A Survey of Advanced Instruction Fetch Architectures for Dynamically Scheduled Superscalar Processors

Slade Maurer
DRS Tactical Systems (West), Inc.
1705 Jet Stream Drive
Colorado Springs, CO, USA
(719) 637-0880
smaurer@drs-ts.com

ABSTRACT

We survey fetch unit design components as a method of improving the IPC of dynamically scheduled superscalar microprocessors. We review RISC processor design techniques to improve the reader's understanding of modern fetch unit designs.

Keywords

Fetch unit microarchitecture, dynamically scheduled, out-of-order execution, branch prediction, multiple branch predictor, branch address cache, target buffer, trace cache, superscalar, microprocessor design, control transfer instruction.

1. INTRODUCTION

Semiconductor feature size continues to decrease permitting superscalar microprocessors to continue to increase the number of functional units available for execution. As the instruction issue width increases, more than one basic block must be issued per cycle to operate the microprocessor at peak performance. Computer architecture research has resulted in methods of fetching instructions beyond the first control transfer instruction to overcome the bottleneck created by the limitations of conventional predictors.

Superscalar microprocessors are becoming an industry standard in the high performance computing market. Examples of this category of microprocessors are superscalar RISC and CISC designs such as members of the PowerPC and Intel x86 processor families. Superscalar microprocessors are able to issue more than one instruction per cycle.

Many of the current microprocessor technologies statically schedule instructions in their runtime order. Dynamic scheduling advances this to aggressively exploit instruction level parallelism. We will discuss the benefits of dynamically scheduled microprocessors to foreshadow the need for more aggressive fetch unit architectures.

This paper presents background information necessary to understand current fetch unit designs employed in modern superscalar microprocessors. We discuss control transfer instructions to provide a general understanding of what they are and what they do. We cover modern instruction fetch architectures that predict the outcome and/or target of a single control transfer instructions. Then we describe newer fetch unit architectures being researched currently that will inevitably lead to new superscalar microprocessor designs.

The newer instruction fetching techniques are able to determine the direction and/or target of more than one control transfer instruction in a single cycle. These advanced techniques are used to issue multiple basic blocks per cycle.

We will discuss trace cache and multiple branch predictor techniques. These techniques are used to improve the average instruction per cycle performance of dynamically scheduled superscalar processors.

2. RISC PROCESSOR REVIEW

We will quickly review of the concept of pipelining and the 5-stage RISC pipeline before we delve further into more advanced topics.

Pipelining is a technique that allows multiple instructions to be overlapped in execution by taking advantage of parallelism inherent in the actions required to process instructions [HP03, pp. A-2]. This is done to permit higher processor clock speeds since each pipeline stage has a lower latency than the sum of all the stages.

The 5-stage RISC (Reduced Instruction Set Computer) pipeline is composed of the following stages in the listed order:

1. instruction fetch
2. instruction decode
3. execution
4. memory access
5. write back

The *instruction fetch* stage fetches an instruction from memory. *Instruction decode* interprets the instruction to determine its resource requirements and typically reads from the register file. The *execute* stage will perform computation required by the instruction and stores the result in a temporary output register. *Memory access* will access memory if required by the executed instruction. Finally, *write-back* writes results from execution or memory access into the register file [HP03].

The process of moving an instruction from the instruction decode to the execution stage is defined as *instruction issue* [HP03, A-33].

Caching of data and instructions is a technique that leverages the higher clock speed and lower latency of smaller memories to store information that is likely to be needed by the processor to improve performance.

When data is requested from the memory system a set of contiguous memory elements, a *cache line*, is read from memory and stored in the cache. Algorithms have been designed such that when data is written the memory system's coherency is preserved.

3. SUPERSCALAR PROCESSORS

Superscalar processors are a processor design category that issues multiple instructions per clock cycle [HP03, pp. 215]. This technique is used to increase the IPC performance of a processor.

IPC, which is an acronym for **I**nstructions **P**er **C**ycle, is defined as the number of instructions executed divided by the number of clock cycles required to execute those instructions [HP03, pp. 43]. IPC is commonly used to quantify the performance of a processor for a specific benchmark program.

Intuitively, if a processor can issue multiple instructions per cycle then it can have a higher IPC than a processor that only issues a single instruction per cycle.

A superscalar processor will have multiple *functional units* that perform the execution of instructions. The functional units may themselves be pipelined.

The decode pipeline stage of a superscalar processor is often divided into more than one stage and pipelined due to the fact that all instructions need to be checked for data hazards. We will discuss data hazards in more detail later on.

Superscalar processors often rely on the technique of *speculative execution*, which uses prediction of instructions to be executed to improve IPC performance. This is done to keep the functional units "well fed" with instructions. Prediction hardware and recovery techniques that are used in the case of incorrect execution are required. This hardware increases the complexity of the processor and number of pipeline stages.

4. DYNAMIC SCHEDULING

A *statically scheduled* pipeline will fetch an instruction, issue and execute it in program order. In a superscalar processor, this can result in *data hazards* that cause the processor to stall because continuing to process an instruction that has a dependence on another instruction would violate data integrity by changing the order of access to the operand involved in the dependence [HP03, pp.177].

Dynamic scheduling rearranges the instruction execution to reduce the occurrence of stalls caused by hazards while maintaining data flow and exception behavior [HP03, pp. 181]. Several techniques are used to reduce the stalls and they most often trade chip area for potential performance improvement by reducing stalls. Dynamic scheduling permits *out-of-order* execution of instructions.

There are four categories of data hazards. They are:
1. WAR – Write After Read
2. WAW – Write After Write
3. RAW – Read After Write
4. RAR – Read After Read

Data hazards are covered in many computer architecture texts such as [HP03] so we provide only a brief example. The following is an example of a WAR hazard in pseudo-assembly code:

```
ADD     F2,F4,F10
LD      F18,0(R1)
SUB     F6,F14,F12
MUL     F10,F16,F8
```

The example above has a WAR hazard (anti-dependence) on the use of F10 between the ADD and MUL instructions. If the MUL instruction were executed **before** the ADD then the updated F10 register would cause the ADD instruction to produce the wrong result. This would not possibly happen in a statically scheduled processor but it is feasible in a dynamically scheduled processor and must be accounted for.

Still using the same example, take a look at the SUB instruction. That instruction can be executed safely and out-of-order with the other instructions since it has no dependencies. If the LD instruction were to stall due to a cache miss then the SUB could still execute out-of-order. This is the benefit of dynamic scheduling.

Register renaming eliminates most hazards by renaming all destination registers of an instruction so that writing of registers can be accomplished out-of-order. This is performed in *reservation stations* that buffer the operands waiting to issue. Each functional unit has a reservation station that determines when an instruction can begin execution at that unit. This is in contrast to the centralized register file of the 5-stage RISC processor [HP03, pp. 186].

Load and store instructions are buffered so that they can be executed out-of-order. The buffering of these instructions is performed in the same manner as in the reservation stations [HP03, pp. 186].

5. CONTROL TRANSFER

Control transfer happens within the runtime instruction stream when the program counter value is not incremented sequentially. A *control transfer instruction* (CTI) may change the program counter to an arbitrary value. The *CTI target* is the instruction that the CTI changes the program counter to.

The *fetch unit* is the microarchitectural component that handles the instruction fetch stage(s) of the RISC pipeline. It is responsible for providing instructions to the instruction decode stage.

CTIs are problematic for fetch units since the instructions following a CTI may not be contiguous to those preceding it and the unknown direction of conditional branches. Many designs have been proposed to alleviate lost fetch opportunities due to these reasons.

There are a few reasons that CTIs are problematic for fetch units of dynamically scheduled superscalar processors, they are:

1. Instruction cache contains lines of contiguous instructions
2. Branch direction is resolved after fetch
3. CTI targets are resolved after fetch

A *basic block* is a sequence of instructions in run-time order that contains a single CTI at the end (there may be a delay slot following the CTI). The instructions in a basic block have sequential program counter values. The basic block has only one exit point and might have more than one entry point. As a side note, a *delay slot* is an instruction that immediately follows a branch and is executed before the branch target.

Our definition of a basic block is slightly looser than the accepted definition since we permit more than one entry point for the reason that it encompasses a larger set of fetch unit designs. The accepted definition of a basic block is as follows, "*basic block* – a straight-line code sequence with no branches in except to the entry and no branches out except at the exit" [HP03, pp. 173].

6. TARGET BUFFERS

The *Program Counter (PC)* is the address in memory of an instruction. *Target buffers* are data structures that hold CTI target PCs for future lookup based on the CTI's PC.

As a speculative execution technique, they improve performance by allowing the fetching of the next basic block following a CTI without waiting for the target to be resolved.

Target buffers are useful for branch and direct jump targets because the effective address of the target does not change.

Target buffers are less helpful in the case of indirect jumps since the effective target address is computed during execution. Most indirect jump instructions are due to function call returns in modern software since the use of "goto" is not very common.

7. BRANCH PREDICTION

Branch prediction uses historical data to predict the direction of a branch instruction. The *direction*, or *outcome*, is either taken or not-taken. A *taken* branch implies the program counter value was changed to an arbitrary value and a *not-taken* outcome means it was incremented sequentially.

Data representing the history of branch outcomes is stored in a *prediction table*. A *lookup algorithm* is used to find a direction prediction from the prediction table.

Dynamic branch prediction is an architectural feature designed to determine the direction of a branch before the direction can be resolved through the execution of the branch instruction. The simplest dynamic branch prediction schemes produce one direction prediction per cycle. The target of the predicted branch may not be determined until the decode stage, the earliest pipeline stage when a branch target can be resolved, unless a branch target buffer or a similar technique is used. The fetch unit uses the direction prediction to issue the next basic block of instructions once the branch target can be determined.

Single branch prediction is a dynamic branch prediction technique that uses a history table to determine the direction of a single branch in a single cycle.

The purpose of branch prediction is to reduce pipeline stalls that occur when instructions cannot be fetched due to the fact that the outcome of a branch has not yet been determined.

Branch prediction is more useful as the pipeline depth increases due to the increased latency until outcome resolution during execution. The dynamically scheduled superscalar processor's multistage issue pipeline is an example.

Predicting branch outcomes is important as the issue width of superscalar processors increases since more instructions are being fetched each cycle. If the fetch unit must wait for the outcome to be determined, it will frequently stall.

For further study, J. Smith covers many branch direction prediction schemes including the bimodal predictor [S81].

Lee et al. evaluate several branch direction predictor designs and introduce branch target buffers to reduce pipeline delay by determining the branch target in the same cycle as the prediction [LS84].

Early work in the area of branch prediction is done by McFarling et al. who compare hardware and software approaches including profiling [MH86].

Pan et al. describe combining global and local history into a dynamic branch predictor [PSR92].

Yeh et al. introduce local and global branch direction prediction schemes [YP92, YP93].

S. Mcfarling decomposes early single branch predictor architectures into subproblems and studies these in detail which produces the GSHARE design (Global History with Index Sharing) [M93].

The YAGS branch predictor is proposed by Eden et al. as a high performance single branch predictor design [EM98].

8. PREDICT MANY BRANCHES

A *multiple branch predictor (MBP)* is a microarchitectural design that simultaneously looks-up multiple branch direction predictions in a single cycle.

A *branch address cache* is used by the MBP to simultaneously determine the branch targets of each predicted branch. The MBP requires a multi-ported instruction cache that can issue more than one line per cycle to be able to fetch the multiple basic blocks referenced by the branch direction predictions and target addresses.

Realignment logic is required to format the noncontiguous basic blocks contained in the lines fetched from the instruction cache into a contiguous block of instructions.

The MBP effectively determines a set of basic blocks and then pulls the instructions from the instruction cache as multiple cache lines. The lines contain the basic blocks' instructions along with others that are extraneous. The realignment logic in the decode stage removes the extraneous instructions and realigns the basic blocks' instructions into one contiguous set of instructions. These are then issued.

T. Yeh et al. propose the multiple branch predictor and branch address cache [YMP93]. Yeh et al.'s multiple branch predictor uses shifted versions of the global history register to predict more than one branch direction per cycle.

A collapsing buffer design is introduced to shorten the latency of the decode stage's realignment logic by Conte et al. [CMMP95].

The tree-like subgraph that is similar to the multiple branch predictor with extensive use of local history [DF95] is studied by Dutta et al.

D. Koppelman analyses a multiple branch predictor design using a YAGS based predictor and a specialized branch address cache that includes direct and indirect jump prediction on a full timing based simulator [K02].

9. CACHING RUNTIME ORDER

The *trace cache (TC)* fetch unit design creates a special instruction cache internal to the fetch unit, the *trace cache*, which is used to hold instructions in their dynamic runtime order.

A *trace* is a snapshot of the dynamic instruction stream that is at most *n* instructions in length and is composed of instructions in their runtime order. The trace is at most *n* instructions long so that it can be cached in the trace cache that has a line length of *n* instructions.

The trace cache design has two conceptual levels of sequencing: instruction level and trace level. Instruction level sequencing is the initial state of the fetch unit and performs its work with the instruction as the atom of operation. Trace level sequencing is entered from instruction level sequencing and performs its work using the trace as the atom of operation.

The trace cache fetch unit design is motivated by the early MBP designs proposed in the literature.

Rotenberg et al. speak to the motivation of their proposed TC design in the section "Problems with other Fetch Unit Mechanisms":

> "Recall that the job of the fetch unit is to feed the dynamic instruction stream to the decoder. Unlike the trace cache approach, previous designs have only the conventional instruction cache, containing a static form of the program, to work with. Every cycle, instructions from non-contiguous locations must be fetched from the instruction cache and assembled into the predicted dynamic sequence. […] The trace cache approach avoids these problems by caching dynamic instruction sequences themselves, ready for the decoder" [RBS96, pp. 5].

In comparison to the MBP, the TC has the advantage that it does not require a multi-ported instruction cache or decode stage realignment logic. This complexity is traded for additional memory in the fetch unit, the trace cache itself, and redundant instruction storage.

Rotenberg et al. introduce an early trace cache design based on multiple branch prediction [RBS96]. This design is limited to a fixed number of basic blocks and traces are terminated by jump instructions or a maximum length.

A trace processor design was developed to explore a new architecture built around the trace concept [RJSS97].

The Next Trace Predictor (NTP) is introduced by Q. Jacobson et al. [JRS97]. The NTP uses a trace as the atom of prediction instead of the CTIs in the trace. The predicted path is navigated from trace to trace. This has been shown to be very accurate and does not rely on branch prediction hardware.

Rotenberg et al. study an improved trace cache design using the next trace predictor [RBS99]. Their design has no limit on the number of embedded basic blocks due to the use of the NTP.

10. CONCLUSIONS

Dynamically scheduled superscalar processors are able to use advanced microarchitectural techniques to issue multiple instructions in a clock cycle out-of-order. As the issue width of these processors increases they need to be able to fetch more than one basic block per cycle.

We have presented some techniques to fetch multiple basic blocks in a cycle. Research in this area continues to this day and more improvements are possible.

Future improvements could reduce the complexity and therefore chip area of fetch unit designs as well as reduce the total number of pipeline stages. There is room for improvement in the accuracy of predictions across multiple basic blocks or traces.

Understanding the benefits of dynamically scheduled superscalar processors is important to the programmer. The sophisticated hardware does not require as much sophistication in the compiler for optimization. It allows code compiled with optimizations for a specific pipeline to be ran efficiently on another improving its portability. Hand coded assembly or machine code is simplified since the programmer can focus more on the algorithm and less on the instruction order to minimize stalls.

A draw back of these processors is the inherent jitter in the execution of a program. Due to the memory hierarchy, dynamic scheduling and speculative execution there is a stochastic nature to a program's execution. If a benchmark is ran under a real-time operating system 100,000 times and a histogram of the execution cycles is produced, there will be a distribution of times. Not all 100,000 execution times will fall in the same bin. Benchmarking a processor and profiling this jitter is important for a real-time system designer with tight jitter constraints.

11. REFERENCES

[CMMP95] T. Conte, K. Menezes, P. Mills, and B. Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates", *22nd International Symposium on Computer Architecture*, June 1995, pp. 333-344.

[DF95] S. Dutta and M. Franklin, "Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors", *International Symposium on Microarchitectures*, December 1995, pp. 258 – 263.

[EM98] A. Eden and T. Mudge, "The YAGS Branch Prediction Scheme", *International Symposium on Microarchitecture*, December 1998, pp. 69 – 77.

[HP03] J. Hennessy, D. Patterson, "Computer Architecture: A Quantitative Approach", *Morgan Kaufmann Publishers*, 2003.

[JRS97] Q. Jacobson, E. Rotenberg and J. E. Smith, "Path-Based Next Trace Prediction," *Proc. 30th Int'l Symp. High Performance Microarchitecture*, December 1997, pp. 14 - 23.

[K02] D. Koppelman, "The Benefit of Multiple Branch Prediction on Dynamically Scheduled Systems," *Workshop on Duplicating, Deconstructing, and Debunking Held in conjunction with the 29th International Symposium on Computer Architecture*, May 2002, pp. 42 – 51.

[LS84] J. Lee and A Smith, "Branch Prediction Strategies and Branch Target Buffer Design", *Computer*, 17(1), January 1984.
[M93] S. McFarling, "Combining Branch Predictors," *WRL Technical Note TN-36*, June 1993.

[MH86] S. McFarling and J. Hennesy, "Reducing the Cost of Branches", *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986, pp. 396-403.

[PSR92] S. Pan, K. So and J. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation", *Proceedings of ASPLOV V*, October 1992, pp. 76 – 84.

[RBS96] E. Rotenberg, S. Bennett and J. E. Smith, "Trace Cache: A Low Latency Approach to High bandwidth Instruction Fetching," *Proceedings if the 30th International Symposium of High Performance Microarchitectures*, December 1997, pp. 138 – 148.

[RBS99] E. Rotenberg, S. Bennett and J. E. Smith, "A Trace Cache Microarchitecture and Evaluation," *IEEE Transaction on Computers*, vol. 48, no. 2, February 1999, pp. 111 - 120.

[RJSS97] E. Rotenberg, Q. Jacobson, Y. Sazeides and J. Smith, " Trace Processors", *Proceedings if the 30th International Symposium of High Performance Microarchitectures*, December, 1997.

[S81] J. Smith, "A Study of Branch Prediction Strategies", *Proceedings if the 8th International Symposium of Computer Architecture*, May 1981, pp. 135 – 138.

[YMP93] T. Yeh, D. Marr, Y. Patt, "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache", *in Proceedings of the International Conference on Supercomputing*, 1993, pp. 67 – 76.

[YP92] T. Yeh and Y. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction", *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, pp. 124 – 134.

[YP93] T. Yeh and Y. Patt, "A Comparison of Dynamic Branch Predictors That Use Two-Levels of Branch History", *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993, pp. 257 – 266.