outputs to the priority encoder consist of *N*NC* grant signals in which up to only *NC* are set high. These set signals are used to reset the *NC* local requests in each bank shown in Figure 3.11. Finally, the priority encoder outputs are used to filter the outputs from the first level logic.

The priority encoder is used in the second level. Figure 3.12 shows a three inputs priority encoder. *S0* has the highest priority, and is always granted if it is high. *S1* will be granted only if *S1* is high and *S0* is low. *S3* has the lowest priority; if either *S0* or *S1* requests to issue, *S3* will not be granted. The global section has *NC* priority encoders operating in parallel to grant up to *NC* requests that will access to the global tag lines. Similarly, *IW/N+NC* priority encoders in the local section grant up to *IW/N+NC* requests simultaneously.

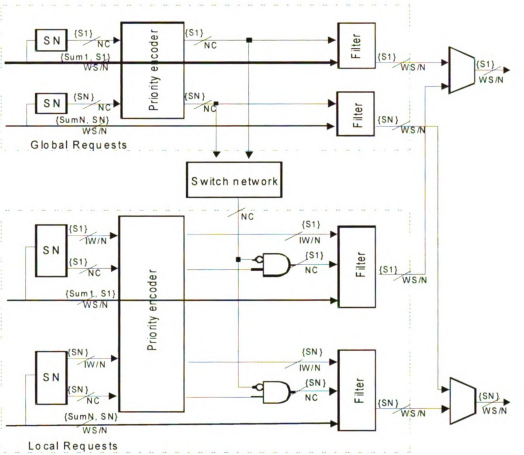


Figure 3.11. The select results of global requests resets grant signal of the local requests.

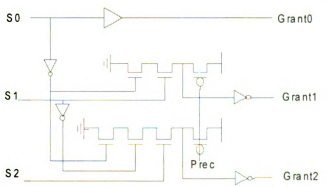


Figure 3.12. A three inputs priority encoder.

Note that this two level scheme does not implement a global oldest-first policy; a newer instruction in a lightly loaded bank could be prioritized over an older instruction in

a heavily load bank. However, simulations show that this quasi oldest-first policy has only slight impact on IPC performance.

3.2.3 *One Cycle Delay On Global Tag Lines*

Global tag lines are slow because they go through all banks and access all entries in the instruction queue. Allowing one extra cycle delay on the global tag line reduces the critical path delay and potentially improves the clock rate. However, this would increase the latency to wakeup instructions that are at different banks from the producer instruction, degrading IPC performance.

An instruction has up to two operands. There are three cases pertaining to the availability of the operands: 1) both are ready: the instruction competes for the issue slots without delay. 2) only one operand not available: there could be one cycle penalty to wakeup the instruction depending on the relative locations of the producer and the consumer. 3) both operands are not available: the location of the producer that will be executed early is not on the critical path and doesn't change the wakeup latency. As simulation will show in the result section, the extra cycle delay in the global bus has a modest effect on the performance when the amount of communication is low.

3.3 *Implementation & Results*

The instruction queue delay and IPC performance varies with the value of N and M. For example, a small M means a few ports of each bank, resulting in a small wakeup delay. However, M limits the maximum number of instructions issued from a bank in one cycle. Due to the dynamic nature of instruction wakeup, the request entries are not evenly distributed to each bank. If the issue request limit is reached in a bank, some requests

might be blocked even though there are empty issue slots in other banks. Thus, low values of M could significantly impact processor IPC performance.

The instruction queue has two operations: instruction wakeup and select. Both the delay and IPC performance on various bank configurations has been simulated. The critical path circuit was implemented on IBM 0.13um CMOS and was simulated with Cadence Spectre with a power supply of 1.5v.

SimpleScalar-3.0 [66] has been modified to model the processor with a banked instruction queue. The register update unit is decomposed into instruction queues, physical register files, and a reorder buffer. The architectural parameters are summarized in Table 3.1. Integer benchmarks having different IPC levels, branch misprediction and memory reference numbers were selected from the SPEC CPU2000 suite. The benchmark executables were precompiled Alpha EV6 binaries available with SimpleScalar. The first one billion instructions were bypassed to skip the startup code and the next 100 million instructions were simulated. The inputs were from the reference inputs for each benchmark.

TABLE 3.1. MICROARCHITECTURE CONFIGURATION.

Fetch/dec/commit rate	8
Issue rate	8
Register file size	256
Integer ALU	8
Instruction queue size	64
Branch predictor	Comb., 1k PHT, 8-bit global history, 2k 11-bit local history
LSQ size	64
D1/I1 cache	32k, 2 way, 32-byte lines, 2-cycle latency
D2 cache	2M, 4-way , 64-byte lines, 8-cycle latency
Main Memory	100 cycle

3.3.1 *Wakeup Delay*

The instruction wakeup delay consists of the delay of reading destination operand tags from the SRAM and searching for the matching tag in the CAM. For SRAM read operation, the selected cell discharges one of the differential bitlines that are both precharged to VDD. The small signal on the bitline is amplified by the sense amplifier to reduce the read delay. To provide a good margin to process variation and noise, the sense amplifier is enabled after a 10% supply voltage swing has been established on the bitline.

The wakeup delay vs. M is shown in Figure 3.13. Decreasing M reduces the length of tag lines and match lines. In addition, a small M requires small OR gates that combine the match line results. Therefore, the delay of searching for matching tags is reduced when the issue width M deceases. The delay of reading operands from SRAM is also improved because a large portion of this delay is establishing a voltage swing between differential bitlines before sense amplifiers turn on. This delay increases more than linearly with the total port number.

The overall wakeup delay of the centralized design (M=8) is 89% larger than the minimum configuration (M=2). Reducing M from eight to four results in a 32% wakeup delay reduction. Considering the effect of wire delay which grows quadratically with the wire length, higher percentage delay reduction can be expected in new processes with smaller feature sizes.

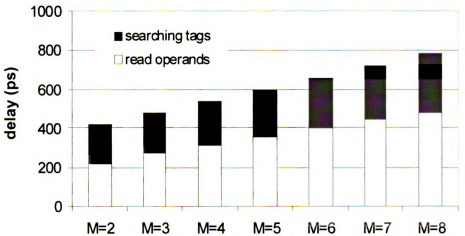


Figure 3.13. The wakeup delay vs. M value.

3.3.2 Select Logic Delay

The critical path of the select logic consists of two parallel sections, each of which consists of CSP circuits, a switch network, priority encoders and a filter. The delay of the CSP circuits dominate the total select delay due to sequential add operations. The delay depends on the adder size and has logarithmic delay in the bank size. The section processing the local issue requests has a larger issue bandwidth, requiring a larger adder and more add operations. Therefore, the delay of the local section determines the selection delay.

Assuming four banks with four issues per bank (N=M=4), there are seven total stages in the critical path of the first-level CSP circuits. The worst delay occurs when the sum starts from zero and increments to three. Delays of the two-level select logic are listed in Table 3.2 along with delays for a conventional select logic constructed with binary tree CSP circuits. The conventional centralized design has a significantly larger delay for sum accumulation because it requires 11 stages and eight sequential add operations in the worst case. The switch network, the priority encoder, and the filter circuits incur a delay

overhead for the two-level hybrid design. However, the total delay is still 36% less than a large centralized implementation.

TABLE 3.2. DELAY OF TWO SELECT LOGICS ON ONE CONFIGURATION

	CSP	Switch network	Priority encoder	Filter	Total
4-CSPs	523ps	83ps	52ps	95ps	0.75n
Baseline	1171ps	-	-	-	1.17n

Figure 3.14 shows the select delay for various bank configurations. N=1 represents the conventional centralized design. The delay increases with M due to the sequential add operations before the accumulated sum reaches M. Dividing the instruction queue into fine grained banks reduces the number of stages and the accumulation operations on the critical path, and results in a smaller total delay than a larger bank. There is a small jump in delay when increasing M from four to five, because a wider adder is needed in the first level select logic for sum accumulation.

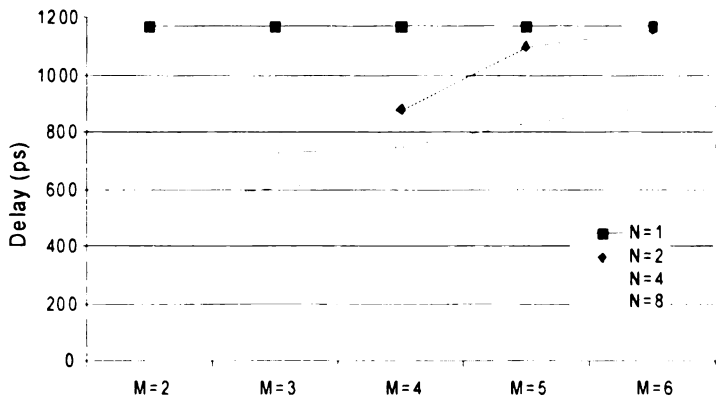


Figure 3.14. Select delay vs. N and M.

Total instruction queue delays normalized to the centralized design are shown in Figure 3.15. An 8-bank instruction queue with 2-issues each bank reduces the total delay by a maximum of 39%. A 2-bank configuration barely improves the delay due to the overheads of the second level select logic. The instruction wakeup and select can be

implemented in two pipeline stages to support a high clock rate. The critical path delay improvement would depend on the longer operation of the two stages. In this design, the select logic is slower, and the two-level select logic reduces the select delay up to 50%. Note that the wakeup delay would increase relative to the select logic as wire delay increases in new technologies.

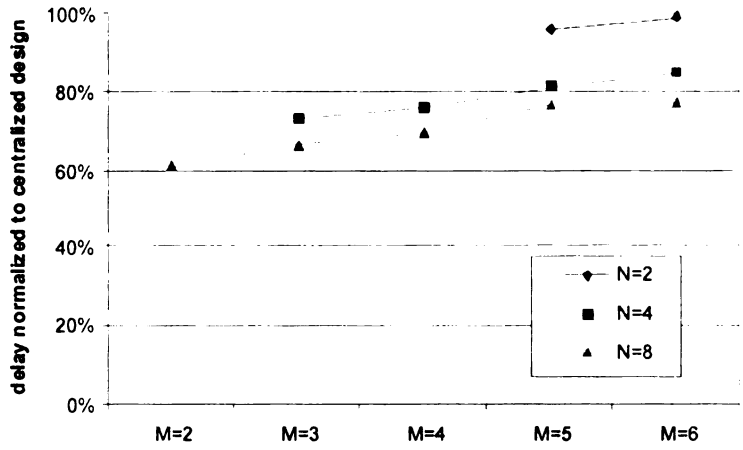


Figure 3.15. Total instruction queue delay normalized to the centralized design.

3.3.3 IPC Results

3.3.3.1 Instruction Steering Policy

Deciding the number of the global tag lines is critical to achieve an optimal tradeoff amongst delay, power, and performance. If there are not enough global tag lines, multiple cycles might be needed before an instruction wins the global resources. There is then a delay to issue the dependent instructions. However, adding global tag lines require more communication ports, essentially eliminating the benefit of banking.

The amount of communication between banks is dependent on the instruction steering policy. Several steering policies [62,63,65] of different complexity were described as follows.

- ***DEP***. New instructions are steered after the register dependence information has been updated based on instructions in the previous cycle and the preceding instructions being renamed in the same cycle. This policy can achieve the performance close to the optimal algorithm that is *NP* complete. The drawback is that it needs to analyze data dependencies between new instructions before the steering decision is made which may hurt clock cycle.

The data dependency logic at the register renaming stage can be reused in instruction steering. However it may increase the size and delay of the register renaming logic. We chose to use dedicated steering logic and add an additional pipeline stage before the instruction wakeup stage for instruction steering.

- ***CTRL***. This policy partitions programs by control flow instead of data flow. It can be done by monitoring branch instructions that are easy to identify at the instruction fetch or decoding stage. The steering logic assigns consecutive instructions to the same instruction queue segment until a branch instruction is reached, and then the instructions are assigned to the next segment. Instructions within a basic-block are likely dependent, so reasonable performance can be achieved. Since branch instructions can be identified at the instruction fetch or decoding stage, no extra pipeline stage is needed for instruction steering.

- ***MOD***. Instructions are assigned in a modulo three fashion. The first three instructions are assigned to segment 0, the next three to segment one and so forth. It has been shown that the mod3 algorithm has minimal complexity and is an efficient steering method for a quad cluster processor [63]. Similarly, the steering decision can be made in parallel with register renaming, and no additional pipeline stage is needed.

- ***RAND***. Instructions are steered to segments blindly. It has no pipeline penalty and is used as a reference to evaluate the effectiveness of other steering algorithms.

Assuming there are unlimited communication ports, Figure 3.16 shows the activity of the global tag line vs. steering algorithms for a 4-bank configuration. DEP requires less communication than other steering algorithms; in about 80% cycles, there is no communication between any banks. CTRL increases the amount of communication by almost 50%. MOD performs the worst, only slightly better than steering instructions blindly.

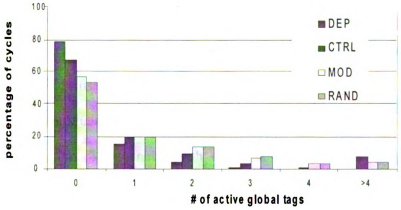


Figure 3.16. Global tag lines activity vs. steering policies (4 bank configuration).

Figure 3.16 shows that the communication ports are idle most of the time. For the *DEP* steering policy, the percentage of cycles when more than two ports are used for communication is less than 3%. For all steering policies, three communication ports can meet more than 90% requirement. However, when the ports for communication are limited, the stalled instructions due to resource confliction would compete for the communication ports again in the next cycle. Therefore, the amount of stalls would be

amplified and causes more performance degradation.

3.3.3.2 *IPC performance*

The amount of communication increases with the number of banks. It is because with more banks, the dependence graph of programs is divided into more pieces, and the communications between them increase. The IPC for an instruction queue with two communication ports are shown in Figure 3.17. With DEP, the 2-bank design reduces IPC by 3.5%. There is slight increase in the IPC penalty (4.1%) when a 4-bank configuration is used. This penalty almost doubled for an 8-bank design.

The other steering policies incur a higher IPC penalty due to the dramatic request increases in global tag line access. For the 2-bank design, MOD performs worst with about 8% IPC degradation. Assigning instructions blindly can achieve the best workload balance, offsetting its higher overhead of bank communication for a 2-bank configuration. The IPC of 4-bank and 8-bank instruction queues are more sensitive to bank communication. The penalty with CTRL and MOD grows significantly, only slightly better than RAND.

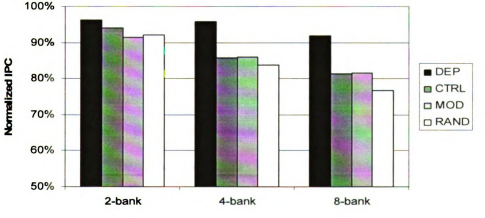


Figure 3.17. IPC performance vs. instruction queue configurations (fixed two communication ports for all configuration).

This result shows that with the same communication resources, a course-grained bank configuration required less communication and therefore has higher performance than a fine-grained design. The IPC performance is susceptible to steering policy and bank configurations when the communication resource is relatively small. DEP requires the least amount of communication and has the lowest IPC penalty for all configurations.

We are more interested in the performance variation with total hardware resources that are directly related to the value of *M*. Figure 3.18(a) shows the IPC variation with *M* on the DEP steering policy. The explored *M* is limited to six to get meaningful hardware resource reduction compared to the non-banked design. For an 8-bank instruction queue, there is a 16% IPC penalty with only one communication port (*NC*=1, *M*=2). The penalty is reduces to only 5.6% when three communication ports are employed (*M*=4). Further increasing the ports, however, doesn't help IPC noticeably. Adding two more ports (*M*=6) only reduces the IPC penalty by 1%. The 4-bank configuration has the highest IPC performance when *M*>3.

The rational behind this is that the 4-bank configuration has more resources for communication compared to a 2-bank design, and therefore has less stall due to global resource confliction. In addition, when the global tag busses are idle, they can be used for local instruction wakeup, so the individual bank has the same peak bandwidth. While an 8-bank CAM provides more resource for communication, Figure 3.16 shows that it is unlikely that more than three global busses are active in the same cycle for communication. On the other hand, more banks compromise the oldest-first select policy. The critical old instructions in a crowd bank might be bypassed by younger instructions in other banks, causing IPC degradation.

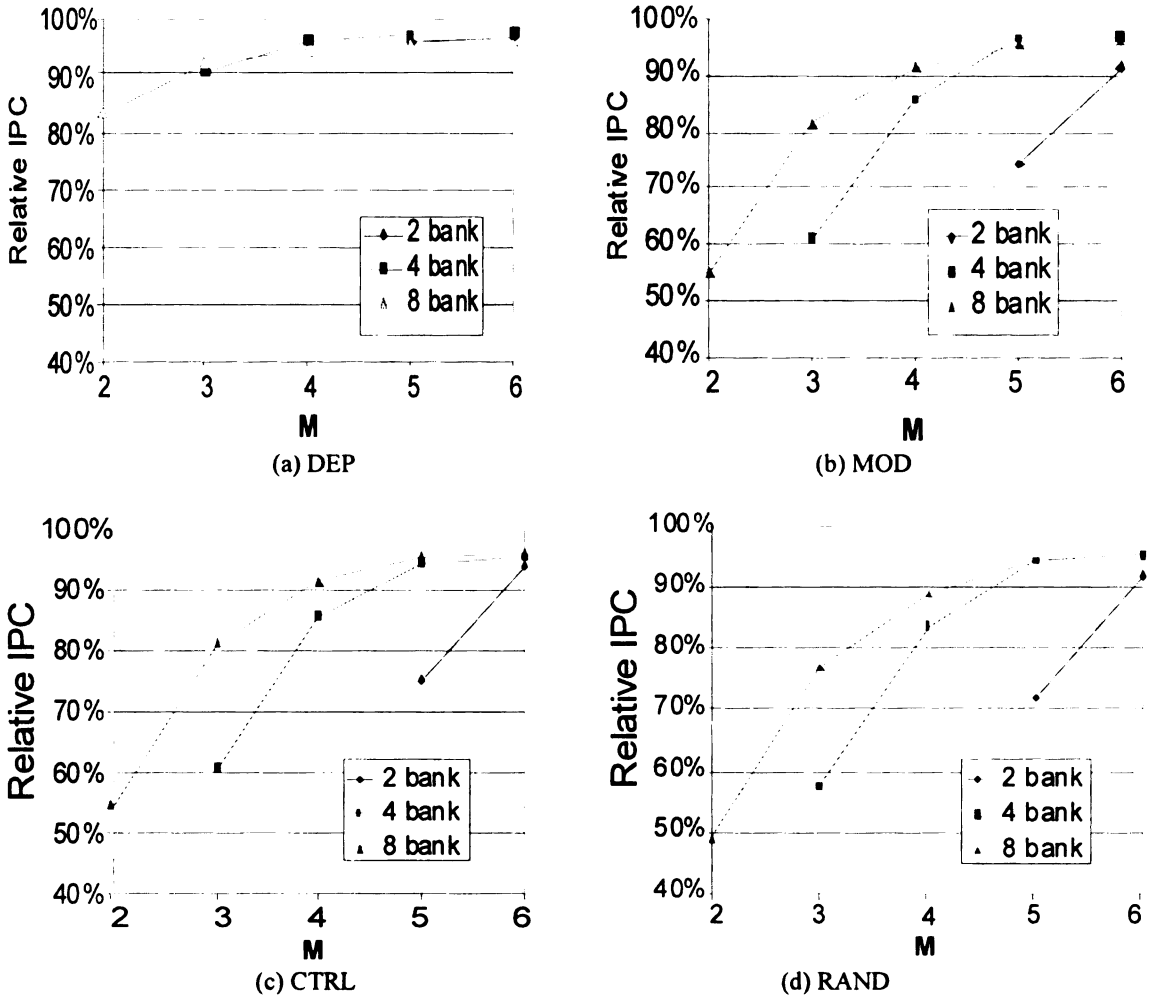


Figure 3.18. IPC vs. port number for various stering algoirthm.

Figure 3.18(b) shows the normalized IPC variation for the MOD steering policy. Because MOD incurs more communication between banks, the IPC performance benefits significantly from a large *M* (more communication ports). Different from DEP, the 4-bank configuration didn't outperforms an 8-bank until $M>4$. This is because the IPC with MOD is more sensitive to communication resources, and an 8-bank CAM has more ports available for communication with the same *M*. Therefore, the 8-bank configuration is preferred using the MOD steering policy if hardware resources are limited ($M<5$). Compared to DEP, one more port is needed when the IPC starts to grow slowly. Similar

to DEP, a 2-bank configuration has the lowest IPC which is mainly limited by its few communication ports under the same hardware budget.

The IPC with the other two steering policies perform similar to the MOD. When there are no more than five ports, the 8-bank configuration has the highest IPC. Figure 3.19 summarizes the relative IPC of all configurations. Because CTRL and RAND have similar or worse performance than MOD, they are not included in the comparison. When there are very limited port resources (M<4), DEP produces higher IPC than other steering methods and the 8-bank configuration has the highest IPC performance with an 8.1% IPC penalty. When the port number is increased to four, a 4-bank instruction queue reduces the IPC penalty to 4.1%, and is slightly better than the 8-bank design. With a 5-port hardware budget or higher, all configurations achieve close IPC with variations within 2% except for the 2-bank. In this case, MOD is preferred over DEP because of the simple hardware to implement the MOD steering policy.

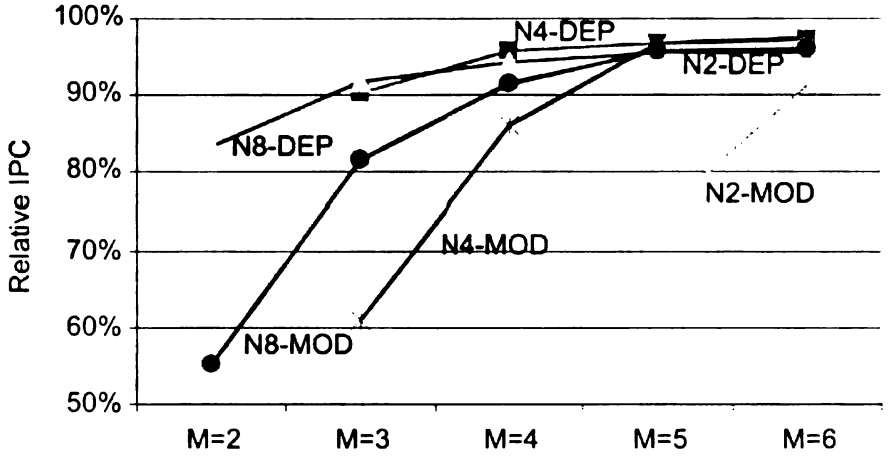
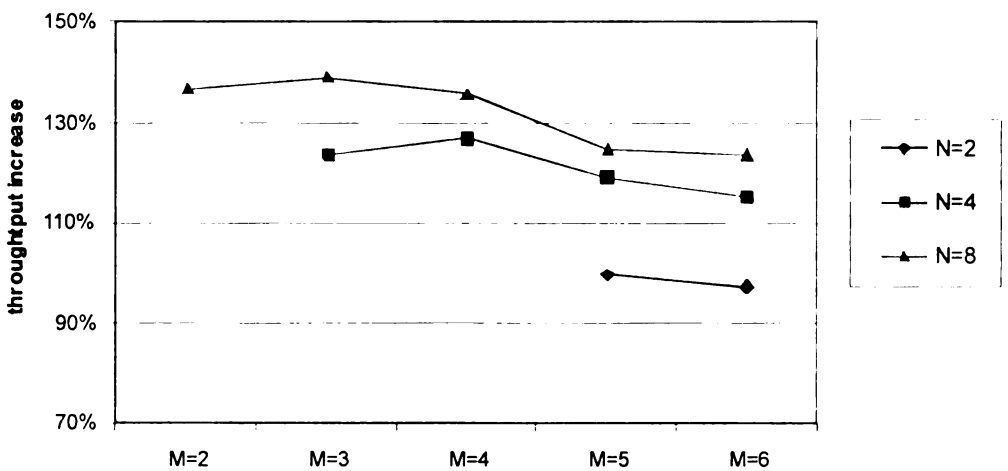


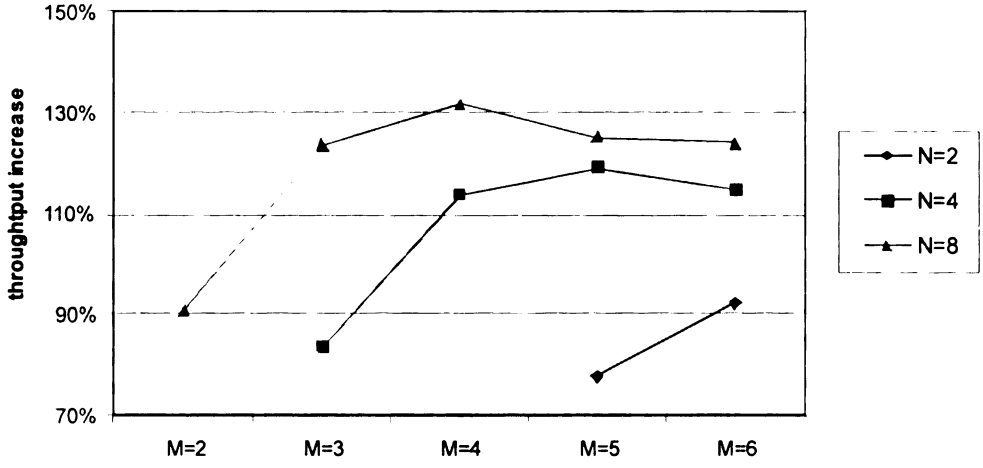
Figure 3.19. IPC performance vs. bank configurations on single program.

3.3.3.3 *Instruction Throughput*

The overall processor performance is measured by the instruction throughput (*clock rate* 譏*PC)*. Assuming the instruction queue delay determines the clock rate, the processor performance vs. bank configuration is summarized in Figure 3.20 based on previous delay and IPC results. With the DEP steering policy, the 8-banked instruction queue has the highest performance and can achieve a 37% higher performance than the centralized design. Note that each bank only needs to support an issue width of three to achieve the maximum performance. A larger *M* degrades the performance because the increase in delay is more significant than the IPC improvement. With the MOD steering policy, 8-bank configurations still outperform the 4-bank designs under the same hardware resources. However, each bank should have an issue width of four to obtain the maximum performance.



(a)



(b)

Figure 3.20. Instructions throughput vs. bank configuration on (a) DEP; (b) MOD steering policy.

3.3.3.4 *One-clock delay in global tag line*

An extra clock delay in the global tag lines allows faster clock rates to issue local instructions at the cost of possibly increasing the latency of dependent instruction executions. Based on the results of the last section, the 2-bank configuration has poorer performance than 4-bank or 8-bank configurations, and therefore is not included in this discussion. CTRL and RAND have similar or worse performance than MOD and are also removed from the design space. Furthermore, only M=3, 4 and 5 are analyzed because they have reasonable IPC and hardware resource tradeoffs.

Figure 3.21 shows that a 4-bank instruction queue with the DEP steering policy has an average of 15% instructions accessing the global tag lines. MOD increases this to more than twice the amount of dependent instructions that are not assigned into the same bank. This number increases modestly for 8-bank configuration.

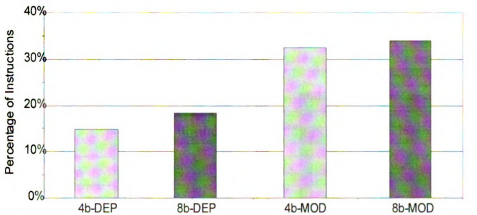
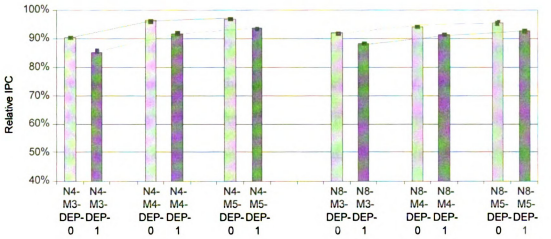


Figure 3.21. Percentage of instrucions accessing to the global tag lines. (4b-dep represents 4-bank configuration with *dep* steering policy)

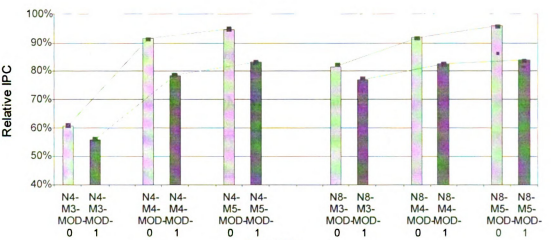
In spite of a significant amount of inter-bank communication, only those instructions whose last unready operand is located at another bank would cause a performance penalty. Figure 3.22 shows the impact of the extra clock communication delay. An instruction queue configuration is denoted by *N-M-SP-d*, where *N* is the bank number; *M* is the total port number; *SP* is the steering policy, and *d* denotes the communication delay. The performance impact with the DEP steering policy is much smaller than that with MOD. Across all configurations, one clock communication delay causes a 2-5% IPC penalty with the DEP steering policy. The 4-bank configuration has slightly higher performance than the 8-bank, which requires a higher amount of communication.

For the MOD steering policy, increasing the port number doesn't noticeably improve the IPC. For example, the IPC gap of an 8-bank configuration is increased from 9% to 13% when *M* increases from four (*N8-M4-MOD-1*) to five (*N8-M5-MOD-1*). It is because the amount of communication is so high that the performance degradation caused by the communication delay is much larger than that caused by port conflictions. Therefore, minimizing the amount of communication is critical when multiple cycle

communication delays are allowed.



(a) DEP



(b) MOD

Figure 3.22. IPC impact of one-clock communication delay.

3.3.3.5 IPC Performance on Multi-program Workloads

The banked instruction queue on a single thread has more than 4% IPC penalty that cannot be improved by employing more communication ports. This is because ready instructions are not distributed evenly among banks. Some banks are idle and others are overloaded on some cycles. The hardware utilization can be improved with multi-

threaded or multi-program workloads. The increased parallelisms reduces the amount of idle cycles at a bank due to lack of ready instructions. Figure 23a shows the IPC with the DEP steering policy. For $M>3$, the IPC degration decreases with more threads. For example, the configuration $(N,M)=(4,4)$ has 4.2%, 3.6%, 1.7% IPC degradation when there are one, two, and four running threads, respectively. $(N,M)=(4,5)$ has even less than 0.1% IPC degradtion on multithreades workloads. While an 8-bank configuraton has a lower IPC than a 4-bank configuration on single thread, it outperforms a 4-bank with the same M on multithread workloads. Fine-grained designs have more communication ports with the same hardware resource, workloads imbalance is not a performance limitation anymore on multithreading. Notice that there should be at most two commucation ports (M=3 for N=8, and M=4 for N=4), otherwise the IPC penalty is even larger on multithreaded workloads than on a single thread.

Figrue 23b shows the IPC on the MOD steering policy. As the performance is mostly limited by the communication resources, it is more sensitive to M than on DEP. Again, the 8-bank configuration has higher performance than the 4-bank configuration. However, the MOD still incurs a 7% IPC degration on four threads when $M\leq 4$. Therefore, a fined-grained bank configuration with the DEP steering policy is prefered on multithreaded workloads.

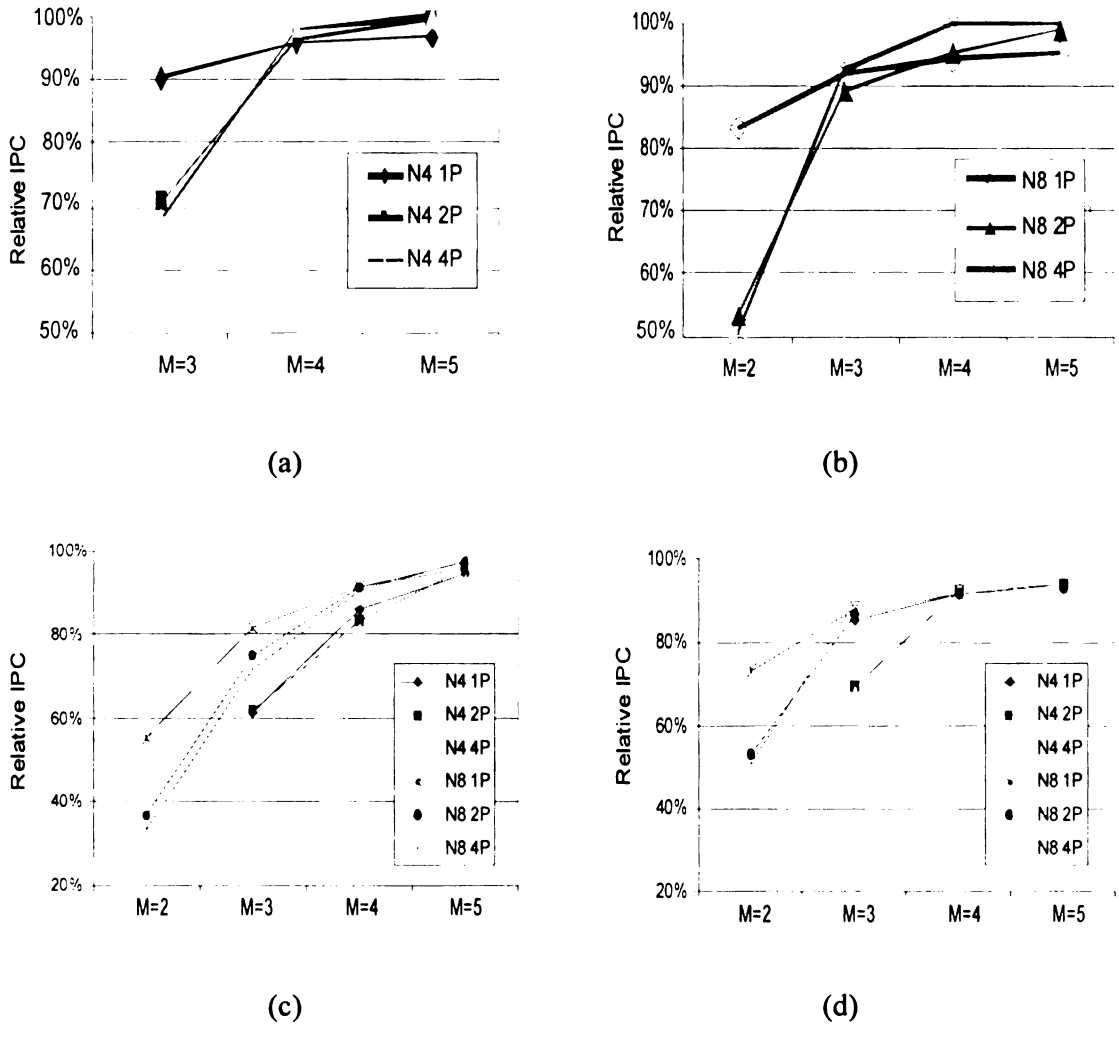


Figure 3.23. Relative IPC on multithreading (a) DEP on 8-bank configuration (b) DEP on 4-bank configuration (c) MOD (d) DEP with 1-clock communication delay.

Figure 23c shows the relative IPC when there is a one clock communication penalty. Based on the above results, only the DEP steering policy is considered. The IPC degration is significant when there is only one port for commuication (*(N,M)=(8,2), or (N,M)=(4,3)*). On the other hand, increasing the communication ports to more than two has only a slight performance improvement. There is a 6.9% IPC degradtion for *(N,M) = (8,4)* on four thread workloads compared to less than 0.1% when there is no

communication delay. Additional port resource reduces the degradations to 5.4%. This result shows that the performance is limited by the communication delay. Only two or three are needed to achieve the best potential perfomance.

It is interesting to note that the communication penalty has a small affect on the performance of processes which have a tight hardware budget. With M=3, the 8-bank configuration increases the IPC degration from 7.3% to 10.7% on four threads with one clock communication penalty. A smaller M contributes to a faster clock rate and may result in higher overall performance.

3.4 *Summary on Banked Instrution Queue Design*

Our design reduces the physical size of the instruction queue by reducing its memory ports. A centralized structure is divided into N parallel banks. The hardware size of each entry is determined by the issue width M of each bank instead of the overall issue width IW. The select logic still selects up to IW ready instructions from all banks to match the other parts of the pipeline. Compared to the IW issued centralized processor, the physical size of the instruction queue is reduced by M / IW. In addition, our design reuses the communication resources when they are idle, achieving a high hardware utilization.

Our design supports easy tradeoffs between hardware complexity, the IPC, and the clock rate. The design space of our banked instruction queue is determined by three factors: the number of banks N, the issue width of each bank M, and the instruction steering policy. The optimal bank configuration depends on the specific constrains of a processor. The physical size of the whole instruction queue is determined by M. When the hardware budget is limited, M is small. With as little as 25% (M=2) hardware resources of a centralized 8-issue processor, the 8-bank configuration obtains 83% IPC

performance of the centralized design. With 50% hardware resources (M=4), the IPC performance can be improved to 95.8%.

High clock rates can be achieved with a fine-grained bank configuration (large *N*) and a small issue width of each bank, potentially resulting in high overall performance (instruction throughput). For example, an 8-bank instruction queue with M=2 can achieve 36% higher performance than the centralized design, assuming that the processor clock rate can be increased at the same rate as the delay improvement of the instruction queue. The IPC performance is not sensitive to the bank granularity and the steering policy when the hardware budget is relatively large. With about 63% (M=5) hardware, the IPC performance varies by 2%. Therefore, simple steering logic can be used to minimize its affect on the pipeline.

3.5 *Another Instruction Queue Optimization Technique*

Optimization of individual stages allows to ulilize the unique characteristics of the target stage and come up an optimal design to meet particular design constrains. We have designed another independent optimization technque [97] described in this section.

In a conventional design, the destination register of each instruction is mapped in the renaming stage to the physical register identifiers. The identifiers are used to wakeup waiting instructions as well as to index the physical register at later cycles. Using the register identifier in the wakeup/select process unnecessarily limits the design space of the instruction queue. In fact, the tags at the instruction queue are only used to keep track of instruction dependencies. They can be separated from the register identifiers without any effect on the pipeline. The instruction queue chooses a set of code as tags to update

the instruction status. The hardware of the instruction queue can be optimized based on the code characteristics to achieve lower delay and/or power consumption. Tag separation greatly increases the design space of the instruction queue. In this section, the potential of such decoupling techniques is demonstrated with two different tag coding methods that were designed to reduce the wakeup delay by targeting different delay components in the wakeup process.

The first method targets the tag match delay component in the wakeup delay. A linear code is used as the tag that is composed of the physical register identifier and redundancy bits. The physical register identifiers are sent from the instruction queue to the pipeline register for register access, and the redundant bits are used in the wakeup process to reduce the tag match delay. The second method uses the one-hot code to remove the OR delay components in the critical path. This scheme allows the use of the grant signals from the select logic as tags broadcast to all entries in the instruction queue, therefore removing the destination tag read as well as the tag OR operations.

Notice that the non-compacting instruction queue is assumed in this section. The delay is limited only on the instruction wakeup delay. Instruction select is in another pipeline and therefore its delay is not in the wakeup critical path.

3.5.1 Reducing Tag Match Delay

As discussed in Chapter 2, tag match delay takes up more than 40% of the total wakeup delay. At the precharge stage, the matchline in Figure 2.9 is precharged to VDD through a pMOS transistor. At the evaluation stage, if there are any mismatch bits between the tag and the data in memory cells, the match line is discharged through the

nMOS transistors of the mismatching cells. The worst case delay occurs when only one bit is mismatched, turning on only one discharge nMOS to pull down the match line.

To reduce the tag match delay, one option is to increase the size of the pull-down transistor. However, this would increase the tag line load, which would result in higher tag drive delay and power consumption. In addition, all of the pull-down transistors on the match lines would have to be increased. This would increase the parasitic capacitance of the match line, and also would require more discharge time thus offsetting the speed benefit of large pull down transistors.

Figure 3.24 shows the tag drive and tag match delay vs. the pull down transistor size. Using four times minimum size transistors reduces the match line delay by 60% percent at the cost of a 10% increase in tag drive delay compared to using minimum size transistors.

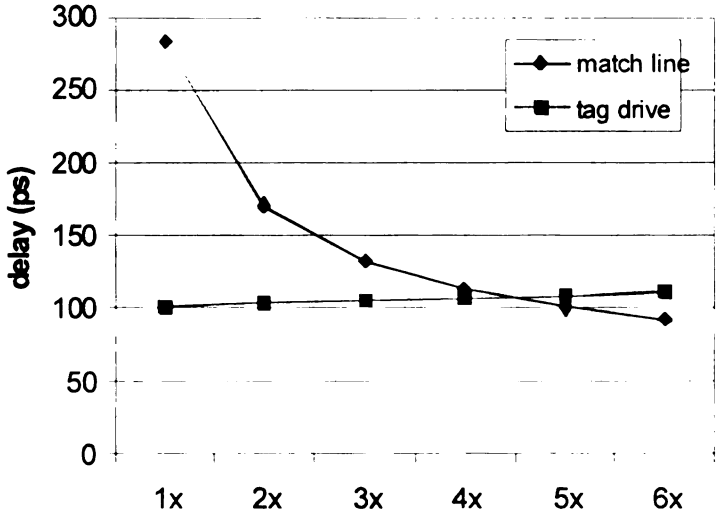


Figure 3.24. CAM delay vs. transistor size.

The tag match delay also depends on the number of pull-down nMOS that turn on simultaneously. This number is equal to the Hamming distance between tags. In the

conventional design, the tags are binary codes referring to physical registers, and the minimum Hamming distance (MHD) between any two tags is one. However, if the tags are encoded so that the minimum Hamming distance is greater than one, tag match delay can be reduced without increasing transistor size.

3.5.1.1 *Tag Encoding*

Implementing a separated set of tags in the instruction queue requires additional hardware to store the tags and link the tags of the issued instructions to the physical register identifiers that are used for register access. Binary linear codes [95] are suitable for this application. A linear code C is referred to as a $[n, k]$ code, where n is the length and k is the dimension of the code. A linear code can be considered as having two parts. The first part of a codeword has k bits representing the information content. In this case, it is the physical register identifier. The second part has n-k bits. These are redundant bits that are used to increase the code Hamming distance. Therefore, there remains a direct mapping from the tags in the instruction queue to the physical register identifier. Only n-k memory cells are needed to store the redundant bits.

A. Minimum Hamming Distance = 2

The minimum Hamming distance of a linear code is determined by the minimum weight of C. The weight of a binary codeword $w(c)$ is the number of one bit in the codeword c. The minimum weight of the code C is defined as the minimum nonzero weight among all nonzero codewords. For linear codes, the minimum weight equals the minimum distance shown as follows:

Let *d(a,b)* represent the Hamming distance between codewords *a* and *b*. Then *d(a,b)=d(a-c,b-c)* for all codewords *c*. In particular, *d(a,b)=d(a-b, b-b) = d(a-b, 0)=w(a-b)*.

To find a code *C* with a minimum Hamming distance *d*, it is only necessary to ensure that each codeword has at least *d* bits with logic value one. Notice that each codeword of the check parity code has an even number of 1's, so any nonzero codeword has at least two 1's. Therefore the minimum Hamming distance of the check parity code is two. For an *n*-bit codeword, only one redundant bit is required, and the other n-1 bits represent the physical register address.

B. Minimum Hamming Distance > 2

A code with a large minimum Hamming distance allows more pull down transistors to turn on simultaneously, reducing the average tag match delay. A greedy algorithm can be used to find sets of codewords with a Hamming distance of *d*. This is done by starting with codeword *0* then counting upward, adding a codeword that has a Hamming distance of at least *d* to the previous codeword. This algorithm will produce a linear set of codes with minimum Hamming distance *d*.

Figure 3.25 shows that the number of redundant bits required grows as the minimum Hamming distance increases. More redundant bits than information bits are needed when the minimum Hamming distance is larger than five. This increases the size of the tag and the number of memory cells to store the tag. More importantly, longer tags incur a larger diffusion capacitance on the match line, offsetting the benefit of increasing discharge paths. Based on these considerations, only codes with a minimum Hamming distance less than four have been considered.

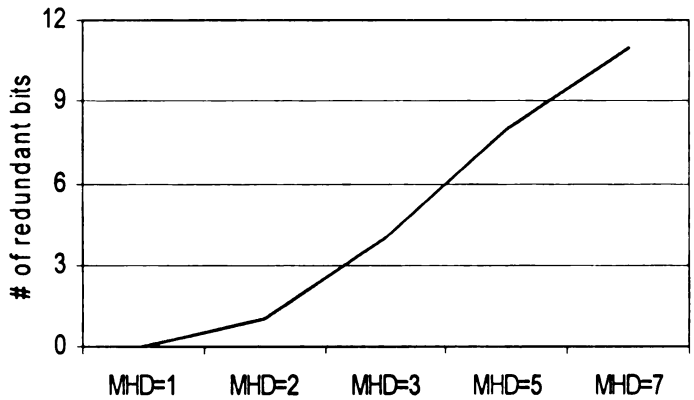


Figure 3.25. The number of redundant bits versus the minimum hamming distance (8 information bits).

3.5.1.2 Implementation

Operand tags are obtained at the instruction decoding stage when the names of logical registers are translated into physical register identifiers. It is assumed that a CAM structure mapping table is used to keep track of the mapping from logical registers to physical registers [45]. The size of the mapping table is equal to the size of the physical register file with each entry representing a physical register. The entry contains the name of a logical register and a valid bit. Since a logical register can be mapped into multiple physical registers, the valid bit is used to indicate the latest mapping. In the renaming process, the source operands of new instructions are searched associatively in the mapping table. The latest entry into which the logical register has been mapped is activated. The word lines corresponding to the entry are used to access a ROM and retrieve the physical register identifier.

In a traditional design, each ROM entry stores the binary code of a physical register identifier. Thus the length of an entry is logarithmically proportional to the register file

size. In this design, each entry stores the codeword of a linear code. The same number of cells store the register identifier, and extra cells store the redundant information. Assume, for example, that the check parity code is used for tag coding, then only one extra cell is needed in the ROM representing the parity of the physical register identifier. No extra delay is introduced because the redundant bits are stored in ROM and accessed in parallel with the information bits.

When a ready instruction is selected for issue, its destination tag consisting of the physical register identifier and the redundant bits are broadcast to the instruction queue. Only the register identifier part of the tag is sent to the pipeline register used to access the register file. Therefore, the added redundant bits in the operand tag have no effect on any other hardware units.

3.5.1.3 Results

The number of information bits in the tag directly depends on the size of the register file. While the number of redundant bits is always one for the check parity code, it varies for a larger minimum Hamming distance depending on the specific code implemented. Coding efficiency is defined as the ratio of the number of information bits over the length of the codeword. Table 3.3 lists the coding efficiency and number of redundant bits for MHD=3 verses the number of information bits. When the register file contains from 64 to 1K entries, the number of redundant bits remain at four. Therefore, the coding efficiency increases as the size of the register file increases over this range, resulting in better delay improvement.

TABLE 3.3. CODEWORD SIZE AND CODING EFFICIENCY VERSUS REGISTER FILE SIZE FOR MINIMUM HAMMING DISTANCE OF THREE.

Size of register file	No. of information bits	Codeword size	No. of redundant bits	Coding Efficiency
32	5	8	3	0.625
64	6	10	4	0.6
128	7	11	4	0.64
512	8	12	4	0.67
1k	9	13	4	0.69

Figure 3.26 shows the tagline delay versus different coding schemes. Tags with MHD=1 represent the conventional binary code. Tags with MHD=2 represents the check parity code. For a minimum transistor size, the check parity code reduces the matchline delay by 42% over the traditional design. Using tags with MHD=3 achieves only an additional 8% reduction because the number of cells storing the redundancy bits grows from one to four as the tag distance is increased from two to three.

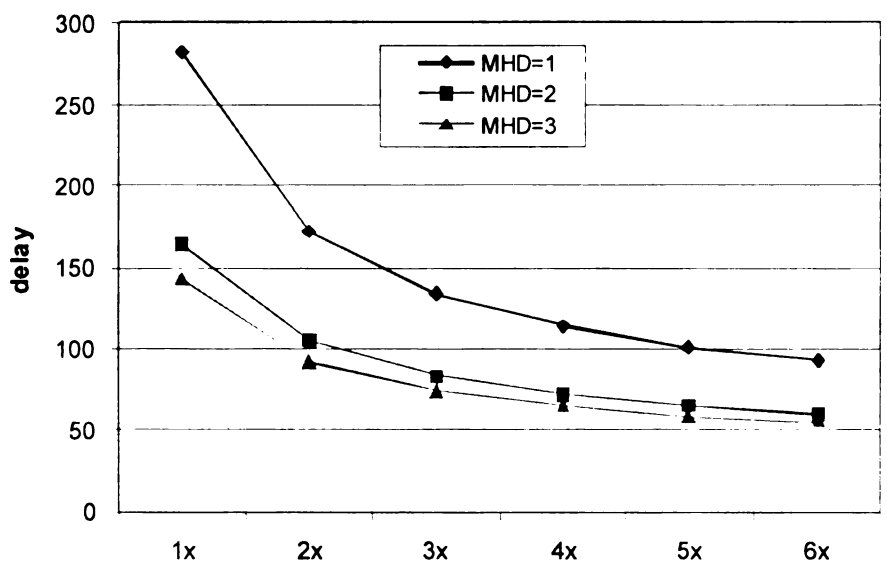


Figure 3.26. CAM matchline delay vs.XOR transistor size in the CAM cell.

3.5.2 *Removing Tag-OR Delay*

Each source operand in an instruction queue entry corresponds to *IW* match lines that are OR'ed to set the ready bit of the operand. Let the destination tags of issued instruction

be $c^1, c^2, \cdots, c^{IW}$, and the source tag of entry k in the instruction queue be c_k. In the traditional design, c_k is encoded in the binary code. Let the compare operation of tags a and b be $a \oplus b = \sum_m a_i \oplus b_i$. Then the function to set the ready bit is $r_k = \overline{c^1 \oplus c_k} + \overline{c^2 \oplus c_k} + \cdots + \overline{c^{IW} \oplus c_k}$. The ready bit is high if any of $c^1, c^2, \cdots, c^{IW}$ is c_k.

The OR delay accounts for about one third of the wakeup delay. This delay can be removed by encoding tags with one-hot code. With the dot operation of tags a and b is defined as $a \cdot b = \sum_m a_i \cdot b_i$, then

$$c_i \cdot c_j = 1, i = j,$$
$$c_i \cdot c_j = 0, i \neq j$$

Consider a function: $r_k' = c^1 \cdot c_k + c^2 \cdot c_k + \cdots + c^{IW} \cdot c_k = (c^1 + c^2 + \cdots + c^{IW}) \cdot c_k$.

If any of $c^1, c^2, \cdots, c^{IW}$ is c_k, then $r_k' = c_k \cdot c_k = 1$, otherwise $r_k' = 0$. Therefore, this function can be used in the instruction queue for associative search operations, and $c^1 + c^2 + \cdots + c^{IW}$ is the broadcast signal to update the instruction queue. The 1' bit in the one-hot code is set to be the grant line of the corresponding entry. Then all of the grant lines from the select logic realize $c^1 + c^2 + \cdots + c^{IW}$; therefore, no OR gates are needed to implement the function r_k'.

In addition, the combination of the grant lines is equivalent to the destination tag read from the RAM in the conventional design. Thus, one-hot encoding removes the RAM

read operation, further reducing the wakeup delay. The wakeup logic that implements one-hot encoding is similar to the dependence matrices described in [49].

3.5.2.1 Implementation

The register rename logic keeps track of the mapping from logic registers to both physical register identifiers and instruction queue identifiers. An entry of the register map table is shown in Figure 3.27. It contains the physical register identifier (R_tag) and the instruction queue identifier (I_tag) that is mapped to the same logical register. The logical register identifier is used to index the map table. An instruction reads the table to obtain the R_tag and I_tag for each architecture source register. The tags of the allocated physical register and the instruction queue are also written to the map table. The same rename logic used in the traditional design can be applied to create, in parallel, the R_tag and the I_tag. I_tags are used for instruction wakeup, and R_tags of issued instructions are sent to the pipeline register to access the register file.

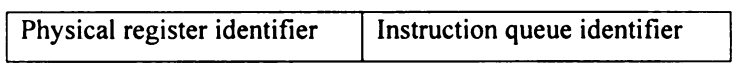


Figure 3.27. An entry in the register map table.

The instruction queue implementing one-hot encoding, shown in Figure 3.28, is similar to the conventional instruction queue in Figure 3.1. XOR operations in the conventional CAM are replaced with AND operations. Each source operand of an entry has only one match line, which is precharged to VDD. When there is a match, one of the pull down transistors turns on, discharging the match line and generating the ready signal. The grant signals from the select logic are used as the tag in one-hot encoding and are

broadcast to all entries in the instruction queue. R_tags of issued instructions are sent to the pipeline register.

One important feature of this design is that it does not have the dissipation-on-mismatch issue. In the conventional design, the mismatch bit turns on a corresponding pull down transistor to discharge the match line. Because most of the entries in the instruction queue are not dependent on the issued instructions, a significant amount of power is wasted by charging and discharging match lines in every cycle. Intensive research has been done to reduce this type of power consumption [58,59]. However, the binary coding used in the conventional design requires bitwise XOR operations that limit power and delay tradeoff. With one-hot encoding, the XOR operation is replaced by a faster, lower power AND operation, and the pull down transistor is turned on only when there is a match. Therefore, this scheme removes the power consumption associated with discharge-on-mismatch that dominates the instruction queue power consumption [57].

However, one-hot encoding requires more memory cells to store operand codes. The length of the instruction queue identifier is the size of the instruction queue. Thus, the number of memory cells in the wakeup logic increases quadratically with the queue size. Moreover, the load of the match line grows linearly with the queue size. For large instruction queues, the increased delay at the match line would quickly offset the benefit of removing the OR delay and destination operand read delay.

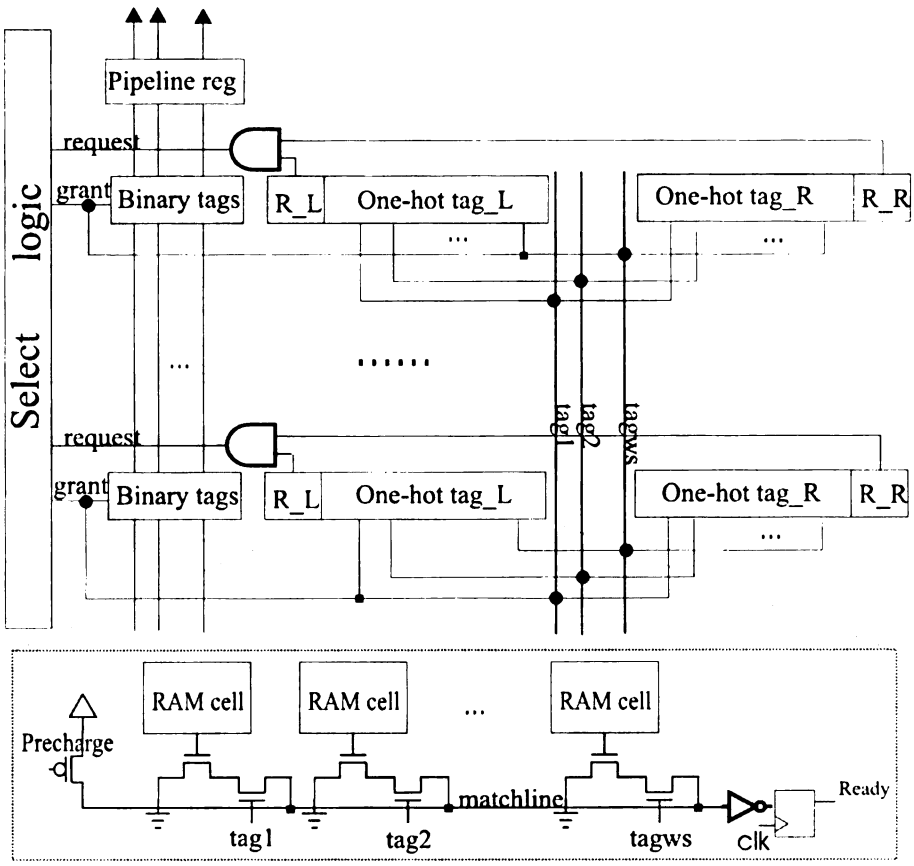


Figure 3.28. Instruction queue implementing one-hot encoding.

3.5.2.2 Results

The wakeup delay is shown in Figure 3.29. For a 16-entry instruction queue, the one-hot encoding scheme improves the delay by 49% due to reduction of the OR delay and operand read delays. The improvement drops to 25% for a 32-entry instruction queue because the loads of the tag match line is doubled, causing much larger tag match delay than in the traditional design. When the size of the instruction queue increases to 64, the dramatic increase in tag match delay exceeds the sum of the OR gates and RAM access delay.

Results show that the one-hot encoding scheme has delay advantage for small instruction queues. Larger instruction queues can be divided into multiple partitions that are executed in a pipeline or in parallel. However, because the size of the instruction queue only affects the tag drive and the SRAM access delay, the wakeup delay in the traditional design reduces slowly as the size of the instruction queue is decreased. Some research work assumes a small and fast instruction queue. The one-hot encoding scheme would provide significant delay improvement in such designs.

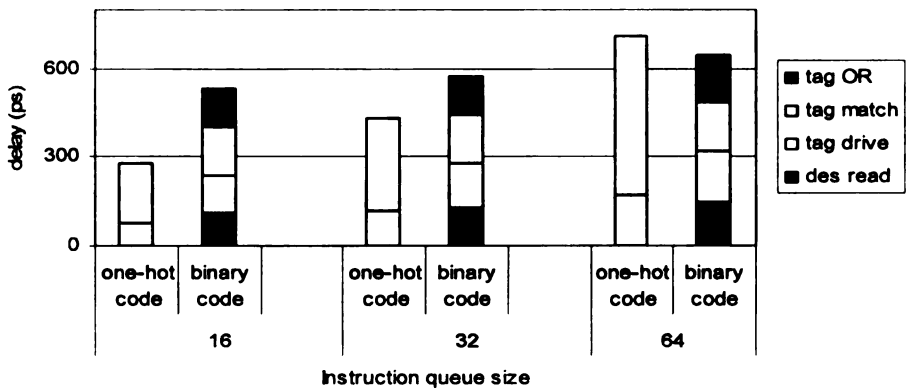


Figure 3.29. Wakeup delay using one-hot coding verses instruction queue size on IBM 0.18μm technology.

3.6 Summary

Our first optimization approach utilizes data dependencies between instructions to reduce hardware resource requirements of the instruction queue. The second approach applies coding techniques to reduce the instruction wakeup delay. Two optimization approaches can be used together to achieve higher performance. For example, the linear code can be used in the banked instruction queue to further reduce the matchline delay components.

Many approaches have been proposed to improve the instruction design. Folegnani and Gonzalez [57] adjusted the effective size of the instruction queue to increase the energy efficiency. Brekelbaum et al. [60] split the instruction window into a large, slow window and a small but fast window. The large window is to extract the far-flung ILP while the small window is issue the critical instructions. Michaud [52] employed a large prescheduler buffer and a small issue buffer to search/issue instructions. New instruction are sent and reordered at a prescheduler buffer, and then dispatched to the issue buffer in the data-flow order.

These approaches reduce the memory size (number of entries) of the instruction queue and could be used together with our banked design that reduces the memory ports. The combination of multiple optimization approaches would result in significant improvement over the conventional design. Therefore, there is large room to improve the critical stage to increase the processor performance. Many hardware of the conventional processor are over-designed, and significant amounts of hardware can be reduced with only a modest impact on the processor performance.

4 AN ADAPTIVE CLUSTERED MULTITHREADED MICROARCHITECTURE

The banked instruction queue in Chapter 3 describes an optimization of a critical pipeline stage of a wide-issue processor (SMT). The overall processor instruction throughput have been improved by reducing the delay of the critical stage while maintaining most of other pipeline stages unchanged. This strategy (Optimization of the current critical pipeline stage) can be applied to more stages to keep driving the performance up. However, a large amount of design effort is needed to optimize each of many critical stages. It is increasingly difficult to meet design constrains when the overall issue width increases.

Multi-core (CMP) design provides a scalable solution to increase the issue width of a processor. An *N*-core processor can be viewed as distributing the hardware of each pipeline stage of a SMT into *N* segments, and each CMP core has a segment of the distributed hardware from each stage. This strategy is scalable because the hardware of additional pipeline can be easily added as a module to the current design to provide more processing resources. However, distribution of some pipeline stages has very limited benefit to the hardware complexity but causes significant performance degradation.

Therefore, the SMT and the CMP processor represent two sides of the design spectrum. SMT has the highest level of hardware utilization. However, the layout area and delay of a SMT processor is large. CMP has the highest layout efficiency and supports clock rate scaling, but its hardware utilization is low when some cores are idle.

This chapter links the two opposing designs in order to explore the optimal hardware distribution scheme between SMT and CMP. We start with a SMT processor and distribute the hardware of each pipeline stages, eventually resulting in a CMP processor.

The performances of SMT and CMP on applications with different parallelisms are discussed. Based on these results, we propose an adaptive clustered multithreaded microarchitecture that achieves higher performance than both SMT and CMP.

4.1 Hardware distribution on A SMT processor

The banked instruction queue in Chapter 3 presented a distributed design that traded IPC performance for delay in a SMT processor. Similarly, other centralized hardware in a SMT processor can be distributed to support high clock rate and overall high instruction throughput. Figure 4.1a represents a conventional *IW*-issue multi-threaded SMT processor in which each centralized unit processes *IW* instructions simultaneously. Figure 4.1b represents a SMT with distributed instruction queue. The monolithic structure is divided into N_{IQ} units, each of which has small issue bandwidth with simple hardware.

Continuing distribution on the register file is shown in Figure 4.1c. N_{RF} small register files in parallel serves up to *IW* instructions. The hardware in the critical path is reduced, and therefore, its access delay is improved. Repeating this to the rest of the pipeline, including the data cache, functional units and instruction cache is shown in Figure 4.1d-f.

Hardware distribution stage by stage allows leveraging the characteristics of each pipeline stage to achieve optimal performance. Therefore the number of hardware segments of each pipeline stages could be different. In this study, however, we divide each stage with the same granularity that enables to derive CMP from SMT.

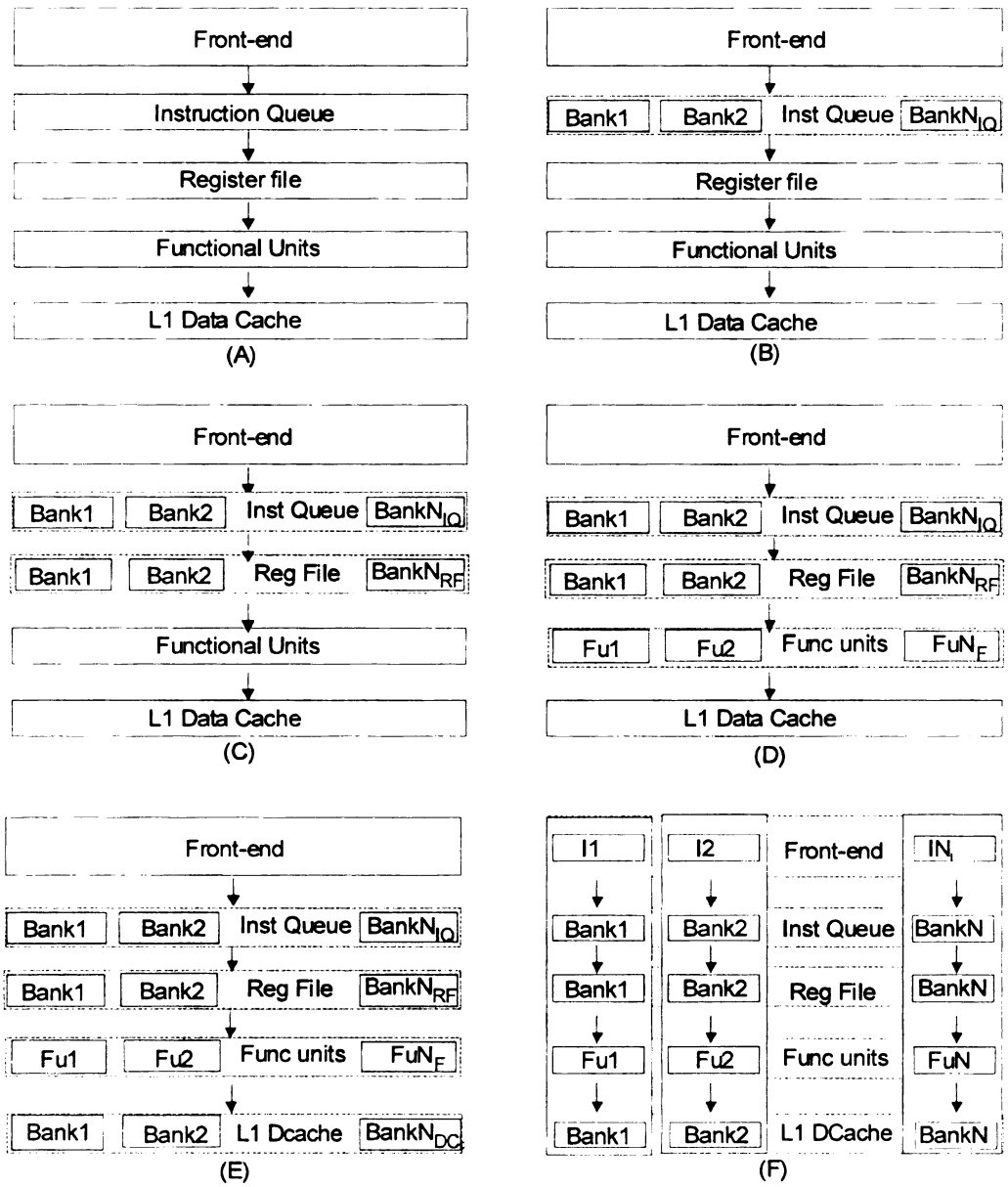


Figure 4.1. Hardware decentralization of a SMT processor.

4.1.1 Instruction queue

The distributed instruction queue uses the banked design discussed in Chapter 3. To achieve reasonable IPC performance, we choose the four-bank and eight-bank configurations; both having the same hardware resources with an issue width of four

(M=4) in each bank. Both DEP and MOD steering policies are among the design options. Based on the results in Chapter 3, the banked instruction queue reduces the hardware resources by half with an as little as 4% IPC penalty.

4.1.2 Register File

The Register file is a port-heavy memory unit in a superscalar processor and is a perfect candidate for distribution. In fact, Alpha 21264 uses two register file copies to reduce the number of read ports [32].

A new distributed register file that easily interfaces to the above banked instruction queue has been design. Note that some distributed register files [36,39] can be used here. Exploration of the optimal register file design is beyond the scope of this thesis.

Similar to the instruction queue, the centralized register file is divided into N banks, each of which is associated with an instruction queue bank. One bank has $2*M$ read ports and M write ports, so that up to M instructions issued from an instruction queue can fetch operand data and write results from/to the associated register file. The combination of a bank of the instruction queue and the register file is called a cluster. Because the physical size of each entry of the register file scales with M instead of the processor issue width IW, the total size of the register file is reduced to M/IW. For simplicity, the rest of this section uses M to represent both the read port number and the write port number instead of $2*M$ for read and M for write.

Similar to the instruction queue design, IW/N ports of each register file bank can be accessed only by the associated instruction queue bank, and $NC=M-IW/N$ ports are shared by all register files. The NC shared search ports in the banked instruction queue are used to wakeup instructions that require data from other banks, and the NC shared

read ports in the banked register file are used to read these operand value from the destination register file.

4.1.2.1 Performance Degradation

Banked register files incur additional stalls and IPC penalty when the total number of issued instructions that read data from other bank simultaneously is larger than *NC*. The amount of access to the shared ports in the register file is determined by the steering policy described in Chapter 3.

Statistically, MOD generates the same amount of access to the shared port in both the instruction queue and the register file. However, more attention needs to be paid to the DEP steering policy.

There are two types of data dependence. 1) the operand data has been available in the register file; 2) the operand data is yet to be generated by the parent instructions. The DEP policy in Chapter 3 steers instructions based on the second type. The location of the ready operand in the register file does not matter because of the centralized register file. However, DEP may cause a large amount of port conflicts in the register file. The steering logic assigns instructions based on the location of the parent instruction whose results have not been generated yet. This may cause access to the shared ports of the register file if the other ready operand is already stored at a different bank.

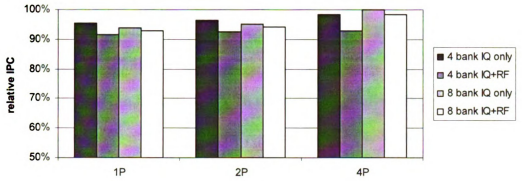
If minimization of shared port access of the register file is favored, on the other hand, access to the global tag bus of the instruction queue would increase. Even worse, this scheme causes severe workload imbalance. Once an operand value is available in a bank, it will pull all its dependent instruction to its bank, consequently drawing most instructions to one cluster.

Simulations show that assigning a dependent instruction only by an un-issued parent instruction has a much better workload balance. On average, one-third new instructions have their operands ready before entering the instruction queue. These instructions can be assigned to the bank with the least loads to distribute instructions. Therefore, the same DEP policy as in the Chapter 3 is used to assign new instructions to clusters.

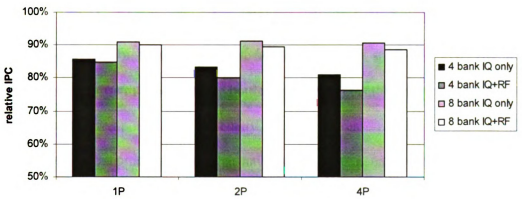
4.1.2.2 Results

A ready instruction will not be selected for issue when it requires an operand from other register files but fails to win a shared port. Figure 4.2a shows the relative IPC of 4-bank and 8-bank configurations with the DEP steering policy. On single thread, the 4-bank register file introduces additional an 4% IPC penalty to the processor with only a 4-bank instruction queue. The total IPC degradation increases to 8.5% compared to the centralized design due to hardware distribution overheads.

The 8-banked configuration has only a 1% additional IPC penalty. This is because the 8-banked register file has more shared ports on the same hardware budget, and therefore has fewer resources conflicts. A processor with an 8-bank instruction queue and register file (8-cluster) is 7.2% slower than the baseline for single program. The IPC penalty decreases on workloads with more threads. Two threads have a 6% IPC penalty and this is further reduced to 1.7% on a four thread workload. In comparison, the 4-bank configuration has a much higher IPC penalty of 7%. Therefore, the 8-bank configuration is preferred for both single and multiple thread workloads.



(a) On DEP steering policy



(b) On MOD steering policy

Figure 4.2. Relative IPC with/without register file distribution on multiple programs(P).

MOD incurs a larger number of resource conflicts on the register file and therefore has lower performance than with the DEP steering policy. Similarly, the 8-banked configuration outperforms a 4-bank because of relatively fewer shared ports in the 4-bank configuration. Because the performance bottleneck on MOD is the high amount of access conflicts to the shared ports, the IPC penalty does not decrease by increasing the thread-level parallelism. Compared to DEP that have a 1.7% IPC penalty reduced from 7.2%,

MOD has an 11.3% IPC penalty increased from 10% when the number of thread increases from one to four.

4.1.3 Functional Units

Function units are in the critical loop in the processor pipeline. The execution results of function units are written back to the register file to be accessed by the dependent instructions in next cycles. In addition, the functional units outputs are also sent to their inputs, forming a close loop, so that the outputs are available for the dependent instruction immediately. The delay of this execution loop is critical for performance. Because the majority of instructions have integer operations, modern high performance processors aggressively keep this loop delay to one clock cycle [85,86]. The Intel Pentium 4, for example, pipelines the ALU to maintain a one cycle loop delay at the cost of higher latency.

In a 32 or 64 bit processor, the physical width of the ALU is much larger than its height. The ALUs in the datapath are typically stacked together and the bypass wires go through the stack [40]. Assuming the number of ALUs is proportional to the issue width, the bypass delay increaes more than linearly with the issue width. Because the wire delay gets larger as technology scales, the bypass delay has become one of the major limitations on the clock rate.

To reduce the execution loop delay, the ALU stack is divided into segments, each of which is physically associated with a register file bank. In other words, the cluster in section 4.2 extends to the functional units. The result of an ALU is available to the ALUs at the same cluster with no latency. There are one or multiple clock latency to bypass the results to the ALUs at other clusters.

4.1.3.1 *Performance Degradation*

Segmentation of the functional units causes IPC degradation due to two factors. Firstly, the issue width of each instruction queue bank is limited by the number of the function units (*IW/N*) in the cluster. The *NC* communication ports in the instruction queue cannot be reused to wakeup local instructions when the global tag busses are idle. This can be viewed as distributing execution bandwidth into clusters without data forwarding delay between clusters.

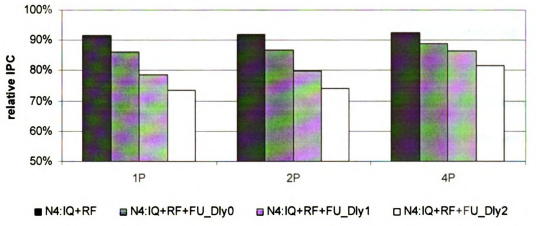
The second factor is the delay in forwarding the instruction's result to its dependent instructions that are in different clusters. This is similar to the communication delay in the instruction queue, resulting in an additional IPC penalty on top of the first factor.

4.1.3.2 *Results*

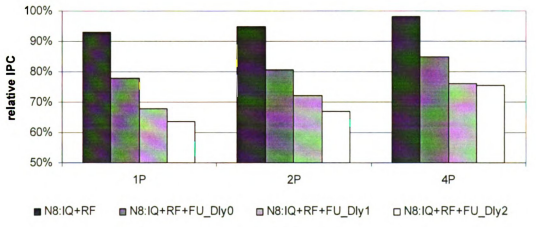
Figure 4.3a shows the effect of the above two factors on performance with the DEP steering policy. *N4:IQ+RF+FU_Dly**X*** denotes four clusters with ***X*** clock data forwarding delays between clusters. When X=0, it denotes distribution of the execution bandwidth and the new instruction results are available to all clusters with no latency.

For the four-cluster configuration, distribution of the execution bandwidth alone causes a 5.6% IPC degradation. On a single program, one clock inter-cluster forwarding delay increases the IPC degradation to 13%; while a two clock forwarding delay increases it to 18% compared to the design that has a clustered instruction queue and register file but shared execution units. The clustered design has an overall IPC degradation of 26% which is reduced to only 18% with multi-program workloads if two clock inter-cluster forwarding latencies are needed. The results show that the execution

bandwidth distribution and data forwarding cause significant performance degradation on both single and multithreaded workloads.



(a) four-cluster configuration.



(b) eight-cluster configuration.

Figure 4.3. IPC performance with distributed functional units on the DEP steering policy. (N4:IQ+RF denotes four clusters without distribution on the functional unit. N4:IQ+RF+FU_DlyX denotes four clusters with distributed functional unit and X clock data forwarding delays between clusters.)

The eight-cluster configuration, which has higher performance than four clusters before the functional units are distributed, has more IPC degradation with distributed functional units. The reason is that the eight-cluster processor has a lower functional unit utilization due to workload imbalance. In addition, the amount of data forwarding on eight clusters is also higher than on four clusters.

The performance of a four-cluster configuration on the MOD steering policy is shown in Figure 4.4. The increased amount of data forwarding on the MOD steering policy greatly limits the throughput, and therefore incurs a larger IPC degradation that increase even more on multithreaded workloads. The overall IPC degradation (35%) is much higher than that on the DEP steering policy (22%) assuming one-clock forwarding latencies. The eight-cluster configuration requires more inter-cluster data forwarding, and has even lower performance than the four-cluster design. Therefore, MOD steering policy can only be used on clusters without distribution on the functional units to obtain reasonable IPC performance.

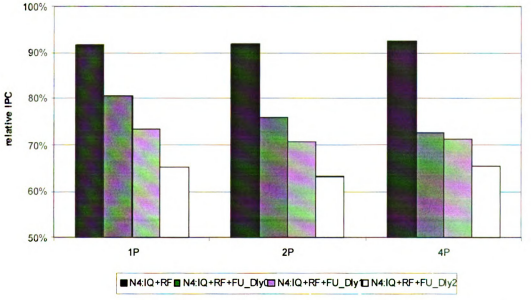


Figure 4.4. IPC performance of four-cluster configuration on MOD steering policy.

4.1.4 *L1 Data Cache*

Because on average about 1/3 of program instructions are load/store instructions, a wide-issue processor should allow for multiple concurrent memory accesses to achieve higher performance. A multi-ported level-1 cache can accommodate several accesses in a single cycle, but it has large latency and power consumption.

The cluster in the last section is extended to the level-1 cache. Each cluster has an independently-addressed cache bank. Load and store instructions are dispatched to the cluster based on the prediction on their target cache bank. Bank predictions are made at the instruction steering stage with a bank predictor [75,76]. Bank check is made concurrently with memory address computations. The instruction queue is notified during the cycle following a bank check. A correct bank prediction does not need further actions, but a misprediction comes along with the correct bank number. A mispredicted memory instruction will be able to be routed again to the correct bank.

Stores are not issued until both the data and address registers become available. Memory dependence speculation is used to expose the parallelism that is hindered by ambiguous memory dependences [73]. Load instructions are executed before the addresses of older store instructions are computed through memory dependence prediction. The issue of a load is delayed until an older store has been issued if a dependency between them has been predicted. Mispredictions are not discovered until stores execute, many cycles after loads and their dependent instruction have left the instruction queue. When this happens, the affected loads and dependent instructions are reissued from the instruction queue.

4.1.4.1 *Performance Degradation*

Bank misprediction requires rerouting the mispredicted memory instruction to the correct bank, causing a delay in its execution. In addition, its dependent instructions need to be reissued, which wastes the processor issue resources. Furthermore, banked cache puts more pressure on the instruction queue size because the issue loads and dependent instructions won't be released until the memory disambiguation is solved.

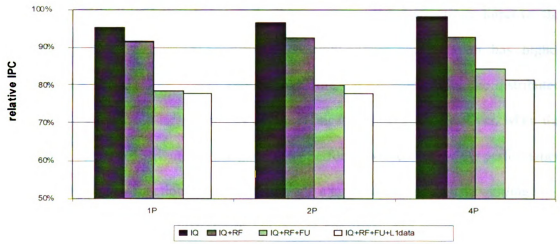
Lastly, banked cache affects other pipeline stages. It adds constraints on the instruction steering logic that not only needs to consider the data dependences between instructions, but also the bank predictor results for memory access instructions. The two constraints may generate conflicting results. The penalty of violating data dependence, which is due to increased communication between clusters, is typically smaller than routing and reissuing the mispredicted and dependent instructions. Therefore, the memory predictor results should have a higher priority over data dependency analysis results in an instruction steering decision, but larger amounts of inter-cluster communications would occurs.

4.1.4.2 *Results*

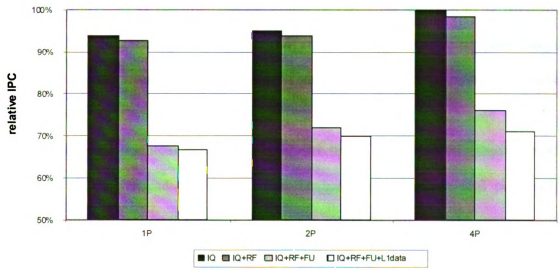
The bank prediction accuracy varies with the number of bank. The results in [74] were used to model the bank prediction accuracy: 85% for four banks and 84% for eight banks. These numbers should be regarded only as a sample of the potential of the banked cached since the specific machine configurations and designs are crucial to the choice and accuracy of the bank predictor.

The processor performance with distributed L1 data cache is shown in Figure 4.5. On a single program, the banked L1 data cache causes less then 2% additional IPC

degradation with the four-cluster and eight-cluster configurations. The degradation is increased to about 5% with four-program workloads. The increased amount of communications introduced by cache bank prediction reduces instruction throughput more significantly on multi programs than on single programs.



(a) four-cluster configuration



(b) eight-cluster configuration

Figure 4.5. IPC performance degradation due to hardware distribution from the instruction queue to the L1 data cache.

4.1.5 *Summary and Discussion*

The SMT on single program workloads has respectively 4.2%, 8.5%, 22% and 23% IPC degradation when the instruction queue, the register file, the functional units and the data cache are distributed in sequence as described above. Distribution of the functional units contributes to the most IPC degradation. Four-program workloads have a reduced total 18% IPC degradation because the increased parallel instructions improve the utilization of the distributed hardware. The eight-cluster configuration has higher performance than the four cluster alternative before the functional units are distributed owning to its relatively large amounts of communication resources. However, when the functional units are distributed, the eight-cluster performs worse because the performance is limited by its low functional unit utilization and a large amount of data forwarding

SMT and CMP are at two end of the design spectrum. The hardware distributions described in this section shows a link between the two designs. An N-core CMP processor can be viewed as extending the distribution from the back-end to the front-end of an SMT as shown in Figure 1f. A CMP core contains a segment of each pipeline stage, and is assigned to a single thread. Individual core exploits the ILP and multi-core exploit the TLP in applications.

The hardware resources in SMT processors are shared by all threads. Exposure of all hardware to each thread would maximize the hardware utilization, and potentially generate high instruction throughput. In addition, single threads can take control of all hardware resources, and the maximum IPC can be obtained when there is only one active thread. On the other hand, CMP has low hardware utilization on a single thread. As the number of threads increases, the total hardware utilization increases proportionally. Only

when there are enough threads in the pipeline, all CMP hardware resources are active, and higher performance can be obtained.

Figure 4.6 shows the performance of a SMT and a CMP processor. SMT-ideal denotes the SMT that issues dependent instruction back-to-back without increased latency as technology scales. SMT-D1 has one-clock latency to issue/execute dependent instructions and SMT-D2 has two-clock latency. The CMP processor has four cores, based on the optimal bank configuration from previous results. One core fetches and executes instructions from a single thread, and the processor support four threads. More threads will cause memory thrashing [28] and therefore is not considered in this work. Because each core of the CMP processor is small, it can issue/execute dependent instruction back-to-back. The SMT processor also supports the same overall issue width and threads.

Results show that the IPC of SMT-ideal is more than twice that of the CMP on single programs. The CMP is essentially reduced to a two-issue processor on a single program, leaving the other three cores idle. The SMT fetches and executes up to eight instructions from that program, and therefore has much higher performance than the CMP. When the increased delay in the SMT is considered, SMT still outperforms the CMP on single program. Even with the two-clock extra latency in the issue/execution loop, the SMT-D2 is 27% faster than the CMP.

The CMP performance doubles on two programs because the programs are executed in two cores in parallel. SMT-ideal is still faster than CMP at this stage, but the improvement is reduced from 103% to 66%. However, SMT-D2 is slightly slower than the CMP due to the two-clock latency in the critical loop. When there are four programs,

the CMP has close performance to the SMT-ideal but is 36% and 50% faster than SMT-D1 and SMT-D2, respectively.

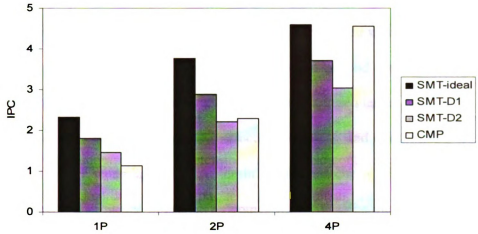


Figure 4.6. Performance of SMT vs. CMP.

The poor performance of the CMP processor on single thread is due to the decentralization of the front-end pipeline including instruction fetch and decoding. However, the hardware complexity of the instruction cache, which dominates the front-end size, scales linearly with the fetch width; and the decentralization does not reduce its hardware complexity significantly.

In fact, an instruction cache can fetch in a single cycle contiguous instructions up to the first predicted taken branches, or up to a maximum branch limit. The upper bound on fetch bandwidth is limited by the frequency of taken branches. Since the average number of instructions between taken branches is 12 on integer programs, and more on floating point applications [87], a wide single instruction cache can be shared by multiple cores accessing the shared cache alternatively. Even with a large capacity, a shared instruction cache hardware complexity would be in the same range as the hardware complexity of

the independent instruction caches of all cores. In addition, a branch predictor is needed to achieve high fetch bandwidth. The hardware cost of a shared branch predictor will also be in the same range as replicating it in each core.

The upper bound on fetch bandwidth can be further improved with a Trace cache [87-90] which provides the capability of fetching instruction beyond taken branches. Trace cache has the property of capturing dynamic instruction sequences, which makes it possible to store the information about instruction renaming and cluster assignments to simplify the register renaming stage. A line in the trace cache is fully specified by the starting address and sequence of branch outcomes which describe the followed path. The length of a trace cache line is limited by the maximum number of instructions and by the maximum number of basic blocks determined by the branch predictor bandwidth.

Therefore, with almost the same hardware complexity, a large single instruction cache has higher averaged fetch throughput than multiple small caches that have the same aggregate fetch width. When a multiprogrammed workload is encountered, the capacity is dynamically shared among different processes. When the workload is a single process, the processor can exploit whole hardware resource, achieving a much higher fetch rate than one slice of the cache.

The next section introduces the Adaptive Clustered Multithreaded Microarchitecture (ACMT) that is able to obtain high performance on both single and multiple programs.

4.2 An Adaptive Clustered Multithreaded Microarchitecture

It is favorable to have a centralized front-end and distributed back-end to achieve high IPC performance. The decentralization of a front-end limits the flexibility to allocate

hardware to instructions, resulting in low hardware utilization and poor performance on applications with low thread level parallelism. On the other hand, distributed back-end keeps the latency of the pipeline critical loop unchanged with technologies, supporting high instruction throughput.

From the hardware complexity perspective, decentralization of the frond-end does not reduce its hardware complexity significantly because the instruction cache, which dominates the front-end, scales linearly with the fetch width. In contrast, the back-end pipeline employs many multi-ports SRAM and CAM circuits, including the instruction queue, register file and load queue. Distribution of these structures could significantly reduce their hardware size and the delay. Therefore, it is reasonable to distribute the back-end pipeline and retain the centralized front-end to achieve overall high performance.

4.2.1 Microarchitecture

The adaptive Clustered Multithreaded microarchitecture (ACMT) was developed to explore the performance potential of an optimally distributed processor. It has a centralized front-end pipeline and distributed back-end as shown in Figure 4.7. The ACMT pipeline stages are described below.

4.2.1.1 Frond-end pipeline

The ACMT processor consists of a centralized front-end and distributed back-end pipeline. The front-end (fetch, decode, and rename) is shared by the backend of the pipeline. Instructions are fetched from the instruction cache or a trace cache. The amount of instructions that can be delivered is complicated by control hazards inherent in the execution of a program. In addition to cache misses and branch mispredictions, sustained

instruction fetch bandwidth is limited by frequent taken branches in the dynamic instruction stream.

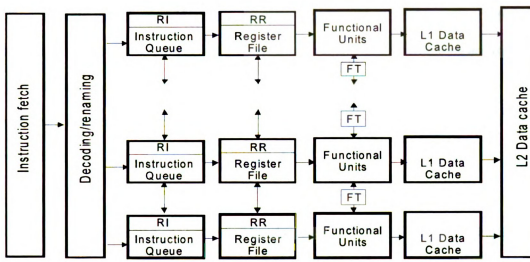


Figure 4.7. Clustered Multithreaded microarchitecture (ACMT).

The trace cache effectively addresses this issue by storing instructions in their dynamic program order in a separated cache. A long sequence of dynamic instructions, containing multiple blocks (possibly noncontiguous) was constructed as a trace over several cycles. When the trace is needed again, as predicted by the branch predictor, the entire trace is supplied in a single cycle to the decoder, thereby exceeding the taken-branch bandwidth limit.

When there are multi threads, one or two threads are granted to fetch instructions and a large block of continuous instructions are read at each cycle. The other non-blocking threads are selected to access the instruction cache in the next cycle. The fetched instructions are decoded and renamed before assigned to the segmented instruction queue.

Totally flexible routing from the decoder to instruction queue banks requires a large crossbar and high latency. The proposed processor steers instructions to an instruction queue based on the instruction's physical placement in the instruction buffer. This scheme results in less complexity but restricts the steering capabilities. The retire-time assignment approach [64] is used to address this problem. The instructions are reordered within a trace cache line at retire time so that they are fetched and sent to the desired position in the decoder. The delay to reorder the instructions is not in the critical path of the pipeline, and therefore does not affect clock rate.

4.2.1.2 Instruction steering

Results from section 3.3 show that the DEP steering policy has higher IPC performance than other steering policies. To reduce the delay of dependence analysis, a backward dependence steering algorithm [65] employing the DEP steering policy is used.

Backward dependency moves dependency analysis operations off the critical path by reconstructing the data dependence graph after instructions are dispatched. Steering decisions are made based on the reconstructed dependence graph. This scheme can be implemented with a history table. After an instruction is assigned to a cluster, new entries in the history table are created for the instruction's parent, and the instruction's cluster number is saved in these entries. The history table is looked up when new instructions are fetched. The new instructions are assigned to the cluster indicated in the table entry when there is a hit.

4.2.1.3 Instruction queue

The banked instruction queue is modified and used in this design. The total instruction queues of all clusters can be viewed as a banked instruction queue with each

bank corresponding to the instruction queue of a cluster. Because instructions from a thread can be steered to any cluster to maximize hardware utilization, dependent instructions may be located in different clusters and communication between different instruction queues are needed.

The instructions that need tags from other clusters are sent to RI which is an IW-port CAM inside of the instruction queue shown in Figure 4.7. Only RI receives operand tags from other cores. The instruction queue has issue width determined by the number of function units of the cluster.

Bit "GT" described in Chapter 3 is used in each entry indicating if dependent instructions are at other clusters. Destination tags of issued instructions are only sent to other clusters when bit "GT" is set. The distance between the central decoding/renaming logic and each cluster is different. Due to the wire delay, decoded instructions might arrive to the destination cluster in different order from when they are renamed. There are three cases:

1) Parent instructions and dependent instructions are steered to the same cluster. Then the latency from the rename logic to the cluster just increases the pipeline stages and thus increases the branch misprediction penalty.

2) Parent instruction is steered to the cluster close to the rename logic, and the dependent instructions go to distant clusters. Then the parent instruction is issued without any extra latency but the dependent instructions would wakeup with a multiple clock penalty.

3) Parent instruction is steered to the distant cluster, and the dependent instruction goes to the cluster close to the rename logic. Then the wakeup of both instructions are delayed.

Wire delay needs to be considered when updating bit "GT". This bit is reset when an entry is allocated in the instruction queue, and it is updated if dependent instructions fetched in the later cycles are assigned to other clusters. Due to the wire delay, the producer might have been issued by the time the updating information arrives. Because the producer is not aware of the new dependent instruction steered to other clusters, it only sends its destination tag to its own cluster when issued, causing deadlock.

Our approach to address this issue is to keep the issued instruction in the instruction queue for DN cycles, where DN is the delay in clock cycles between clusters. This approach increases the capacity pressure on the instruction queue size. Assuming the cluster issue width is IWi, then the instruction queue keeps at least DN*IWi issued instruction. An alternative is to move these instructions to another table. The table only stores the latest instructions and can be implemented with a FIFO structure. Its hardware capacity and complexity is much smaller than the instruction queue, and therefore doesn't compromise the cycle time.

4.2.1.4 Data path (Register File and Function Units)

In the register renaming stage, if the operand value of the new instruction is already ready in other cluster's register file, this data will be read and copied to the RR of the cluster to which the new instruction is assigned. The issued instruction read the register file and RR in parallel in order to fetch operand data. Simulations show that only a small RR is needed and therefore its delay is not critical.

A table FT is added at the functional units of each cluster. Each entry of FT contains two fields: tag and cluster number. The tag field stores the register tag. A FT entry is allocated at the dispatch stage for the instruction that needs data from other clusters. After

the instruction is renamed, if its source operand will be generated from other cluster, a FT entry in the producer cluster is allocated. The tag field is set to be the source tag of that instruction, and cluster number field is set to be the consumer cluster. After an instruction is executed, its destination tag is checked with the FT tag field. If there is a match, the results are forwarded to the destination core.

The wire delay of transmitting tags from the source to the destination cluster is designed to be the same as the wire delay of forwarding data, so that the instructions at the destination cluster can wakeup, issue and read operand values from the bypass network. The FT entry is allocated at the register renaming stage. The wire delay from the register renaming logic to the FT of the producer cluster is not critical because it is smaller than the total latency of issuing and executing an instruction.

4.2.2 Adaptive Hardware Allocation

4.2.2.1 Single Program Workload

When there is only one running program, the program can access all clusters to achieve high performance. The whole processor operates as a SMT with a decentralized back-end. While more clusters could provide more hardware to exploit the ILP in the program, it has higher communication overheads which offsets or even exceeds the benefits obtainable with extra clusters. Adjusting the number of available clusters according to the dynamic requirement of the program could possibly result in high performance and high energy efficiency.

The policy to adjust the number of active clusters depends on the particular goals of the applications, such as maximizing performance, minimizing the energy dissipation or

energy-delay product. In this work, the object function is to select the optimal number of active cluster to achieve maximal performance.

For every T1 cycles, the processor starts to search for the optimal configuration. The IPC of the different configurations is sampled for T2 cycles, and the configuration with the highest IPC is selected for the next T1 cycles. Parameter T2 need to be large enough to allow the processor state to be updated based on the cluster configuration. Increasing T2, however, would increase the time when the processor operates at an un-optimal configuration, resulting in performance degradation. Parameter T1 determines the algorithm's overhead and the delay to respond to the dynamic requirements of programs. To allow the processor to change the configuration during T1 cycles, the cluster search procedure can be triggered by a significant variation of the number of branches or memory references. Simulation results show the algorithm generates stable results for a wide range of T1 and T2.

Disabling an extension cluster can be realized by not assigning any new instructions to the cluster at the steering stage. After all of the instructions in the instruction queue are executed, the disable cluster can be shut down completely to save power. When an extension cluster is just added into the pipeline, the steering logic still assigns the consumer instructions to clusters of the producer instruction. The new independent instructions are assigned to the new cluster and their subsequent dependent will move to the new cluster in later cycles.

The instruction fetch and rename units are shared by all clusters. When a cluster is shutdown, the width of the backend pipeline is reduced. To match the width of the backend pipeline, the instruction fetch and decoding rate could be reduced accordingly to

save power without significantly affecting the performance. This can be implemented by gating the instruction fetch logic to provide the required fetch rate.

4.2.2.2 *Multi Programs Workloads*

When there are two active programs, each program takes control of two clusters only. The processor operates as a two four-issue SMT and each SMT has two segmented back-end. Limiting the number of clusters that can be accessed by one program yields a lower communication overhead than distributing the program to all clusters. High hardware utilization is obtained by running two programs in parallel.

When there are four programs, each program is assigned to one cluster. The processor operates as a four-core CMP. There is no communication between clusters, and the instruction from each program executes independently.

4.2.3 *Simulation Results*

The SimpleScalar have been modified to model SMT, CMP and ACMT. The benchmarks are SPEC CPU2000 INT. The processor architectural parameters are summarized in Table 4.1.

TABLE 4.1 MICROARCHITECTURE CONFIGURATIONS.OF SMT, CMP, AND ACMT

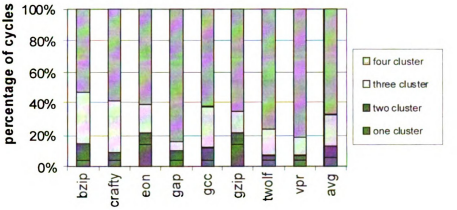
	SMT	4-core CMP	4-cluster ACMT
Fetch/dec rate	8	2 / core	8
Issue rate	8	2 / core	2 /cluster
Register file size	256	64 /core	64 /cluster
Integer ALU	8	2 /core	2 / cluster
Instruction queue size	64	16 / core	16 /cluster
LSQ size	64	16 / core	16 /cluster
D1/I1 cache	32k, 2-way, 2-cycle lat.	8k, 2-way, 1-cycle lat. / core	8k, 2-way, 1-cycle lat. / cluster
D2 cache	2M, 4-way, 8-cycle lat.	512k, 4-way, 8-cycle lat. /core	512M, 4-way, 8-cycle lat. / cluster
Main Memory	100 cycle	100 cycle	100 cycle

4.2.3.1 *Single Program*

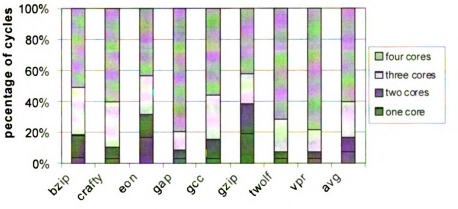
Based on the results in previous sections, the four cluster configuration has overall higher performance than other alternatives. Therefore, four two-issue clusters are employed in the ACMT. Figure 4.8a shows the clusters activity. The adaptive mechanism allows the processor operating with only a subset of the clusters. In benchmark bzip, the processor shuts down one cluster for 48% of the cycles due to the limited ILP in the program. In the case of benchmark gap, the processor activates all cores almost all of the cycles in order to exploit the ILP available in the program. On average, one cluster is shut down for 32% of the cycles. The processor rarely selects only one cluster, showing that the benefit of more execution bandwidth is larger than that of the increased communication overhead.

As the communication overheads increase, the processor operates with fewer clusters in more cycles. Figure 4.8b shows the cluster activity assuming a two-cycle communication delay between clusters. One cluster is shut down for about 40% of the cycles, increased from 32% with a 1-cycle communication delay. Still, the processor turns on at least two clusters for most of the cycles in order to achieve high performance.

The eight-cluster configuration, which has higher performance than four clusters before the functional units are distributed, has more IPC degradation with distributed functional units. The reason is that the eight-cluster processor has a lower functional unit utilization due to workload imbalance. In addition, the amount of data forwarding on eight clusters is also higher than on four clusters.



(a). One-cycle communication delay.



(b) Two-cycle communication delay.

Figure 4.8. Cluster activity in a 4-cluster ACMT (T1=480k, T2=16k in the cluster adjustment algorithm).

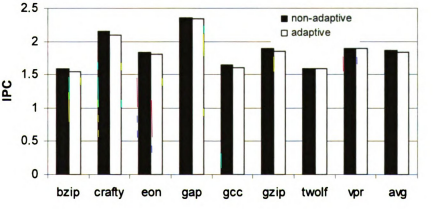


Figure 4.9. The IPC Performance of the ACMT. The non-adaptive ACMT has all clusters active all the time.

The IPC performance of the adaptive processor is shown in Figure 4.9. Shutting down some clusters dynamically has performance very close to utilizing the full hardware capacity all the time. There is a slight performance degradation due to configuration exploring that requires the processor to operate with a non-optimal configuration in order to sample its IPC. This result is not sensitive to the values of the T1 and T2 parameters. A stable performance can be achieved when T1 varies from 8k to 32k and T2 from 320k to 1M cycles.

The increased wire delay in advanced technology affects the IPC of both SMT and ACMT, but in different ways. In a SMT processor, multiple clocks are needed to access the increasingly large centralized hardware, resulting in more execution latency for all instructions. The ACMT processor support technology scaling inside clusters but suffers from communication delay between distributed hardware. However, only dependent instructions located on different clusters are affected by this communication delay. CMP supports technology scaling and have no communication overheads.

The IPC performance comparison between SMT, CMP, and ACMT is shown in Figure 4.10. The horizontal axis represents the delay in clock of the execution loop in SMT. It is the minimal number of clocks to execute two dependent instructions. For ACMT, the delay is the delay to forward instructions results between clusters. The IPC of both SMT and ACMT degrades when the delay increases. The CMP supports clock scaling and has no communication; therefore, its performance does not change with this variable.

The SMT with delay=0 represents an ideal SMT processor. Its performance degrades quickly when the execution loop delay increases. The ACMT with delay=0 represents a

decentralized SMT without a communication penalty, and has lower IPC than the ideal SMT. However ACMT outperforms the SMT when the delay is larger than zero. In addition, the delay degrades the ACMT performance much slower than the SMT because only a small portion of instructions (dependent instruction assigned to different clusters) are affected in the ACMT. The CMP has the worst performance because of its low hardware utilization on single program.

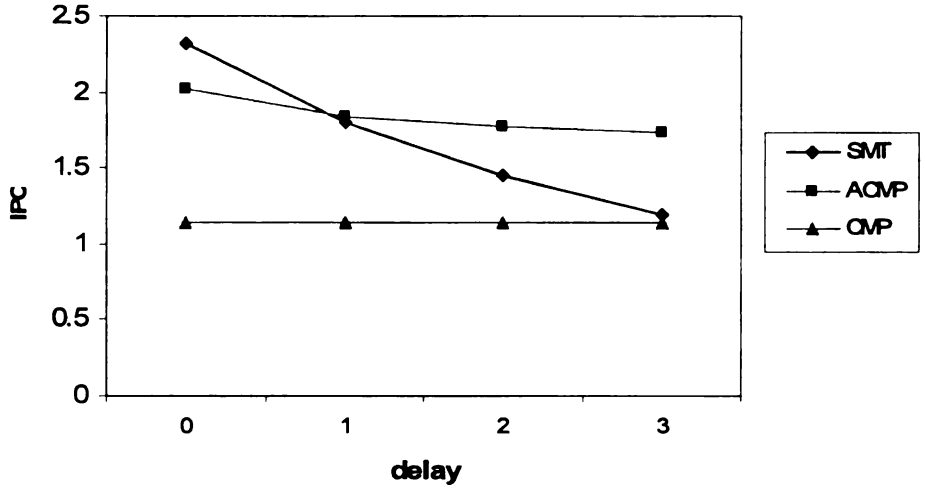


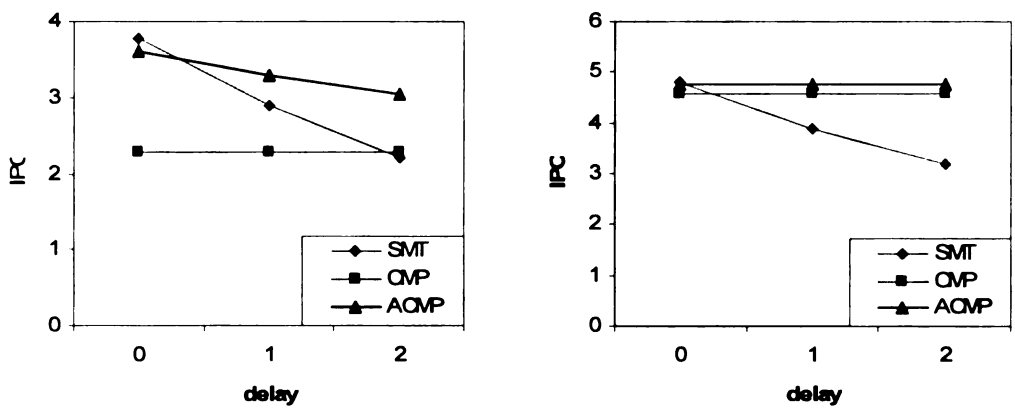
Figure 4.10. The IPC performance of SMT, CMP, and ACMT on single program.

4.2.3.2 *Multiple Programs*

Figure 4.11 shows the IPC performance of SMT, CMP and ACMT on multiple programs. Without considering the increased delay of large centralized hardware and wires between clusters, ACMT has slightly less IPC than the ideal SMT on two-program workloads (Figure 4.11a). However, ACMT has significantly higher performance than the SMT when the delay penalty of centralized hardware and communication wire are

considered. The CMP was still outperformed by SMT and ACMT, but the performance difference decreases to 25% of ACMT, reduced from 35% on single programs.

When there are four programs, the ACMT works similar to the CMP processor. Since one thread is attached to a back-end cluster, there is no data forwarding between clusters. The centralized Trace cache in the ACMT can fetch instructions beyond taken branches of a single program. It has higher averaged fetch throughput than multiple small caches that have the same aggregate fetch width. Therefore, ACMT has slightly higher performance than the CMP. Both ACMT and CMP have much higher performance than the SMT that has a one clock delay penalty in its critical execution loop.



(a) two program workloads (b) four program workloads

Figure 4.11. The IPC performance of SMT, CMP, and ACMT on multiple programs.

Figure 4.12 shows the performance of the three processors on all types of workloads assuming a two-clock-delay on both SMT and ACMT. ACMT has a consistently higher performance on all workloads. The ACMT requires communication resources between distributed cores, which results in a high hardware utilization and therefore high

performance when the parallelism in applications is low. When the thread level parallelism is high, the ACMT is similar to a CMP processor with an shared Trace cache.

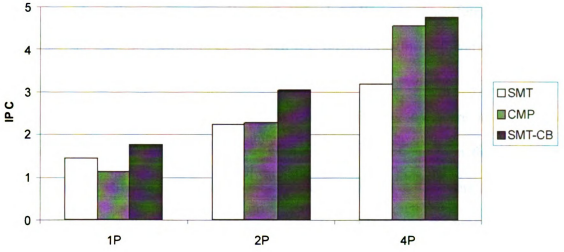


Figure 4.12. Performance of SMT, CMP and ACMT (assuming two-clock-delay for both the SMT and the ACMT)

4.3 Summary

Multi-core processors provide a scalable hardware platform for multi-threaded applications. The overall high instruction throughput $\sum_{t} IPC_i \times clock\_rate$ is achieved by improving the clock rate and the summation factor (high layout area efficiency and therefore more cores with the same hardware budge) but sacrificing the IPC of individual threads. SMT, on the other hand, supports high IPC of individual threads at the cost of large layout area and a slow clock rate.

An ACMT processor achieves higher performance than both SMT and CMP having the same issue width. An ACMT processor is a SMT with clustered back-end that provides a high performance execution engine. The dependent instructions in the same cluster can be executed back-to-back to achieve high IPC performance. In addition,

ACMT is similar to a CMP processor with a shared instruction fetch engine. Advanced instruction fetch techniques (such as Trace cache) is used to achieve higher averaged fetch throughput than multiple small caches in each CMP cores. Therefore, the ACMT processor supports high clock rates and high IPC on applications with various ILP and TLP.

The centralized front-end of the ACMT allows flexible hardware allocation to instructions. Some of the clusters are deactivated if contributions from the additional clusters are offset by the increased communication overheads. All clusters are activated to meet the peak requirements of programs. This feature improves the processor energy efficiency when ruing single threads.

ACMT trades chip area to achieve high performance on various workloads. An ACMT needs extra communication hardware compared to a CMP. Data forwarding between clusters takes large routing area that could have been used to accommodate more cores in a CMP processor. The communication hardware is not used in applications when their TLP is high. Therefore, ACMT is suitable for applications having limited TLP and requiring high performance on single program.

5 SUMMARY AND FUTURE WORK

Advances VLSI technology provide more hardware resources that can be utilized in a superscalar processor to exploit the ILP and the TLP of the applications. The additional transistors resources have been harnessed to increase the pipeline depth and processor issue width to achieve high instruction throughput. A wide-issue superscalar processor maintains an increasingly large number of in-flight instructions in pipelines to exploit parallelisms in applications. It employed multi-port SRAM and CAM circuits in many pipeline stages and the circuit complexity typically increases with the issue width in the conventional centralized design. Because the delay of the SRAM and CAM does not scale with technologies, the critical path delay of the processor increases, causing significant performance degradation.

Optimization of the hardware in the critical pipeline stage can remove the performance bottleneck with minimal impact on other pipeline stages. Our work on the instruction queue shows that there is large room for performance improvement through hardware optimization. However, trying to push performance improvements by optimizing each of many pipeline stages is not an ideal methodology because it requires significant design effort that does not necessarily scale with advances in fabrication technology. Multi-core processors provide a scalable hardware platform for applications with high thread-level parallelism. However, they have low hardware utilization and fail to yield high performance on single threads, which is important in many applications. This thesis presents a solution that achieves high performance on a diverse set of applications.

5.1 *Summary of Contributions*

o *Multiple approaches have been developed to optimize the delay of the instruction queue of a superscalar processor.* We have designed a banked instruction queue that supports high clock rate with modest IPC penalty. The banked design requires 50% hardware resources with only 4% IPC degradation on single program workload and virtually no IPC degradation when running multiple programs, providing 28% improvement in overall instruction throughput compared to a conventional instruction queue. In addition, our design provides easy tradeoffs between hardware resources and the IPC. With 25% of the hardware resources, the banked instruction queue causes 18% IPC degradation but achieves 36% higher instruction throughput owning to the improvement in delay. This work has conducted thorough simulations at both the circuit level and the architectural level, enabling evaluation of the overall performance. A microarchitectural simulator has been developed to model distributed microarchitecture running multi-threaded workloads based on the SimpleScalar tool.

In addition, two coding techniques have been developed to reduce individual delay components on the critical timing path of the instruction queue. One uses a linear code to increase the Hamming distance between operand tags, significantly reducing the instruction wakeup delay. This method can be combined with the banked instruction queue to achieve the maximum performance. The other uses one-hot code that can remove the OR gate and the tag read delay from the critical path, resulting in significant delay improvement in a small instruction queue.

o *A methodology has been developed to identify the optimal hardware distribution scheme of wide-issue superscalar processors.* We start with a SMT processor and analyze

the effect of hardware distribution of each of the major pipeline stages. We have shown that the distributions of the instruction queue, the register file, and the L1 data cache can achieve significant reduction in circuit size and delay with relatively small IPC degradation. This methodology provides a link between SMT and CMP, which represent two opposite sides of the design spectrum. The performance relative to the SMT and the CMP processors has been analyzed on single and multiple program workloads. Results show that, even with multiple clock latencies in the critical pipeline loop, an 8-issue SMT still outperforms an 8-issue 4-core CMP by 27% on single program. However, the CMP has a 42% higher IPC than the SMT on multi-program workloads with the advantage of small layout size and high clock rate.

- *This thesis introduces the Adaptive Clustered Multithreaded (ACMT) Microarchitecture that achieves higher overall performance than either SMT or CMP architectures.* The ACMT employs clustered back-end to support high execution bandwidth; it utilizes a centralized frond-end to obtain a high instruction fetch bandwidth and high hardware utilization. Simulations show that an 8-issue 4-cluster ACMP running single programs has 22% and 55% higher IPC than an 8-issue SMT and 8-issue 4-core CMP, respectively. The ACMP outperforms the SMT and the CMP by 49% and 4%, respectively, on four-program workloads.

We have shown that single threads have large ILP that can yield to higher IPC with more cluster resources. When the ILP in a portion of program is high, the contribution of additional clusters (higher execution bandwidth) exceeds the increased performance degradation due to cluster communication overheads. We also show that the ILP varies

significantly within programs. Shutdown a set of clusters dynamically reduces the communication overhead, and has minimal affect on the overall performance.

We have shown that the ACMT is suitable for applications with dynamic TLP and ILP. The centralized front-end is flexible to allocate hardware resources to exploit both TLP and ILP. When the TLP is high, a thread is assigned to a reduced subset of clusters. A smaller subset of clusters has less communication overhead, and multiple subsets of clusters yields overall higher performance. When the TLP is low, a larger subset of clusters provides higher execution bandwidth to exploit the ILP in applications.

5.2 *Future Work*

Based on the results of this thesis, we suggest the following directions for future research.

- *Performance comparison of SMT, CMP, and ACMP under the same hardware budgets.* ACMT needs more wire area for communications between clusters. The instruction queue, register file and functional unit of each cluster also consume transistors to send/receive data from other clusters. Under the same hardware budget, more cores could be accommodated in a CMP processor which could result in higher instruction throughput for applications with high TLP.
- *Heterogeneous Clustered Microarchitecture.* For homogeneous cluster designs (each cluster is the same size), the granularity of the clusters determines performance of each cluster and the overheads of the whole processor. Fine-grained clusters support high clock rate but have low IPC performance on applications with low TLP and high ILP. Course-grained clusters have high IPC on the above applications but the advantage diminishes when the application TLP increases. The relatively low clock rate of course-

grained clusters may result in lower instruction throughput than a fined-grained design. A heterogeneous design employing clusters with different performance may be more efficient than the homogeneous design to adapt to the application requirements. By assigning the applications with high ILP to a large cluster and high TLP applications to small clusters, an overall higher instruction throughput could possibly be achieved.

o *Optimization of Energy Efficiency.* This thesis aims to maximize the processor instruction throughput with increased hardware budget. As power has become a critical design limitation, it would be interesting to explore the optimal design for maximizing processor energy efficiency.

The energy efficiency of homogenous clustered microarchitecture has been analyzed in [25]. The performance and energy dissipation of homogeneous clustered microarchitectures respond radically and inversely to the cluster granularity, making it difficult to minimize the energy-delay product. Fined-grained clusters need a large number of small clusters and incur significant performance degradation resulting in lower energy efficiency as measured by energy-delay metric. Using a larger cluster would reduce the delay factor but may still result in higher energy-delay due to increased energy dissipation within each cluster. A heterogeneous design that incorporates clusters with different performance and energy dissipation characteristics could potentially achieve an overall lower energy-delay than homogeneous clustered designs. Large clusters could be used to provide higher performance (less delay) and small clusters could be used to achieve low energy dissipation.

APPENDIX: MICROARCHITECTURE SIMULATOR

SimpleScalar [66] was created by Todd Austin to simulate real programs running on a range of modern processors and systems. It contains a detailed, dynamically scheduled processor model that supports out-of-issue, speculative execution. In addition to simulators, the SimpleScalar includes performance visualization tools, statistical analysis resources, and debug and verification infrastructure. Many tools have been developed based on the SimpleScalar infrastructure for research and instruction.

The microarchitecture simulator used in this thesis work was developed based on the SimpleScalar out-of-order simulator. Because SimpleScalar only models single core and single thread microarchitecture, it was modified to model SMT and cluster microarchitecture to meet the simulation requirements of this thesis.

1. Support SMT Processor Architecture.

i. Instruction Fetch.

In SimpleScalar, the instructions of simulated program are loaded into "*mem*" array by calling function *MD_FETCH_INST*. To support SMT, two-dimension array are used to store the instructions of each thread, and function *MD_FETCH_INST* is reused to load the instructions of each thread. The global variable "*active_thread*" selects the threads to be processed.

MD_FETCH_INST(inst, mem, fetch_regs_PC);

➔*:MD_FETCH_INST(inst,threads[active_thread].mem, threads[active_thread].fetch_regs_PC);*

The instruction fetch unit implements an 8-2 fetch algorithm. Two threads are selected to fetch instructions each cycle. Assuming the processor issue width is IW, up to IW instructions can be fetched from each selected threads each cycle. The thread that has

L2 cache miss (set *threads.stall* variable to "1") is stalled from instruction fetch. All un-stalled threads are selected in rotation.

ii. Instruction Renaming

SimpleScalar has two structures variables (*regs* and *spec*) representing the architectural and physical register file of the processor. In SMT, each thread has its own architectural register file, but all threads share the same physical register file. To reuse the SimpleScalar code, each thread still associates with its architecture register file (*regs*) and physical register file *(spec*). Another scalar variable *gRUU_num* is used to represent the total usage of processor physical register file. If the value of *gRUU_num* is larger than the size of the physical register file, all threads are stalled from register renaming until some instructions are executed & retired. A new structure thread is used to describe threads.

```
struct thread {

        struct regs_t *regs; //architectural register file
        struct regs_t *spec; //physical register file
        struct mem_t *mem; // hold instructions of all threads
        struct cache_t *cache_il1;
        struct cache_t *cache_il2;
        struct cache_t *cache_dl1;
        struct cache_t *cache_dl2;
        struct cache_t *itlb;
        struct cache_t *dtlb;
        struct bpred_t *pred;
        ......

}
```

iii. Instruction Issue & Execution

Issue and execution of instructions in a SMT processor is the same as in a superscalar processor. All instructions wait in a shared instruction queue until they are ready to issue. The ready instructions from all threads compete for execution units with age-based

select policy. For each thread, the ready instructions are inserted into a queue (*readyq[active_thread]*) based on the instruction ages. In each cycle, the top entries of each queue are issued until the total issued instructions reach the processor issue width. This prevents a single thread to occupy all execution units and block execution of other threads.

All threads share a load/store queue for memory access instruction. In this SMT model, each thread has its own instruction and data cache. The shared load/store queue interfaces with multiple separated data cache.

iv. Instruction Retirement

The executed instructions of each thread remain in a single reorder buffer until they are retired. A segment of the reorder buffer is assigned to a thread. Similar to the instruction issuing, the top entry of each reorder buffer segment is allowed to retire. All reorder buffer segments are searched each cycle, so that the executed instructions from each threads can be removed from processor hardware fairly.

2. Support Clustered Microarchitecture

The above SMT processor simulator was further modified to model SMT clustered processors. The single instruction queue, register file, and execution unit pools were divided into multiple segments. Multiple steering algorithms were implemented. User chooses the cluster configuration and steering algorithm in the configuration file. All other units, including instruction fetch, decoding, and instruction retirement are not affected.

i. Segmented Instruction Queue

The instruction dependency analysis is modeled in function *ruu_link_idep*. After a new instruction is decoded, it is added into the dependent list of the parent instruction. When an instruction is issued, its dependent list is looked up, and all instructions on the list update their operand status to ready. To model the segmented instruction queue, a decoded instruction is associated with an instruction queue. The new instruction was assigned to a queue based on the steering algorithm. The implemented steering algorithms are described in the next section. If the designed queue is full, the new instruction is assigned to the queue with the least number of entries.

Instruction wakeup process is modeled in function *ruu_writeback*. When an instruction is issued, the cluster number of its dependent instructions is compared with that of the issued instruction. If they are different, the dependent instructions are sent to the ready queue with the hold flag set true. The instructions with a true hold flag remain in the queue for cycles specified in the configuration file. Otherwise, they complete for the execution units and are removed from the queue if they win.

ii. Steering Algorithm

The steering algorithms are modeled in function *ruu_link_idep*. For dependence algorithm, if its parent instructions are still in the ready queue, the instruction is assigned the same cluster value as its parent instructions. The cluster number of the second operand overwrites that of the first operand. If the instruction has no parent instructions or all of them have been issued, then the instruction is assigned to the queue with the least number of entries. In our simulation, average 60% instructions are in this case. The other three steering algorithms are straightforward in the code.

iii. Segmented Physical Register file

The results of executed instruction are stored in the physical register file. After the instruction retires, the physical register is marked as a logic register. For an instruction still in the ROB (not retired), the cluster number that is assigned to the instruction after it is steered is also used to indicate the segmented physical register file. For the instruction not in the ROB (retired), its cluster number variable doesn't exist anymore. Therefore, additional table (*struct *log_reg*) is used to keep track the location of the retired instructions.

```
struct log_reg_table{
      md_addr_t pc; //instruction PC
      int lsq;  // is in LSQ or not.
      int cluster; //physical register file

}log_reg[LOG_REGISTER_NUMER]
```

After a new instruction gets its cluster number at the instruction steering stage, the table log_reg is updated by the clustered number. The updated entry is indexed by the destination operand of the new instruction.

iv. Segmented Execution Units

The segmented execution units is modeled in module *ruu_issue*. In each cycle, ruu_issue() checks the ready instruction queue of each cluster. The top entry in the queue requests for execution units in its segment and the total number of allocated executed units is countered. If the maximum quote has been met, all ready instructions of this cluster are reinserted to the queue that will be checked in the next cycle.

v. Banked Data Cache

The banked data cache is modeled in function *ruu_link_idep*. If a new instruction is not a memory access instruction, then the instruction is assigned to a cluster based on the specified steering algorithm. If the instruction is a load/store instruction, the instruction is

assigned to a cluster randomly. For a two-operand instruction memory instruction, it can be assigned to two different clusters. In our model, the instruction is assigned based on the result of the steering algorithm at X% cycle. The value of X is the data cache bank prediction accuracy. The results in [74] were used to model the prediction accuracy: 85% for four banks and 84% for eight banks.

BIBLIOGAPHY

1. J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors," *Proceeding of the IEEE*, Vol. 83, No. 12, December 1995.

2. M. Johnson, "Superscalar Design," *Englewood Cliffs*, NJ:Prentice-Hall, 1990.

3. J. A. Fisher and B. R. Rau, "Instruction-Level Parallel Processing," *Science*, pp. 1233-1241, September 1991.

4. T. M. Conte, P. M. Mills, K. N. Menezes, and B. A. Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates," *Proc. 22nd Annual International Symposium on Computer Architecture*, pp. 333-344, June 1995

5. B. R. Rau, D. W. L. Yen, W. Yen, and R. Towle, "The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs,'' *IEEE Computer*, vol. 22, pp. 12-35, January, 1989.

6. D. Kaeli and Pen-Chung Yew, "Speculative Execution in High Performance Computer Architectures," *CRC press*, 2005.

7. C. H. Perleberg and A.J. Smith, "Branch Target Buffer Design and Optimization," *IEEE Trans. on Computers*, 42(4), pp. 396-412, Apr. 1993.

8. Q. Jacobson, al et., "Path-based Next Trace Prediction," *Proc. 30th Int'l Symp. Microarchitecture*, 1997.

9. A. Uht and V. Sindagi, "Disjoint Eager Execution: An Optimal Form of Speculative Execution," *Proc. 28th Int'l Symp. Microarchitecture*, 1995.

10. T.F. Chen, "Supporting Highly Speculative Execution via Adaptive Branch Tree," *Proc. 4th Int'l Symp. HPCA*, February1998.

11. A. Roth et al., " Dependence Based Prefetching for Linked Data Structures," *Proc. 8th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1998.

12. B. Calder and G. Reinman, "A Comparative Survey of Load Speculation Architecture," *Jounal of Instruction-Level Parallelism*, 1(39), January 2000.

13. F. Gabbay and A. Mendelson, "Using Value Prediction to Increase the Power of Speculative Execution Hardware," *ACM Trans. Computing Systems*, 16(3), 1998.

14. M.H. Lipasi and J.P. Shen, "Exceeding the Dataflow Limit via Value Prediction," *Proc. 29th Int'l Symp. On Microarchitecture*, December 1996.

15. J. Gonzalez and A. Gonzalez, "Control-Flow Speculation through Value Prediction for SuperScalar Processors," *Int'l Conf. On Parallel Architecture and Compilation Technique*, September 1999.

16. D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22th Int'l Symp. Computer Architecture*, pp. 392-403, 1995.

17. D. Koufaty, and D. T. Marr, "Hyperthreading Technology in the Netburst Microarchitecture," *IEEE Micro*, vol. 23, No. 2, pp. 56-65, March, 2003.

18. D. Tullsen, et al., "Handling Long-latency Loads in a Simultaneous Multithreading processor," *Proc. 34th Int'l Symp. On Microarchitecture*, pp. 318-327, 2001.

19. D. Tullsen, et al., "Exploiting Choice: Instruction Fetch and Issue on An Implemnetable Simultaneous Multithreading Processor," *Proc. 23th Int'l Symp. On Computer Architecture,* pp. 191-202, 1996.

20. Z. Chishti and T.N. Vijaykumar, "Wire delay is not a problem for SMT," *Proc. 31th Int'l Symp. On Computer Architecture*, May 2005.

21. F. J. Cazorla, al et., "*Dynamically Controlled Resource Allocation in SMT Processors*," *Proc. 36th Int'l Symp. Microarchitecture*, 2004.

22. G. S. Sohi, S. E. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. 34th Int'l Symp. on Computer Architecture*, pp. 414-425, June 1995.

23. E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace processor," *Proc. 30th Int'l Symp. on Microarchitecture*, pp. 138-148, November, 1997.

24. K. I. Farkas, P. Chow, N. P. Jouppi and Z. Vranesic, "The Multicluster Arcitecture: Reducing Cycle Time Through Partitioning," *Proc. 30th Int'l Symp. on Microarchitecture*, pp. 149-159, November, 1997.

25. V. Zyuban and P. Kogge, "Inherently Lower-Power High Performance Superscalar Architecture," *IEEE Transactions on Computers*, Mar 2001.

26. P. Kongetira, "A 32-way Multithreaded SPARC Processor," *Hot Chips 16*, http://www.hotchips.org/archive/, 2004.

27. T. Takayanagi, al et., "A Dual-Core 64-bit UltraSPARC Microprocessor for Dense Server Applications," *IEEE J. Solid-State Circuits*, Vol. 40, No. 1, January 2005.

28. R. Kalla, B. Sinharoy, and J. M. Tendler, "IBM Power5 Chip: A Dual-Core Multithreaded Processor," *IEEE Micro*, vol. 24, No. 2, pp. 40-47, March, 2004.

29. J. Burns and J. L. Gaudiot, "Area and System Clock Effects on SMT/CMP Throughput," *IEEE Trans. on Computers*, Vol. 54, No. 2, February 2005.

30. D. Sima, "The Design Space of Register Renaming Techniques," *IEEE Micro*, vol. 20, No. 5, pp. 70-83, 2000.

31. V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitecture," *Proc. 27th Int'l Symp. on Computer architecture*, pp. 248 –259, June, 2000.

32. R.E. Kessler et al., "The Alpha 21264 architecture," *Proc. Int'l Conf. On Computer Design*, pp. 90-95, December 1998.

33. C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The AMD Opteron Processor for Multiprocessor Servers," *IEEE Micro*, vol. 23, No. 2, pp. 66-76, March, 2003.

34. H. Ando, al et., "A 1.3-GHz Fifth-Generation SPARC64 Microprocessor," *IEEE J. Solid-State Circuits*, vol. 38, No. 11, pp. 1896-1905, Nov. 2003.

35. N. S. Kim and T. Mudge, "The Microarchitecture of a Low Power Register File," *Proc. Int'l Symp. Low-Power Electronics and Design (ISLPED)*, pp. 384-389, 2003.

36. I. Park, M. D. Powell, and T. N. vijaykumar, "Reducing Register Ports for Higher Speed and Lower Energy," *Proc. 35th Int'l Symp. Microarchitecture*, pp. 171-182, 2002.

37. R. Balasubramonian, S. Dwarkadas, and D. Albonesi, "Reducing the complexity of the register file in dynamic superscalar processors," *Proc. 34th Int'l Symp. Microarchitecture*, pp. 237-249, 2001.

38. A. Gonzalez, J. Gonzalez, and M. Valero, "Virtual-Physical Register," *Proc. 4th Int'l Symp. High Performance Computer Architecture*, 1998.

39. K. L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham, "Multiple-banked Register File Architectures," *Proc. Int'l Symp. Computer Architecture*, pp. 316-325, June 2000.

40. E. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad, "A Fully Bypassed Six-Issue Integer Datapath and Register File on the Itanium-2 Microprocessor," *IEEE J. Solid-State Circuits*, vol. 37, No. 11, pp. 1433-1440, Nov. 2002.

41. R. K. Krishnamurthy, et al., "A 130-nm 6-GHz 256 x 32 bit Leakage-Tolerant Regsiter File," *IEEE J. Solid-State Circuits*, vol. 37, No. 5, pp. 624-632, May. 2002.

42. G. Hinton, et al., "A 0.18-um CMOS IA-32 Processor With a 4-GHz Integer Execution Unit," *IEEE J. Solid-State Circuits*, vol. 36, No. 11, pp. 1617-1627, Nov. 2001.

43. J. Leenstra, et al., "A 1.8-GHz Instruction Window Buffer for an Out-of-Order Microprocessor Core," *IEEE J. Solid-State Circuits*, vol. 36, No. 11, pp. 1628-1635, Nov. 2001.

44. J. Abella, et al., "Power-and Complexity-Aware Issue Queue Designs," *IEEE Micro*, vol. 23, No.5, pp. 50-58, September, 2003.

45. S. Palacharla, N. P. Jouppi, and J. Smith, "Complexity-effective Superscalar Processors," *Proc. 24th Int'l Symp. On Computer Architecture*, pp. 206-218, June 1997.

46. H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint Processing and Recovery: an Efficient, Scalable Alternative to Reorder Buffers," *IEEE Micro*, vol. 23, No.6, pp. 11-19, Nov, 2003.

47. A. R. Lebeck et al., "A large, Fast Instruction Window for Tolerating Cache Misses," *Proc. 29th Int'l Symp. Computer Architecture*, pp. 59-70, 2002.

48. T. Moreshet and R. I. Bahar, "Complexity-Effective Issue Queue Design under Load-Hit Speculation," *Proc. Workshop Complexity-Effective Design*, 2002.

49. M. Goshina et al., "A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors," *Proc. 33th Int'l Symp. Microarchitecture*, pp. 225-236, 2001.

50. M. Huang, J. Renau, and J. Torrellas, "Energy-Efficient Hybrid Wakeup Logic," *Proc. Int'l Symp. Low-Power Electronics and Design (ISLPED)*, pp. 196-201, 2002.

51. J. Stark, M. D. Brown, and Y. N. Patt, "On Pipelining Dynamic Instruction Scheduling Logic," *Proc. 32th Int'l Symp. Microarchitecture*, pp. 57-66, 2000.

52. P. Michaud, A. Seznec, "Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors, " *Proc. 4th Int'l Symp. on High-Performance Computer Architecture*, Janunary 2001.

53. D. Ernst, A. Hamel, and T. Austin, "Cyclone: A Broadcast-Free Dynamic Instruction Scheduler with Selective Replay," *Proc. 36th Int'l Symp. On Computer Architecture*, May, 2003.

54. S. E. Raasch, N. L. Binkert, and S. K. Reinhardt. "A Scalable Instruction Queue Design Using Dependence Chains," *Proc. 29th Int'l Symp. on Computer Architecture*, pp. 318-329, May 2002.

55. M. D. Brown, J. Stark, and Y. N. Patt, "Select-Free Instruction Scheduling Logic," *Proc. 34th Int'l Symp. Microarchitecture*, pp. 204-213, 2001.

56. S. Onder and R. Gupta. "Superscalar Execution with Dynamic Data Forwarding," *Int'l Conf. on Parallel Architectures and Compilation Techniques*, pp. 130--135, Oct. 1998.

57. D. Folegnani and A. Gonzalez, "Energy-Effective Issue Logic," *Proc. 28th Int'l Symp. on Computer Architecture*, pp. 230-239, June, 2001.

58. D. Ponomarev, G. Kucuk, and K. Ghose, "Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources," *Proc. 33th int'l Symp. Microarchitecture*, pp.90-101, 2001.

59. D. Ponomarev, al et., "Energy Efficient Comparators for Superscalar Datapaths," *IEEE Trans. on Computer*, vol. 53, No. 7, pp. 892-904, 2004.

60. E. Brekelbaum, J. Rupley II, C. Wilkerson, and B. Black, "Hierarchical Scheduling Windows," *Proc. 35th Int'l Symp. on Microarchitecture*, pp. 27-36, November, 2002.

61. J. M. Parcerisa, J. Sahuquillo, A. Gonzalez, and J. Duato, "Efficient Interconnects for Clustered Microarchitectures," *Int'l Conf. on Parallel Architectures and Compilation Techniques*, pp. 291--300, Oct. 2002.

62. R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Dynamically Managing the Communication-Parallelism Trade-off in Future Clustered Processors," *Proc. 30th Int'l Symp. On Computer Architecture*, pp. 275-287, June 2003.

63. A. Baniasadi and A. Moshovos, "Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors," *Proc. 33th Int'l Symp. on Microarchitecture*, pp. 337-347, December, 2000.

64. R. Bhargava and L. K. John, "Improving Dynamic Cluster Assignment for Clustered Trace Cache Processors, " *Proc. 30th Int'l Symp. On Computer Architecture*, pp. 264-274, June 2003.

65. R. Canal, J. M. Parcerisa and A. Gonzalez, "Dynamic Cluster Assignment Mechanisms," *Proc. 6th Int'l Symp. on High Performance Computer Architecture*, pp. 133-143, January, 2000.

66. T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, vol. 35, no. 2, pp. 59-67, February, 2002.

67. D. M. Brooks, V. Tiwari, and M. Maronosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimization," *Proc. 27th Int'l Symp. On Computer Architecture*, June 2003.

68. T. M. Austin and G. S. Sohi, "Dynamic Dependency Analysis of Ordinary Programs," *Proc. 19th Int'l Symp. on Computer architecture*, pp. 342 – 351, 1992.

69. M. B. Taylar, W. Lee, S. Amarasinghe and A. Agarwal, "Scalar Operand Networks: On-Chip Interconnect for ILP in Partitionad Architectures," *Proc. 10th Int'l Symp. on High Performance Computer Architecture*, January, 2003.

70. V. A. Singh, K. Sankaralingam, S. W. Keckler and D. Burger, "Design and Analysis of Routed Inter-ALU Networks for ILP Scalability and Performance," *Proc. 21th Int'l Conf. on Computer Design*, October, 2003.

71. R. Kuma, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," *Proc. 36th Int'l Symp. on Microarchitecture*, pp. 81-92, December, 2003.

72. S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylar, and R. Laufer, "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," *Proc. 26th Int'l Symp. On Computer Architecture*, pp. 28-39, May, 1999.

73. G. Z. Chrysos and J.S. Emer, "Memory Dependence Prediction Using Store Sets," *Proc. 25th Int'l Symp. On Computer Architecture*, pp. 142-153, June-July 1998.

74. H. Neefs, H. Vandierendonck and K. D. Bosschere, "A technique for high bankdwidth and deterministic low latency load/store access to multiple cache banks," *Proc. 32th Int'l Symp. On Computer Architecture*, May 2005.

75. A. Gandhi, al et., "Scalable Load and Store Processing in Latency Tolerant Processors," *Proc. 32th Int'l Symp. Computer Architecture*, 2005.

76. P. Racunas and Y. N. Patt, "Partitioned First Level Cache Design for Clustered Microarchitectures," *Proc. 17th Int'l Conf. on Supercomputing*, pp. 22-31, June, 2003.

77. B. S. Amrutur and M. A. Horowitz, "Speed and Power Scaling of SRAM's," *IEEE J. Solid-State Circuits*, Vol. 35, No. 2, pp. 175-185, February 2000.

78. J. D. Meindl et al., "The impact of stochastic dopant and interconnect distributions on gigascale integration," *IEEE Int. Solid-State Circuits., Dig. Tech. Papers*, pp. 232–233, 1997.

79. S. I. Association, "International Technology Roadmap for Semiconductors," 2002.

80. R. Ho, K. Mai, and M. Horowitz, "The Future of Wires," *Proceedings of the IEEE*, 89(4), pp. 490-504, 2001.

81. G. A. Saihalasz, "Performance trends in high-end processors," *Proc.IEEE*, vol. 83, Jan. 1995.

82. K. J. Schultz and P. G. Gulak, "Architectures for Large-capacity CAMs," *Integration, the VLSI Journal* 18, pp. 151-171, 1995.

83. L. Chisvin and J.R. Duckworth, "Content-addressable and associative memory: alternatives to the ubiquitous RAM," *IEEE Comput.* 22, pp. 51-64, 1989.

84. F. Shafai, al et., "Fully Parallel 30-MHz, 2.5-Mb CAM," *IEEE J. Solid-State Circuits*, Vol. 33, No. 11, pp. 1690-1696, November 1998.

85. S. K. Mathew al et., "A 4-GHz 300-mW 64-bit Integer Execution ALU Width Dual Supply Voltages in 90-nm CMOS," *IEEE J. Solid-State Circuits*, Vol. 40, No. 1, January 2005.

86. G. Hinton, al et., "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal* Q1, 2001.

87. E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A Low Latency Approach to High Bandwidth Instruction Fetching," *Proc. 29th Int'l Symp. On Microarchitecture*, pp. 24-34, December 1996.

88. E. Rotenberg, S. Bennett, and J. E. Smith, "A Trace Cache Microarchitecture and Evaluation," *IEEE Trans. on Computers, Special Issue on Cache Memory*, 48(2), pp. 111-120, February 1999.

89. Q. Jacobson, E. Rotenberg, and J. E. Smith, "Path-based Next Trace Prediction," *Proc. 30th Int'l Symp. On Microarchitecture*, pp. 14-23, December 1997.

90. Q. Jacobson, J.E. Smith, "Trace Preconstruction," *Proc. 27th Int'l Symp. On Computer Architecture*, pp. 37-46, June 2000.

91. L. Hammond et al., "The Stanford Hydra CMP," *IEEE MICRO, Magazine*, Vol. 20, No. 2, pp. 71–84, 2000.

92. Advanced Micro Devices, "AMD Demonstrates Dual Core Leadership," http://www.amd.com/, 2004.

93. Q. Jacobson, "UltraSPARC IV Processors," *Microprocessor Forum*, 2003.

94. D. S. Henry, B. C. Kuszmaul, G. H. Loh and R. Sami, "Circuits for Wide-Window Superscalar Processors," *Proc. 27th Int'l Symp. On Computer Architecture*, pp. 236-247, 2000.

95. V. Pless, "Introduction to the Theory of Error-Correcting Codes, 3rd Edition," Wiley, 1998.

96. J. Zhou and A. Mason, "A Two-level Hybrid Select Logic for Wide-Issue Superscalar Processors," to appear in *Int'l Symp. On Circuits and Systems (ISCAS)*, May 2006.

97. J. Zhou and A. Mason, "Increasing Design Space of the Instruction Queue with Tag Coding," *ACM Great Late Symp. On VLSI (GLSVLSI)*, April, 2005.