Miss Path Speculative Scheduling For High Issue Rates

Sanjeev Banerjia   Sumedh W. Sathaye   Kishore N. Menezes
Thomas M. Conte

Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, North Carolina 27695-7911
(919)-515-7983
{sbanerj,knmeneze,swsathay,conte}@eos.ncsu.edu

Abstract

Many contemporary multiple issue processors employ out-of-order scheduling hardware in the processor pipeline. Such scheduling hardware can yield good performance without relying on compile-time scheduling. The hardware can also schedule around unexpected run-time occurrences such as cache misses. As issue widths increase, the complexity of such scheduling hardware increases considerably and can have an impact on the cycle time of the processor.

This paper presents the design of a multiple issue processor that uses an alternative approach called *miss path scheduling*. Scheduling hardware is removed from the processor pipeline altogether and placed on the path between the instruction cache and the next level of memory. Scheduling is performed at cache miss time, as instructions are received from memory. Scheduled blocks of instructions are issued to an aggressively clocked in-order execution core. Details of a hardware scheduler that can perform speculation are outlined and shown to be feasible. Performance results from trace-driven simulations are presented that highlight the effectiveness of the approach.

1 Introduction

Current multiple issue processors such as the Hewlett Packard PA-8000 and the Intel Pentium Pro employ out of order issue [1], [2]. One advantage of such an approach is that compiler scheduling is not required to achieve acceptable performance. Additionally, dynamic scheduling hardware can deal effectively with unanticipated run-time events (e.g., cache misses). Since the hardware is used on every instruction cache access, this approach can be termed *hit path scheduling*. There are also disadvantages to hit path scheduling. Scheduling is performed on a subset of instructions, the size of which is limited by the size of a central hardware window; the hardware does not have global knowledge of the instruction stream. The complexity of the hardware required for out of order issue can lead to an increase in the processor cycle time as issue widths increase. For example, the 500 MHz in-order-issue Alpha 21164 outperformed its contemporary, the out-of-order issue 200 MHz Intel Pentium Pro in July of 1996 [3]. What is desirable is the flexibility of an out-of-order issue design combined with the aggressive cycle time of an in-order-issue design without the compatibility limitations of a strong dependence on compile-time instruction scheduling.

An alternative to out-of-order issue is *miss path scheduling hardware* (the basic idea is depicted in Figure 1). A miss path design schedules instructions at instruction cache miss time. At cache hit time, instructions are issued to an aggressively-clocked in-order core. Scheduling hardware is inserted on the path between the instruction cache and the next level of memory. As a schedule of instructions is formed, it is placed into a specially designed instruction cache. Each line in this cache is composed of multiple instructions that form a unit of parallel issue and are guaranteed (by the scheduler) to be free of any constraints that could prevent parallel issue. Instructions within a cache line are aligned with the functional unit on which they will execute.

Miss path scheduling was first proposed by Melvin, Shebanow and Patt [4] and extended by Franklin and Smotherman [5]. Contemporary out-of-order designs speculate across multiple branches in order to find sufficient parallelism. Previous miss path schedulers did not address speculation. This paper proposes a miss path hardware scheduler that speculates across multiple branches. A hardware implementation of the operation scheduling algorithm [6] is developed.

This paper is organized as follows. Section 2 discusses previous work related to miss path scheduling. Section 3 introduces miss path scheduling through an example and discusses how speculation is performed. Section 4 details unpipelined and pipelined implementations of a miss path scheduler. Results are presented for a design that performs basic block only scheduling. Section 4.2 outlines how the design can schedule instructions speculatively

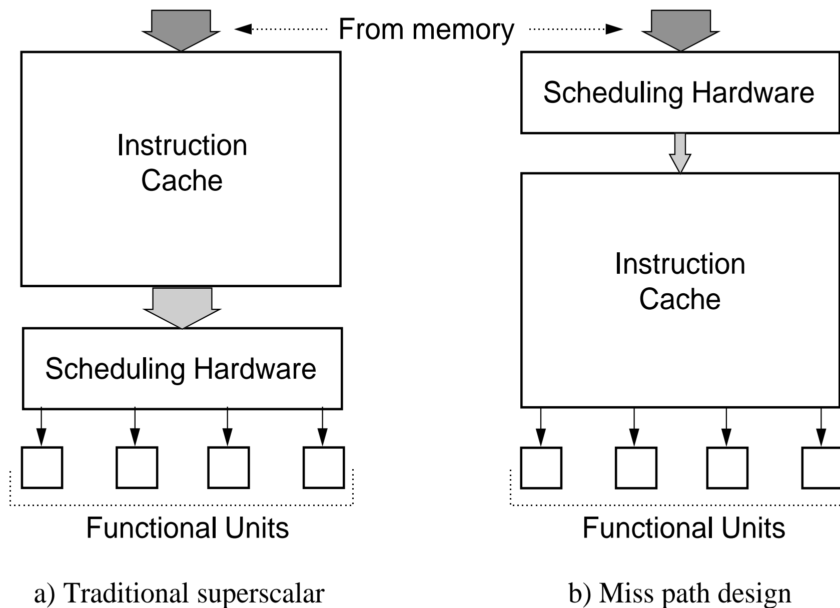

Figure 1: A traditional superscalar vs. a miss path design: The traditional superscalar has an instruction cache which is accessed on every cycle for instructions. These instructions are then examined by scheduling hardware which determines which instructions are safe to issue in the current cycle and performs dynamic scheduling. In the miss path design, instructions are fetched from a cache that holds wide instruction words. The wide words are composed of instructions that are guaranteed to be free of any dependencies or restrictions that can prevent parallel issue. Dependency checking or dynamic scheduling hardware is not required in the path between the cache and the execution pipeline (although simpler in-order dependency enforcement might be useful for cache miss handling).

across multiple branches. Results are presented for a speculative miss path design. Section 5 outlines possible variations on the proposed design and areas for future research. Section 6 presents the conclusions of the paper.

2 Background work

The foundation of the concept of miss path scheduling in a microprocessor is the fill unit[4]. The original fill unit proposal was geared for use in a dynamically (hit path) scheduled multiple issue processor. The fill unit forms what are essentially multiple issue wide words by analyzing instructions as they are received from a lower level of memory. Instructions are processed one at a time to determine into which wide word they can be placed. Filling completes when either the fill unit is full (a hardware limit) or a branch instruction is encountered. When filling stops, the contents of the fill unit are copied into a decoded instruction cache. Normal instruction execution *i.e.*, without exceptions, is done by accessing

lines in the decoded cache. Dynamic scheduling hardware is used between the decoded cache and the execution pipeline. A more recent study examined the use of a fill unit for constructing dependence-free, VLIW-like instructions to be placed into a shadow cache [5]. The idea was to use a fill unit to build wide instruction words that contained no inter-instruction constraints on multiple issue, similar to a VLIW. Speculation across multiple branches was not performed, but the design was able to fetch both paths of a branch, similar to a tree instruction [7]. The design assumed that the fill unit executed in parallel with the execution pipeline (the original fill unit proposal does not explicitly comment on this aspect). A followup to the shadow cache idea was a proposal to use the fill unit for use in decoding a CISC instruction set [8].

Another miss path scheduling design was the expanded parallel instruction cache (EPIC). EPIC performs limited dynamic scheduling at cache miss to ease decode requirement at run-time and form VLIW-like instructions from a RISC instruction stream [9]. Limited speculation is performed: instructions that follow a branch are allowed to issue in the same cycle as the branch but are not allowed to begin execution before the branch. A novel feature of an EPIC machine is that a wide word can hold instructions that can be issued in multiple cycles. This allows instructions slated to issue in different cycles to co-exist in the same cache line. A different approach to increasing the performance of superscalars was taken in the design of the trace cache [10]. The design performs alignment and merging of instruction runs across multiple branches and places the resulting instruction sequence – the trace – as a line into a trace cache. Traces are formed based on the current fetch address and branch predictions returned from a multiple predictor. A trace consists essentially of a sequence of basic blocks. The trace cache is accessed on each clock cycle using the current fetch address and the multiple predictions given by a hardware branch predictor. An instruction cache is accessed in parallel with the trace cache for the situation in which the trace cache misses. Dynamic scheduling hardware is used between the caches and the execution pipeline.

3 A simple scheduling algorithm

One algorithm that can be implemented by a miss path scheduler is operation scheduling [6], [11]. This particular algorithm is well-suited for hardware implementation because scheduling is performed by processing an instruction stream sequentially and does not require first constructing a dependency graph. The algorithm is referred to as the miss path scheduling algorithm for the remainder of this paper The following section introduces the technique through an example and comments on the hardware structures required (an informal outline is presented in the appendix).

3.1 An example of miss path scheduling

A portion of assembly code and a machine for which it is to be scheduled are shown in Figure 2. The machine is a three issue processor that has two pipelined integer ALUs and a pipelined load/store unit. The ALUs have a latency of three cycles for multiply and divide operations and one cycle for all other operations. The load/store unit has a two cycle latency for loads and a one cycle latency for stores. An empty reservation table and a register definition and use table (**def-use table**) to be used while scheduling are also shown. The def-use table holds def-time and last-use entries for every register in the architecture. The def-time entry for a register indicates the most recent cycle in which the register is written (defined). The last-use entry for a register indicates the most recent cycle in which the register is read (used) as a source operand. Assume that each entry in the def-use table also has a busy bit associated with the register id.

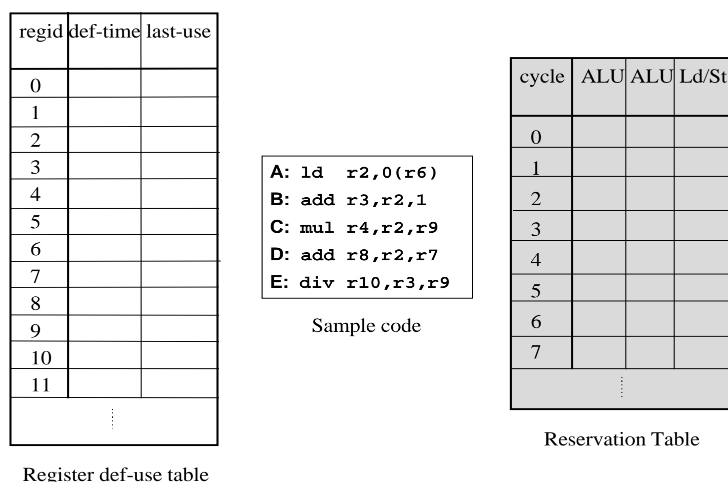

Figure 2: Sample code and an empty schedule

Consider instruction A. A reads `r6` and writes `r2`. The source operands of A are scanned to determine when they will be available for use. The def-use table is checked to see in what prior cycle `r6` is defined. As the def entry is empty (assume that the busy bit is not set), `r6` is available, so A can potentially issue in cycle 0. Next, the destination operand `r2` is reserved by setting its busy bit. The reservation table is checked to see if a load/store unit is available in cycle 0. The unit is available in cycle 0, so A can issue in cycle 0. Since A has a two cycle latency, `r2` will be available for use in cycle 2, after A has written (defined) it. The def-use table is updated to reflect that A uses `r6` in cycle 0 and defines `r2` in cycle 2. The busy bit for `r2` is cleared to show that `r2`'s def-time has been updated. The reservation table is updated to reflect that A begins execution on the load/store unit in cycle 0. Now

consider instruction B. B uses `r2` as a source operand, and the def-use table indicates that `r2` is produced in cycle 2. The busy bit for operand `r3` is set in the def-use table. The reservation table is checked starting at cycle 2 for the the first empty cycle in which an ALU is available (which happens to be cycle 2). B is a single cycle instruction, and so writes its result into `r3` in cycle 3. The def-use table is updated with a use time of 2 for `r2` and a def-time of 3 for `r3`. The reservation table is modified to indicate that B will use an ALU in cycle 2. The state of the def-use table and the reservation table after scheduling instructions A and B is shown in Figure 3. The state after scheduling all of the instructions is shown in Figure 4.

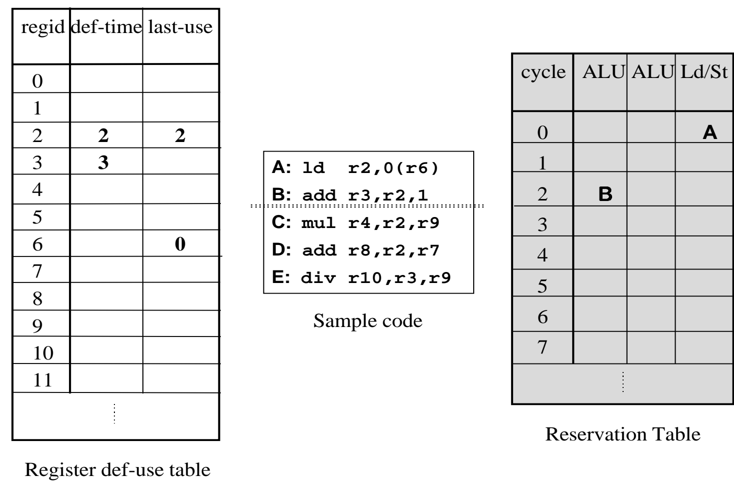

Figure 3: State after scheduling instructions A and B.

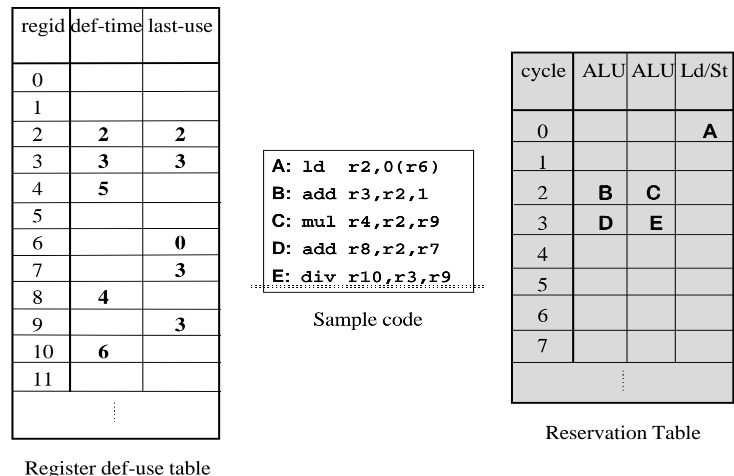

Figure 4: State after scheduling all instructions.

The preceding example illustrates that the steps involved in miss path scheduling an instruction are 1) checking the availability of source operands in the def-use table, 2) reserving destination operands in the def-use table to indicate that they will be defined, and 3) searching the reservation table for an available slot. The algorithm requires only one pass through a set of instructions and does not need to re-process an instruction after scheduling it. The data structures required for scheduling can be easily implemented in hardware. The register def-use table can be constructed as additional fields within a register file that store def-time and last-use entries. The size of the entries depends on the maximum length allowed for a schedule, but in general this will be much smaller than an integer or floating point value. The reservation table can be a two-dimensional bit matrix that is indexed with the value of the first cycle in which an instruction can be scheduled. A priority encoder can be used for a quick search of the reservation table, returning the first available issue cycle when given an initial potential issue cycle. Section 4 discusses the additional requirements for scheduling.

The example in Figures 3-4 considers only flow dependencies. Anti- and output dependencies can be handled also. To honor an anti-dependency while scheduling instruction X, the def-use table is checked to get the last use time of X's destination operands (call this `LastDstUse`). The scheduling time for X is set so that X does not retire its results before `LastDstUse`. To honor an output dependency, the def-use table is checked to get the def times of X's destinations operands (call this `LastDstDef`). X's scheduling time is set so that it does not retire its results before `LastDstDef`. Note that the restrictions on instruction issue caused by anti- and output dependencies can be relieved by compiler techniques or hardware support.

3.2 Speculative scheduling at miss time

The previous example illustrated how miss path scheduling works on straight line code (code without branches). Scheduling in the presence of branches is as follows: as a sequence of non-branch instructions is scheduled, a `LatestScheduleTime` can be maintained to track the latest cycle in which an instruction is scheduled. When a branch is encountered, its scheduling time is constrained to be no earlier than `LatestScheduleTime`. This prevents the branch instruction from executing prior to instructions that precede it.

Miss path scheduling can speculate an instruction across an arbitrary number of branches. The technique performs greedy scheduling, attempting to schedule an instruction at the earliest time that its source operands are available, while honoring anti- and output dependencies. When a conditional branch is encountered, the hardware makes a prediction as to the direction of the branch and begins scheduling instructions from the predicted path. Instructions from the predicted path are scheduled and may be placed higher (earlier) in the schedule

than the preceding branch. They may be placed ahead of *several* preceding branches. Such a strategy is analogous to a compile-time instruction scheduling algorithm known as superblock scheduling [12]. When such speculation is performed, speculated instructions must be prevented from retiring their results when their dominating branches are mispredicted. Hardware support identical to that used by speculative out-of-order issue designs can be used to accomplish this [13], [14], [15].

4 Details of a miss path scheduler

The previous section introduced some of the hardware structures required for miss path scheduling. The data stored in the def-use table and the reservation table are used to make scheduling decisions. Additional logic is needed to interpret the data and perform scheduling. The logic must execute the steps of the miss path scheduling algorithm, as discussed in Section 3.1. They are listed again below to make the requirements of the scheduling logic explicit.

Miss Path Scheduling:

1. Check the availability of the instruction's source operands by reading their def-times and busy bits from the def-use table. This is used to compute an initial value for the earliest cycle in which the instruction can execute. Call this value `ScheduleTime`. The register ids required for this (and all following) steps can be determined using logic identical to that used for instruction decode.
2. Reserve the instruction's destination operands by setting their busy bits in the def-use table.
3. (a) Read the def-use table to get the def-time for the instruction's destination operands and the last-use time for the instruction's destination operands. (b) Use these values to adjust `ScheduleTime` so that output and anti-dependencies are honored.
4. Determine the resources that the instruction requires to execute. This can be done using the instruction's opcode[1].
5. Search the reservation table starting at cycle `ScheduleTime` for the first available cycle in which all of the instruction's required resources are available. The value of that cycle is used to update the `ScheduleTime` value.
6. Set entries in the reservation table to reserve the resources needed by the instruction.
7. Set the def-times for the instruction's destination operands by writing to the def-use table. Clear the destination operands' busy bits.

[1]We assume a machine in which all of the resources required by an instruction are supplied by the functional unit on which the instruction executes. Because of this, the reservation table need model instruction issue slots only.

8. For each of the operation's source operands, if the last-use time in the def-use table is less than `ScheduleTime`, set the entry to the `ScheduleTime`. Nominally, this step requires reads and writes from the def-use table but in fact, only writes are required. The last-use times can be read in either Step 2 or 3 and stored for use in this step.

9. Place the instruction into the instruction cache at a location determined by the address of the first instruction in the schedule and the value of `ScheduleTime`.

The actual number of sequential steps within an implementation can be less than the number of steps listed above, as many of the steps can be performed in parallel by hardware. Step 1 is a write to the register file and Steps 2-3 are reads from the register file. These three steps can be performed in parallel in one clock cycle (Step 3 requires logic to adjust `ScheduleTime` and therefore could take an additional cycle). Step 4 can be performed in parallel with Steps 1-3 as it uses the same decode logic. Steps 7 and 8 are writes to the register file and Step 9 is a write to the instruction cache and can be performed in parallel in one clock cycle, after `ScheduleTime` has been set. Step 6 can be merged in with Steps 7-9 also. By combining steps as discussed, the original nine steps listed above can be reduced to three (3) steps. These three steps can be performed in four (4) clock cycles: two cycles for Steps 1-4, one cycle for Step 5, and one cycle for Step 6-9. Therefore, an unpipelined miss path scheduler requires four cycles to schedule an instruction.

The number of ports required on the def-use table is moderate. Accessing a register in the def-use table accesses all of the information regarding that register: the busy bit, last-use time, and def-time. Step 1 requires one write port to write the busy bit. Steps 2 and 3 both perform reads for all of an instruction's register operands. For most current ISAs, this is a maximum of three operands and hence requires three read ports. Therefore, Steps 1-3 require a total of one write port and three read ports. Steps 7 and 8 require three write ports for parallel execution. The maximal overall port requirements on the register file are three write ports and three read ports.

Scheduling of individual instructions has been outlined but overall program execution has not. The following outline enumerates the steps involved in program execution on a miss path design. Figure 5 shows a high-level hardware organization of a miss-path design.

Program Execution Flow:

1. Use the PC to access the instruction cache, which holds scheduled blocks of instructions. This is identical to instruction fetch in a traditional superscalar processor.

2. If the cache hits, retrieve an issue width of instructions and send it to the execution pipeline.

3. If the cache misses, initiate cache miss processing. Miss processing consists of fetching instructions from memory and performing miss path scheduling.

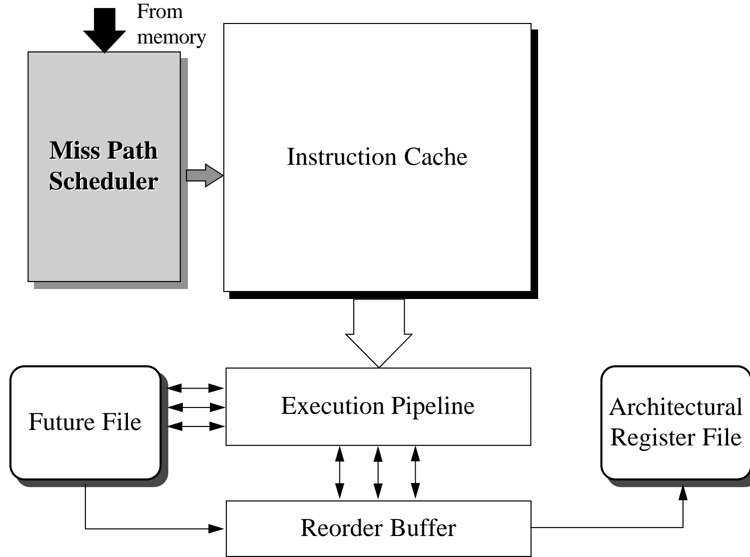

Figure 5: Hardware organization of a miss path scheduler. The def-use table and reservation table are part of the Miss Path Scheduler in this diagram.

4. After cache miss processing completes, use the PC to re-access the cache for instruction fetch. Resume execution in the execution pipeline.

An arbitrarily long instruction stream can be scheduled but it is more practical to use a *halt condition* to terminate miss processing/miss path scheduling. For example, scheduling could terminate when a conditional branch is encountered. (This enforces a basic block only scheduling policy with no speculation.) Because scheduling is not performed across the entire program by one invocation of the miss path scheduler (one cache miss). Inter-block dependencies are not honored by the list scheduler. Such dependencies can be handled within the execution core by using the busy bits as a scoreboard [16], [17].

The performance of basic block only miss path scheduling with an infinite sized instruction cache was measured and is presented in Figure 8 (discussed further below). Trace driven simulations were used to obtain all of the results presented in this paper. All eight of the integer programs from the SPEC95 suite were used as the benchmark set (reference inputs were used). The programs were compiled for execution using a target machine model of the Hewlett Packard PA-7100 processor. The compiled code was passed through a cycle-by-cycle simulator that modeled the behavior of the hardware scheduler, the instruction cache, and the execution pipeline. The execution pipeline was assumed to have eight pipelined functional units (the exact configuration and latencies are listed in Table 1). The machine supports execution of multiple branches in parallel on any of the four ALUs. A 32 bits/cycle bandwidth with a five cycle latency was assumed between the instruction cache and next level

of memory. A perfect L1 data cache and unified L2 cache was assumed. Each benchmark was run for 20,000,000 instructions to bypass initialization code and additional 500,000 instructions for cache warmup. Statistics were then gathered by running the program for an additional 200,000,000 instructions.

Table 1: Machine configuration: functional units and latencies

FUNCTIONAL UNIT	QUANTITY	LATENCY
Integer ALU/Branch	4	1
Memory units (Ld/St)	2	2/1
FP ALUs (Add/Mpy/Div)	2	1/3/9

4.1 Pipelining the scheduler

There is a sequential nature to most of the steps in miss path scheduling, which allows a natural mapping of the process to a pipelined implementation. Pipelining can reduce the average scheduling time per instruction and with little hardware overhead.

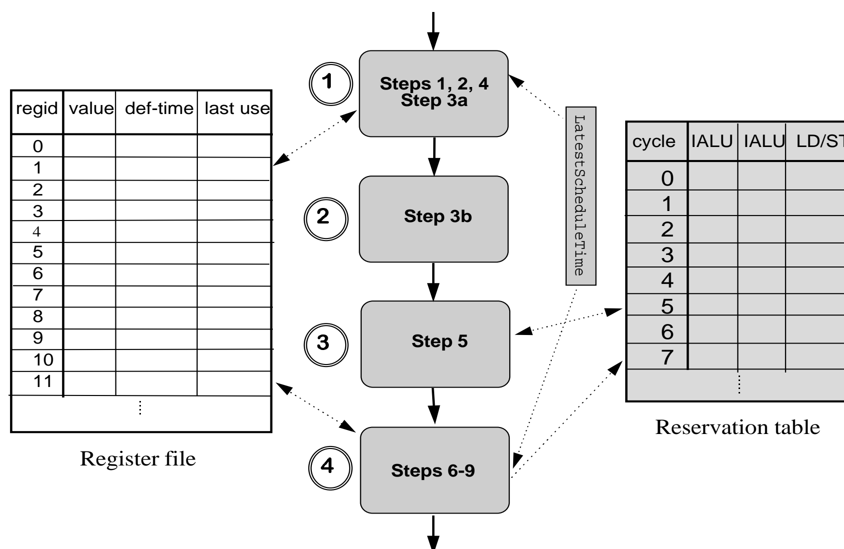

Figure 6: Pipeline stages for miss path scheduling.

Figure 6 illustrates a pipelined organization for a miss path scheduler. The pipeline mimics an execution pipeline in its ability to interlock and stall. Figure 7 depicts an example of this. Instructions A and B from the sample code in Figure 2 are shown in stages 1 and 2, respectively. A RAW hazard on `r2` exists between the two instructions. Because A has yet to clear the busy bit for `r2` when B is in stage 1, B cannot obtain an initial value for

`ScheduleTime` value. B stalls until A clears `r2`'s busy bit in stage 4. After A exits stage 4, B can proceed to stage 2.

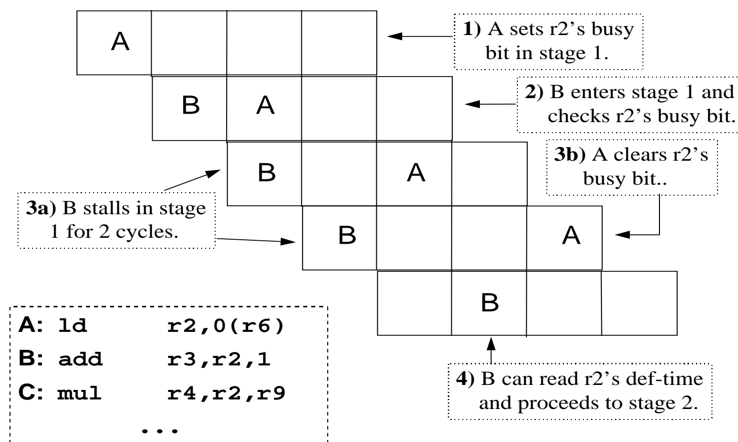

Figure 7: Stalling due to RAW hazard in the pipelined implementation.

The performance of pipelined and unpipelined miss path schedulers was measured for basic block miss path scheduling with an infinite instruction cache. The results are presented in Table 2. The pipelined implementation required on average 30% fewer clock cycles to form a schedule than the unpipelined implementation.

Table 2: Performance of piplined vs. unpipelined scheduling: The average number of clock cycles to form a schedule for basic block scheduling is shown.

Benchmark	Pipelined	Unpipelined
099.go	67.02	96.75
124.m88ksim	34.55	57.40
126.gcc	46.98	72.88
129.compress	29	55
130.li	23.99	42.51
132.ijpeg	121.96	218.39
134.perl	27.64	42.88
147.vortex	68.24	100.24

4.2 Speculative miss path scheduling

Extracting larger amounts of ILP from most general purpose programs requires scheduling beyond basic blocks. Miss path scheduling can speculate instructions above branches based purely on their data dependencies, as explained in Section 3.2. If speculation is used, a

mechanism is required to prevent incorrectly speculated instructions from retiring their results. A method that is well-suited for a speculative miss-path scheduler is a reorder buffer with a future file to supplement the architectural register file [13]. Slots are allocated in the reorder buffer in original program order (this is preserved by the scheduler and stored with the individual instructions in the cache).

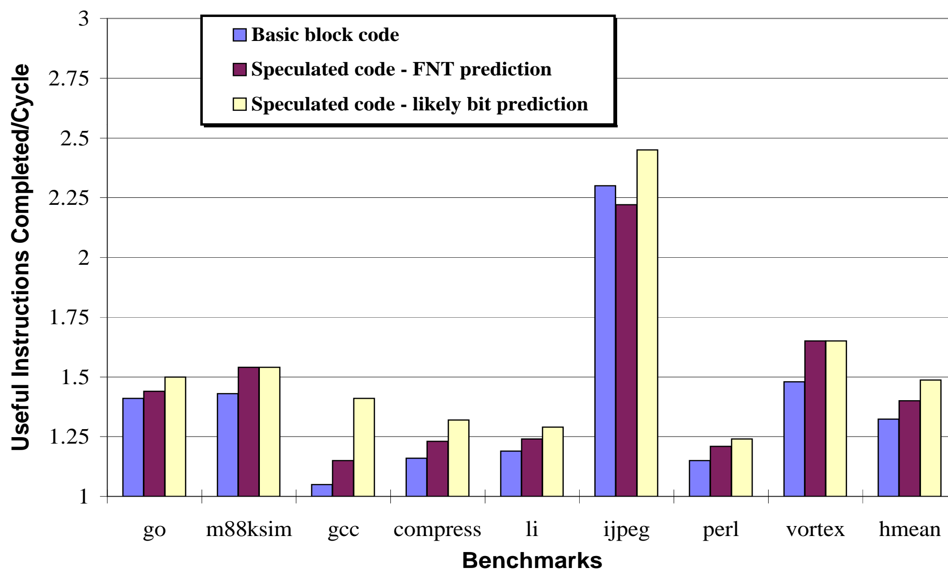

Figure 8: Performance of basic block and speculated schedules. Schedules formed using speculation used a likely-bit branch predictor. The metric used is useful instructions completed per cycle (IPC). Each individual bar represents the IPC for the indicated program and scheduling scheme. Harmonic means for the various schemes are shown in the rightmost set of bars.

The central issue in speculating instructions is choosing *which* instructions to speculate, a decision that relies on predicting which path a branch will take. Branch prediction has been a heavily researched topic [18], [19], [20], [21] For architectures that support a prediction bit in the instruction encoding [22], [23], the scheduler can use this bit to make a prediction. The bit setting may be based on profile information or a simple heuristic such as backward taken, forward not taken (BTFNT). In the absence of an ISA-level prediction bit, a simple heuristic such as FT can be directly implemented in the hardware (a BT approach is not used in order to prevent loop unrolling during scheduling). (Dynamic prediction hardware can also be used, but is left for future work).

When a branch is predicted taken by the scheduler, the scheduler issues memory requests using the branch target address to change the path along which instructions are fetched. The cost (in clock cycles) of the memory latency is re-incurred. To change the fetch path dynamically, the scheduler must first determine the branch target address. The target of

a PC-relative branch is determined in stage 1 of the schedule pipeline, after the immediate field has been decoded (if a static heuristic such as FT is used, the sign of the immediate field is used for the prediction). Indirect branches use a register source operand to determine the target address and are more difficult to process. For this reason, the current implementation of the scheduler stops scheduling which it encounters a non-PC-relative branch.

The performance of a speculative miss path scheduler was measured using an infinite cache and several branch prediction strategies. The scheduler speculated across an arbitrary number of branches, with a halt condition being a backward branch or a branch through a register. The branch prediction strategies used were a FT heuristic and an ISA-level prediction bit that was set via profiling[2]. The results are presented in Figure 8. In general, speculation yielded better performance than basic block scheduling. The likely-bit predictor proved to be more effective than the FT heuristic. Additional improvements could possibly be gained if the programs were optimized for ILP [24] and is a topic of future research.

4.3 Instruction cache support

As a schedule is constructed, the scheduled instructions are placed into an instruction cache (Step 9 of the miss path scheduling steps outlined in Section 4). The uncompressed and compressed cache designs similar to those discussed in [25], can be adapted for use with a miss path scheduler. The *uncompressed cache* places each wide instruction word into a line in the cache. Instructions are placed into cache so that they are aligned to their functional units, as in the scheduler's reservation table. The cache is addressed using the address of only the *first* instruction in the trace. The cache line into which an instruction X is placed is determined by the starting index (in the cache) of the trace and the cycle in which X is scheduled. One-word sized sub-blocks are used to allow placement of instructions into arbitrary positions within a cache line. Nops can be present within a cache line (these nops are called *horizontal nops*). Note that the reservation table used for scheduling indicates that there are nops present in a schedule (i.e., issue slots in which no instructions are scheduled). In the Figure 4, if no instructions are eventually placed in cycle 1, the empty cycle becomes a *vertical nop*. Also in Figure 4, if no instructions are placed into cycle 2 with instructions B and C, the issue slot for the Ld/St unit holds a *horizontal nop*. When used with a miss path scheduler, an uncompressed cache holds both vertical and horizontal nops. Unlike the uncompressed cache [25], a pause field cannot be used to eliminate vertical nops because their presence is not known until the entire schedule is built, due to the greedy nature of the miss path scheduling algorithm. This illustrates that a disadvantage of the uncompressed design is its potentially low space utilization.

[2]Training inputs were used for the profiling runs.

During cache access, the first line of a schedule is addressed using a PC value. The remaining lines of the schedule are addressed using a **pseudo-NextPC**. PC values are not used to address the cache once a schedule is found in the cache. Accesses to the successive lines of the schedule are done using the value of the previous cache index plus one. A NextPC value is not generated and used from the execution core (hence, the index+1 value is referred to as pseudo-NextPC). When a branch misprediction occurs within a trace, the correct address where control should flow (which is generated by the branch unit) is used to address the cache, overriding the pseudo-NextPC.

Unlike the uncompressed cache, a compressed cache design does not contain either vertical or horizontal nops [25]. Each wide word is stored with horizontal nops compressed out. Vertical nops are represented using a pause field that indicates to the instruction fetch logic to stall fetching for the indicated number of clock cycles. A compressed organization does not allow arbitrary placement of instructions directly into the cache at schedule time. When using a compressed cache, a *schedule buffer* is used to hold instructions as they are scheduled. When the scheduler has finished with the current instruction stream (a halt condition is encountered), instruction fetch resumes by fetching wide words directly from the buffer. As the wide words are sent to the execution core, they are also processed by compression logic so that they can be placed into the cache. When the schedule from the buffer is completed, it supplies an address to the instruction fetch logic, indicating to where control flows. The address is used to access the cache. If a cache hit occurs, the requested schedule resides in the cache. Because the wide words in the cache are stored in a compressed fashion and individual instructions are not aligned with their functional units, routing logic is used between the cache and the execution core to direct instructions to functional units [25]. This logic adds an extra stage to the pipeline between the cache and the execution core and increases the branch misprediction by one cycle.

Simulations were run to gauge the performance of the uncompressed and compressed cache designs with a speculative miss path scheduler. A 32KB, 4-way set associative organization was used for both designs. Results are presented in Figure 9. The compressed design yielded significantly better performance than the uncompressed design across practically all benchmarks. The results also indicate that certain benchmarks are more cache sensitive than others. Several benchmarks fit almost entirely within the cache (124.m88ksim, 129.compress, 132.ijpeg) while others performed very poorly (099.go, 126.gcc, 147.vortex). The characteristics of the individual programs give insight as to reasons for their cache behavior. Table 3 shows the number of unique traces each program produces and the average length of the schedules. (The measurements were made on the 200,000,000 instruction traced portion of the simulations and did not include statistics for the cache warmup period.) The programs

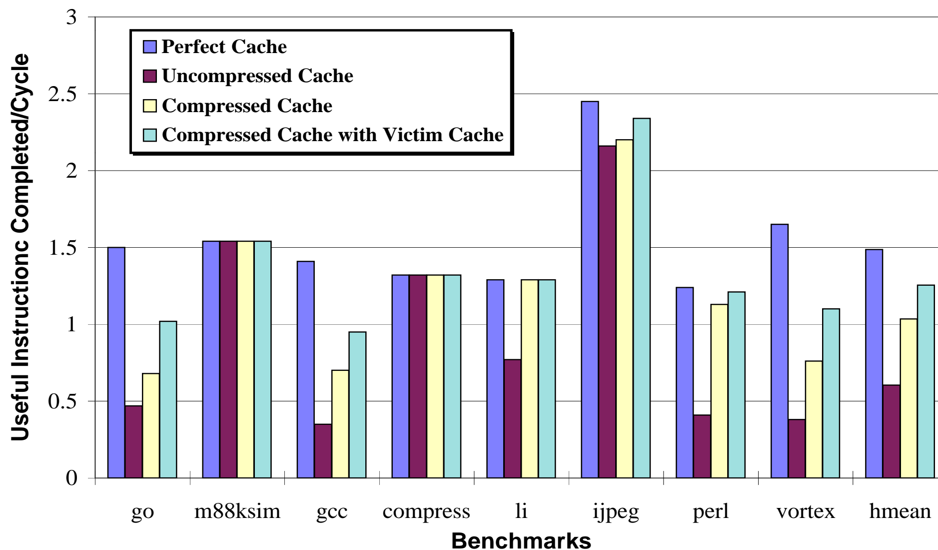

Figure 9: Performance of the uncompressed and compressed cache designs with speculated schedules. Speculation across three (3) branches was allowed. A likely-bit branch predictor was used. The metric used is useful instructions completed per cycle (IPC). Each individual bar represents the IPC for the indicated program. Harmonic means are shown in the rightmost set of bars.

that perform the worst with the cache also form the most traces. This highlights one of the differences between a schedule's footprint in a cache versus cache behavior in a superscalar processor. A conflict miss in a cache for a superscalar processor causes the replacement of one cache line. A conflict miss in an uncompressed or compressed cache can cause the replacement of *multiple* lines in the cache, depending on the size of the schedule that will be formed.

A victim cache can be used to mitigate the effects of conflict misses [26]. Simulations were run for configuration of a compressed cache backed by a 1KB victim cache. The results are presented in Figure 9. Performance improved significantly for the cache sensitive benchmarks. The victim cache effectively reduces the overhead of scheduling for benchmarks that form a large number of unique schedules.

5 Extensions to miss path scheduling

There are several variations on the miss path scheduler presented in Section 3- 4 that can be used to potentially improve performance. The pipelined scheduler honors all anti- and output dependencies when scheduling by adjusting the cycle when an instruction can first issue. A simple hardware register renaming scheme can be used to increase scheduling flexibility in

Table 3: Characteristics of SPECint95 programs for miss path scheduling. Figures shown are for speculative miss path scheduling across a maximum of three (3) branches using a likely-bit predictor. The schedule length figures are with respect to the height of a schedule in an uncompressed cache.

Benchmark	# unique schedules formed
099.go	1,057,300
124.m88ksim	104
126.gcc	2,903,170
129.compress	1
130.li	192,734
132.ijpeg	62,754
134.perl	15
147.vortex	949,098

the presence of such dependencies. Non-architected registers can be used to eliminate anti- and output dependencies. The destination of the instruction is renamed, and the entry for the original destination register in the def-use table is marked with the id of the renamed register. Entries in the reorder buffer maintain the original destination register id, so that at the result is written to an architected register at writeback. Miss path scheduling can be used for CISC architectures as well as RISC architectures. As is done in contemporary multiple issue CISC processors [27], a CISC instruction can be decomposed into multiple RISC-like instructions (micro-ops) [1]. These instructions can then be individually scheduled. Care must be taken so that the newly created micro-ops are ordered such that the dependencies between them are honored. Exception handling is achieved by the reorder buffer and is done identically for both RISC and CISC architectures.

Dynamic branch prediction can be used to increase the effectiveness of speculation. As shown in Figure 8, speculation using a FT heuristic or a like bit predictor performs better than basic block-only scheduling. A dynamic predictor could yield even greater performance. The current miss path design speculates instructions across branches on a chosen path. Another option is to speculate across multiple paths, as done in a tree instruction [7]. The shadow cache proposal explored placing both paths of a branch into a tree-like instruction but performed no speculation [5]. A miss path scheduler could choose to fetch from both paths when an unpredictable branch is encountered. Instructions from *both* paths could be speculated freely above their dominating branches and interspersed within a cache line. Such a technique is essentially a dynamic, purely hardware implementation of performing "on-the-fly" predication, as is done at compile-time in hyperblock scheduling [28]. Another variation is to speculate along one path when branches can be accurately predicted and use

dynamic predication to perform speculation along both paths when an unpredictable branch is encountered [29].

6 Conclusion

This paper has presented and developed an alternative to out-of-order issue that relegates the complexity of speculative scheduling to the Icache miss path. The advantage of this approach is the potential for aggressive cycle time for a simplified in-order processor core.

A specific implementation of a miss-time scheduler was presented in depth, including a method for pipelining this unit. The scheduling hardware is based on operation scheduling, a compiler algorithm. Three classes of instruction cache designs were also explored: the uncompressed, compressed and compressed/victim caches. Of these, the compressed cache combined with a victim cache was the best at hiding the penalty for scheduling at miss time. Speculation was performed via a likely-bit predictor, although there is opportunity to extend this work to more accurate predictors (as outlined above). Overall, the scheduler approached what would be expected from a superscalar processor employing an 80-85% accurate branch predictor, in spite of the non-traditional method for extracting parallelism.

The results suggest that miss path speculative scheduling provides a viable alternative to traditional out-of-order superscalars, especially when the likely reduction in cycle time is taken into account. This latter effect is difficult to model in this study, but evidence for this advantage is available in comercial designs (e.g., the 200MHz out-of-order PentiumPro vs. the same generation 500 MHz DEC Alpha 21164 [3]). The suggested enhancements to miss path speculative scheduling (see previous section) are reserved for future work.

References

[1] D. B. Papworth, "Tuning the Pentium Pro microarchitecture," *IEEE Micro*, vol. 16, pp. 8–15, Apr. 1996.

[2] L. Gwennap, "PA-8000 combines complexity and speed," *Microprocessor Report*, Nov. 1994.

[3] L. Gwennap, "Digital's 21164 reaches 500 mhz," *Microprocessor Report*, vol. 10, July 1996.

[4] S. Melvin, M. Shebanow, and Y. Patt, "Hardware support for large atomic units in dynamically scheduled machines," in *Proc. 21th Ann. International Symposium on Microarchitecture*, (San Diego, CA), pp. 60–66, Dec. 1988.

[5] M. Franklin and M. Smotherman, "A fill-unit approach to multiple instruction issue," in *Proc. 27th Ann. International Symposium on Microarchitecture*, (San Jose, CA), pp. 162–171, Dec. 1994.

[6] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donell, and J. C. Ruttenberg, "The Multiflow Trace scheduling compiler," *J. Supercomputing*, vol. 7, pp. 51–142, Jan. 1993.

[7] K. Ebcioğlu, "Some design ideas for a VLIW architecture for sequential-natured software," in *Proceedings of the IFIP Working Group 10.3 Working Conference on Parallel Processing*, (Pisa, Italy), pp. 3–21, North Holland, 1988. (published as *Parallel Processing*, M. Cosnard, et al., (eds).).

[8] M. Smotherman and M. Franklin, "Improving CISC Instruction Decoding Performance Using a Fill Unit," in *Proc. 28th Ann. International Symposium on Microarchitecture*, (Ann Arbor, MI), pp. 313–323, Dec. 1995.

[9] J. D. Johnson, "Expansion caches for superscslar processors," Tech. Rep. CSL-TR-94-630, Computer Systems Laboratory, Stanford University, Palo Alto, CA, June 1994.

[10] E. Rotenberg, S. Bennet, and J. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," Technical report, Department of Electrical and Computer Engineering, University of Wisconsin, Madison, WI, Apr. 1996.

[11] J. R. Ellis, *Bulldog: A compiler for VLIW architectures.* Cambridge, MA: The MIT Press, 1986.

[12] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective structure for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, Jan. 1993.

[13] J. E. Smith and A. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *Proc. 12th Ann. International Symposium Computer Architecture*, (Boston, MA), June 1985.

[14] M. D. Smith, M. A. Horowitz, and M. S. Lam, "Efficient superscalar performance through boosting," in *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, (Boston, Massachusetts), pp. 248–259, Oct. 1992.

[15] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling: A model for compiler-controlled speculative execution," *ACM Trans. Comput. Sys.*, vol. 11, pp. 376–408, Nov. 1993.

[16] J. E. Thornton, *Design of a Computer– The Control Data 6600.* Glenview, IL: Scott, Foresman, and Co., 1970.

[17] S. Weiss and J. E. Smith, "Instruction issue logic for pipelined supercomputers," *IEEE Trans. Comput.*, vol. C-33, pp. 1013–1022, Nov. 1984.

[18] J. E. Smith, "A study of branch prediction strategies," in *Proc. 8th Ann. Int'l. Symp. Computer Architecture*, pp. 135–148, June 1981.

[19] S. McFarling and J. L. Hennessy, "Reducing the cost of branches," in *Proc. 13th Ann. International Symposium Computer Architecture*, (Tokyo, Japan), pp. 396–403, June 1986.

[20] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," in *Proc. 16th Ann. International Symposium Computer Architecture*, (Jerusalem, Israel), pp. 224–233, May 1989.

[21] T. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proc. 24th Ann. International Symposium on Microarchitecture*, (Albuquerque, NM), pp. 51–61, Nov. 1991.

[22] S. Weiss and J. E. Smith, *POWER and PowerPC*. San Francisco, CA: Morgan Kaufmann, 1994.

[23] Hewlett Packard, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*. Palo Alto, CA: Hewlett Packard, 1994.

[24] S. A. Mahlke, *Exploiting instruction level parallelism in the presence of branches*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[25] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, "Instruction fetch mechanisms for VLIW architectures with compressed encodings," in *Proc. 29th Ann. International Symposium on Microarchitecture*, (Paris, France), Dec. 1996.

[26] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-assoctaive cache and prefeetch buffers," in *Proc. 17th Ann. International Symposium Computer Architecture*, (Seattle, WA), pp. 364–373, May 1990.

[27] L. Gwennap, "Intel's P6 uses decoupled superscalar design," *Microprocessor Report*, vol. 9, Feb. 1995.

[28] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the Hyperblock," in *Proc. 25th Ann. Int'l. Symp. on Microarchitecture*, (Portland, OR), pp. 45–54, Dec. 1992.

[29] P.-Y. Chang, E. Hao, Y. N. Patt, and P. P. Chang, "Using predicated execution to improve the performance of a dynamically-scheduled machine with speculative execution," in *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques*, June 1995.

A Algorithm for miss-path scheduling

```
while ( (Op = GetNextSequentialOp()) != NULL ) {
  1) Check Op's source operands to determine when all
  of them will be ready,
  EarliestStartTime=MaxReadyTime(Op's Src Operands).
  2) Determine ResourceSet, which is the set of
  resources that Op needs to execute: functional
  units, register ports, result buses, etc.
  3) Search the reservation table, starting from
  EarliestStartTime, for a cycle in which all
  resources in ResourceSet are available.
  4) When such a cycle - ScheduleCycle - is found,
  reserve the needed resources for Op by marking
  entries in the reservation table.
  5) Set the definition (def) time of all of OP's
  destination operands to ScheduleCycle+Latency(Op).
}
```