MPS: Miss-path Scheduling for Multiple-issue Processors

Sanjeev Banerjia, Sumedh W. Sathaye, *Student Member, IEEE*, Kishore N. Menezes, and Thomas M. Conte, *Member, IEEE*

Affiliation: This work was conducted while all of the authors were with the Department of Electrical and Computer Engineering, Box 7911, North Carolina State University, Raleigh, NC 27695–7911.

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, however, 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* or MPS. 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 simulations are presented that highlight the effectiveness of an MPS design.

Keywords

MPS, Multiple Instruction Issue, Miss Path Scheduling, Instruction Level Parallelism, Schedule Cache

I. Introduction

Current multiple issue processors such as the Hewlett Packard PA-8000 and the Intel Pentium Pro employ out-of-order instruction issue [1], [2]. One advantage of this approach is that compiler scheduling is not required to attain high levels of instruction level parallelism (ILP). Additionally, dynamic scheduling hardware can deal effectively with unanticipated run-time events (e.g., cache misses). There are also disadvantages. 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. What is desirable is the adaptability of an out-of-order issue design and the aggressive cycle time of a less complex design without the compatibility limitations of a strong dependence on compile-time instruction scheduling.

An alternative to dynamic scheduling within the processor core is *miss path scheduling*

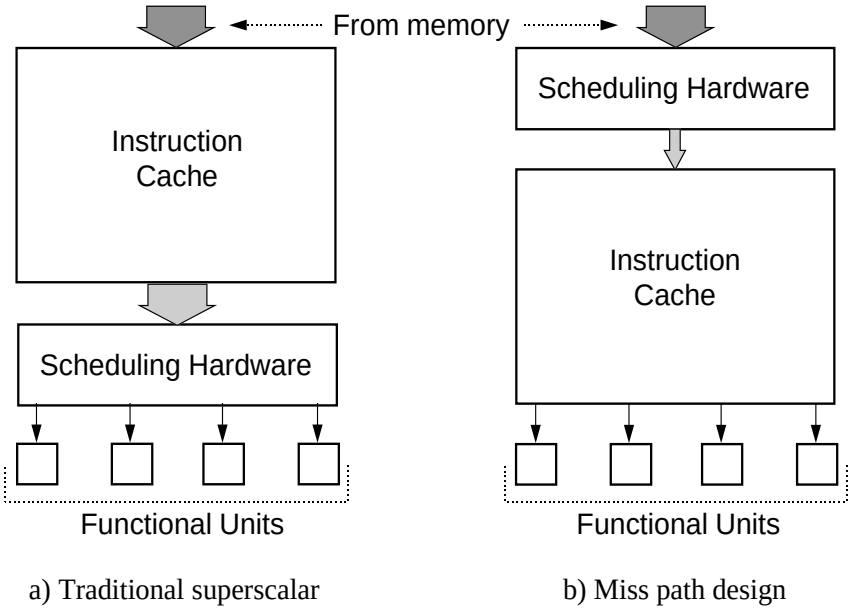

Fig. 1. A traditional superscalar vs. an MPS design: The traditional superscalar has an instruction cache which is accessed on every cycle for instructions. These instructions are then examined by scheduling hardware which determines which instructions are safe to issue in the current cycle and performs dynamic scheduling. In the MPS 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).

(MPS) (the basic idea is depicted in Figure 1). Scheduling hardware is inserted on the path between the instruction cache and the next level of memory. Instructions are scheduled for execution at instruction cache (i-cache) miss time. A schedule is composed of a set of wide instructions words, as in a VLIW machine [3]. As a schedule of instructions is formed, it is placed into a specially designed instruction cache. After a schedule is formed, its constituent instructions are issued to a simple but aggressively-clocked execution core.

Earlier schemes proposed by Melvin, Shebanow and Patt [4] and extended by Franklin and Smotherman [5] were among the first to use the paradigm of forming a schedule of instructions outside of the processor core. Their approaches scheduled between branch instructions only, limiting the amount of ILP the designs could extract. This paper describes a miss path scheduling design that speculates across multiple branches. An algorithm for

MPS is outlined, the details of the hardware implementation are discussed, and performance results are presented.

This paper is organized as follows. Section II discusses previous work related to miss path scheduling. Section III introduces MPS through an example and discusses how speculation is performed. Section IV describes the implementation of an MPS scheduler and presents detailed experimental results on the performance of MPS-scheduled code. Section VI introduces a special instruction cache called a schedule cache (SC) that holds the schedules constructed by MPS. Results for a proposed SC design are presented. Section VII concludes the paper.

II. 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 for use in a dynamically scheduled multiple issue processor. The goal was to form large groups of instructions called multinode words from which the scheduling hardware could more easily extract parallelism. A fill unit operates as follows. As instructions are received from memory, they are merged into multinode words and placed into a buffer. Filling completes when either the 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. Instruction fetch is performed 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 VLIW-like instructions to be placed into a shadow cache [5]. The idea was to build wide instruction words that contained no inter-instruction constraints on multiple issue, similar to a VLIW. Speculation across branches was not performed, but the design was able to fetch both paths of a branch, similar to a *tree instruction* [6]. 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). The fill unit accepts instructions from a backup instruction cache. A followup to the shadow cache idea was a proposal to use the fill unit for use in decoding a CISC instruction set [7].

Another miss path scheduling design was the expanded parallel instruction cache (EPIC) [8]. EPIC performs limited dynamic scheduling at cache miss to ease decode requirements at

run-time and form VLIW-like instructions from a RISC instruction stream. Limited speculation is performed: instructions that follow a branch are allowed to issue in the same cycle as the branch but not any earlier. A novel feature of an EPIC is that a cache line can hold multiple wide-words that are to be issued in different cycles. The trace cache takes a different approach to multiple-instruction issue [9]. 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 branch predictor. A trace consists 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. A back-up 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.

Nair and Hopkins recently described a technique called dynamic instruction formatting (DIF) [10]. A DIF-based machine consists of two execution engines, a primary (sequential) engine and a parallel engine. Instructions are initially executed on the primary engine. As an instruction sequence is executed, the dependencies between the instructions are detected and used to construct a schedule for the instruction sequence that can subsequently execute on the parallel engine. The schedule is placed into a DIF cache that feeds the parallel engine. A DIF machine has a learning mode and a parallel execution mode and two execution engines (logically distinct execution pipelines), one for sequential execution and the other for parallel execution. The first time an instruction sequence is fetched, it is scheduled and cached (in the DIF cache) for execution on the parallel engine as it executes on the sequential engine. Subsequent accesses to the instruction sequence access the schedule from the DIF cache. DIF and MPS have much in common but there are also significant differences. MPS uses one i-cache, a variant on a conventional design; DIF uses a conventional i-cache and a separate DIF cache to hold scheduled instructions. MPS uses one execution pipeline to execute all instructions; DIF uses two pipelines, the sequential and parallel engines. Because of these two reasons, a DIF design could require more die space than an MPS design. DIF does have the advantage of masking schedule formation

time by scheduling while executing sequential code on the sequential engine.

TABLE I

A CLASSIFICATION OF PREVIOUS WORK RELATED TO MISS PATH SCHEDULING. A DESIGN IS SAID TO SPECULATE INSTRUCTIONS IF SPECULATION IS PERFORMED BY MISS-PATH HARDWARE AND NOT BY DYNAMIC SCHEDULING HARDWARE IN THE EXECUTION PIPELINE.

SCHEME	SCHEDULES INSTRUCTIONS	SPECULATES INSTRUCTIONS	PARALLEL PIPELINES OR INSTRUCTION CACHE
Fill unit [4]	No	No	Yes
Shadow cache [5]	Yes	No	Yes
EPIC [8]	Yes	No	No
Trace cache [9]	No	No	Yes
DIF [10]	Yes	Yes	Yes
MPS	Yes	Yes	No

Table I summarizes the previous work related to MPS and characterizes the previous approaches based on several factors. The table provides a simple comparison between the various approaches and also lists the characteristics of MPS.

III. A SIMPLE SCHEDULING ALGORITHM

A variation of the operation scheduling algorithm as presented by Ellis [11] is well-suited for MPS. The algorithm is practical for hardware implementation because scheduling is performed by processing an instruction stream sequentially and does not require prior construction of a dependency graph. The algorithm is referred to as the MPS 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).

A. 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 fully pipelined integer

ALUs and a fully 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. A 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).

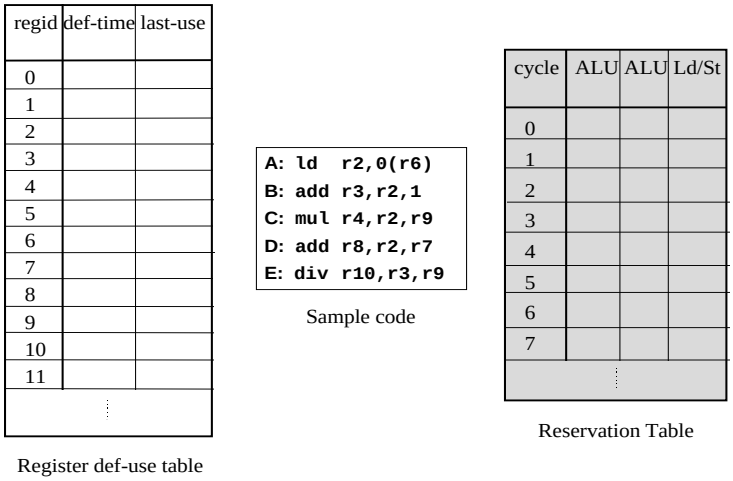

Fig. 2. Sample code and an empty schedule

Consider instruction A. A reads `r6` and writes `r2`. The def entry for `r6` in the def-use table is read to determine the last cycle in which `r6` was defined. As the def entry is empty, `r6` is already available, so A can potentially issue in cycle 0. The reservation table indicates that a load/store 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 reservation table is updated to reflect that A will begin 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 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.

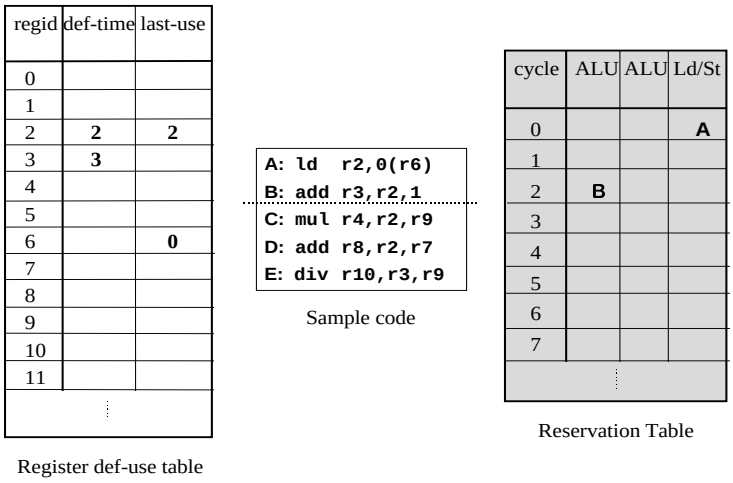

Fig. 3. State after scheduling instructions A and B.

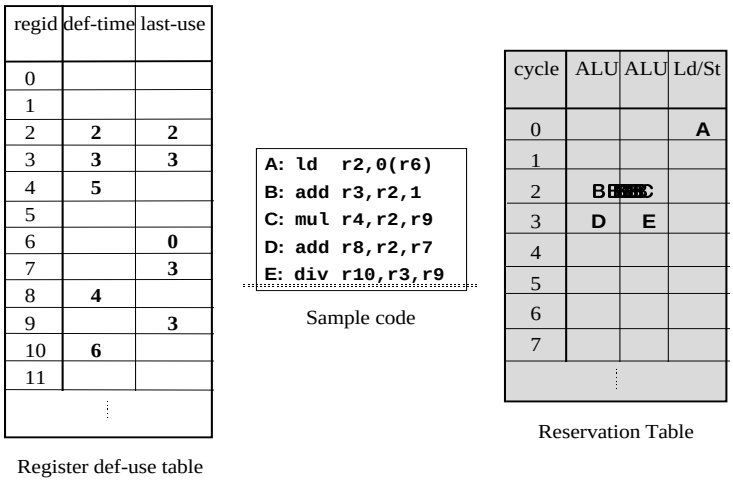

Fig. 4. State after scheduling all instructions.

The preceding example illustrates that the steps involved in miss path scheduling an instruction are simply: 1) checking the availability of source operands in the def-use table, 2) searching the reservation table for an available slot, and 3) updating the def-use and reservation tables. The algorithm requires only one pass through a set of instructions.

Each row in the reservation table represents a group of instructions that issue in parallel, as in a VLIW machine. Using terminology from Rau [12], each row is labeled a *MultiOp*: a group of multiple operations or *Ops* that issue in parallel (as in a VLIW). The data structures required for scheduling MultiOps can be implemented easily in hardware. Since the def-use table holds information about individual (architected) registers, it can be integrated within a register file. It can also be constructed as a separate (smaller) table with potentially faster access. 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 is 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 IV discusses the additional requirements for miss path scheduling.

The example in Figures 3-4 considers only flow dependencies. Anti- and output dependencies must also be honored. 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 for 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`.

B. Speculative scheduling at miss time

The previous example illustrated how miss path scheduling works on code without branches. Scheduling in the presence of branches is as follows: as a sequence of non-branch instructions is scheduled, a `LatestScheduleTime` is 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 multiple branches. When a conditional branch is encountered, the scheduler predicts the direction of the branch and begins scheduling instructions from the predicted path. Instructions from the predicted

path may be scheduled higher (earlier) than the preceding branch. They may be placed ahead of *several* preceding branches. When speculation is performed, speculated instructions must be prevented from retiring their results when their control-dependent 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].

IV. Details of a miss path scheduler

The previous section introduced some of the hardware structures required for an MPS implementation. The data stored in the def-use table and the reservation table are used to make scheduling decisions. This section details the requirements on this scheduling logic.

A. Scheduling logic

Section III-A informally outlined that an MPS design can schedule an instruction I in three steps. To review briefly and add some details, these steps are:

1. Read I's source and destination operands' def-use times from the def-use table and use these values to compute the earliest clock cycle – `ScheduleTime` – in which I can begin execution. In parallel, use I's opcode to determine what resources it requires to execute.
2. Search the reservation table starting at cycle `ScheduleTime` for a cycle in which resources are available for I to commence execution. Update `ScheduleTime` with the value of this cycle.
3. Use the final value of `ScheduleTime` to update the def-use table and the reservation table. In parallel, place I into an appropriate position in the instruction cache. (I-cache issues are left vague purposefully at this point. They will be discussed in detail in Section VI.)

These three steps can be performed in as few as four (4) clock cycles: two cycles for Step 1 (one cycle to read values, an additional cycle to compute `ScheduleTime`), one cycle for Step 2, and one cycle for Step 3. (The lookup of the reservation table could require more than 1 cycle if the number of cycles that are searched is large. However, we will show in Section V that the number of cycles to search is usually moderate, and so this table can

in fact be small enough to search in 1 cycle.) Therefore, a miss path scheduler requires a minimum of four cycles to schedule an instruction.

The number of ports required on the def-use table is moderate. Accessing an entry in the def-use table accesses all of the information regarding the corresponding register: last-use time and def-time. Step 1 perform reads and Step 3 performs writes 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 and 3 require three read ports and three write ports, respectively, to update the def-use table. So the maximal overall port requirements on the table 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.

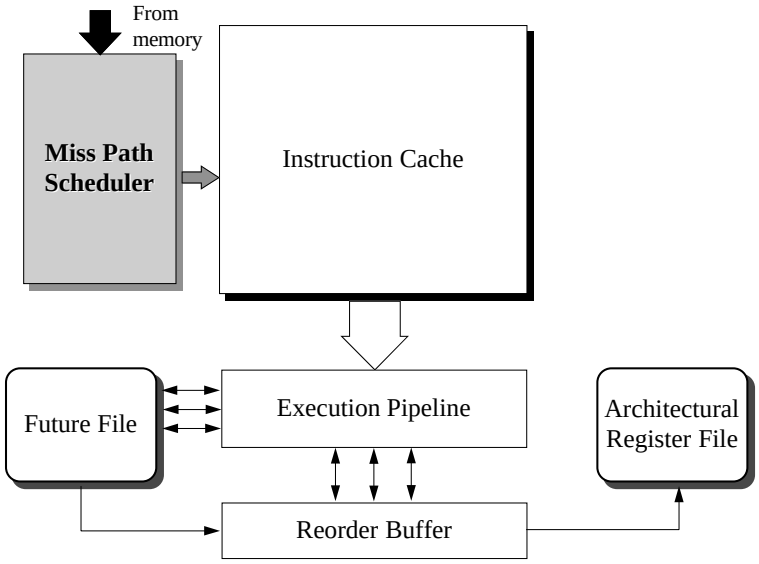

Fig. 5. Hardware organization of a miss path scheduler. The def-use table and reservation table are part of the Miss Path Scheduler in this diagram.

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 conventional 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. A schedule of instructions is formed starting from the cache miss address. The processor does not execute any instructions while scheduling take place.
4. After schedule formation/cache miss processing completes, use the PC to re-access the cache for instruction fetch. Wide-words from the newly-formed schedule are sent to the processor core for execution.

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 implements basic-block-only scheduling). Because scheduling is not performed across the entire program by one invocation of the miss path scheduler (one cache miss), dependencies between schedules are unknown to the scheduler. Such dependencies can be handled within the execution core by using a scoreboard [16], [17].

There is a sequential nature to the three 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. The scheduling pipeline mimics an execution pipeline in its ability to interlock and stall by using busy bits as in a conventional scoreboard. Figure 6 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. The appendix contains a detailed diagram of a pipeline for miss path scheduling along with a detailed explanation.

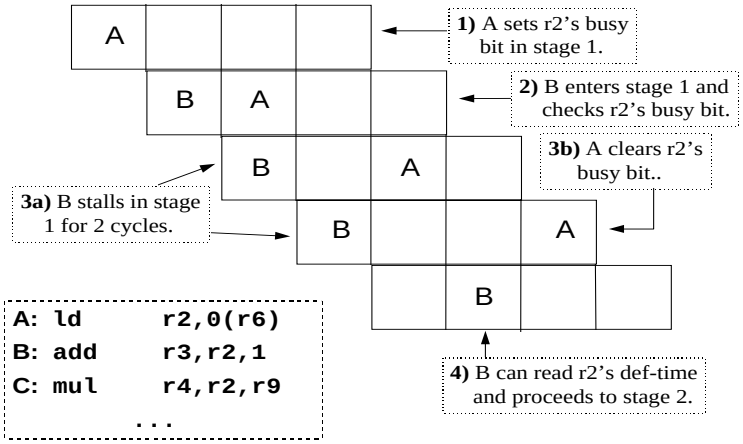

Fig. 6. Stalling due to RAW hazard in the pipelined implementation.

B. Speculative miss path scheduling

Extracting larger amounts of ILP from most general purpose programs requires scheduling beyond basic blocks. Miss path scheduling can perform *speculative scheduling*, as explained in Section III-B. 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).

The central issue in speculating instructions is choosing *which* instructions to speculate, a decision that relies on predicting which path a branch will take. For architectures that support a prediction bit in the instruction encoding [18], [19], the scheduler can use this bit to make a prediction. The bit can be set based on profile information – termed *profiled prediction* – 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 forward-taken (FT) can be implemented in hardware. Dynamic prediction can also be used.

When a branch is predicted taken by the MPS scheduler, the scheduler issues memory requests using the branch target address to change the path along which instructions are fetched. The cost of the memory latency (in clock cycles) 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 scheduler pipeline, after the immediate field has been decoded. Indirect branches use a register source operand to determine the target address and are more difficult to process. The branch target address is not known, so a direction-based prediction cannot be made. If a predictor that can predict indirect branches is used, the branch target address is unknown, so for a predicted-taken branch, the scheduler is unable to fetch and schedule instructions from the predicted target. For this reason, the current design of MPS stops scheduling when it encounters an indirect branch (this is a halt condition). MPS also stops scheduling when it encounters a backward branch to prevent loop unrolling, as we have yet to develop robust heuristics for schedule-time estimation loop trip counts.

When a schedule S is executed the entire way through (no internal branches are mispredicted), the machine needs to know the address of the next schedule to execute. If S does not contain any branch instructions, the address calculation is simply $next\_address$ = starting address of S + # of instructions in S. If S contains branch instructions, the calculation is not so straightforward and in fact cannot performed when the execution of S completes. The MPS scheduler tracks the path along which S is formed and computes a $next\_address$ field after the scheduling of S completes. When S is executed, its associated $next\_address$ is passed to the machine and is used to locate the next schedule.

V. Experimental evaluation of a miss path scheduler

The performance of an MPS scheduler was evaluated using trace-drive simulations. All eight of the integer programs from the SPEC95 suite were used as the benchmark set (reference inputs were used). The programs were compiled with the IMPACT compiler from the Univeristy of Illinois [20] 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 MPS scheduler and the execution pipeline, including the effects of mispredicted branches and mis-speculation. The execution pipeline has four stages – fetch, decode, execute, and writeback. The processor core has eight fully pipelined functional units (the configuration and latencies are listed in Table II). The machine supports execution of multiple branches in parallel on any of the four I-ALUs.

The branch misprediction penalty is two cycles. A 32 bits/cycle bandwidth with a five cycle latency is assumed between the scheduler and next level of memory. Each benchmark was run for 20,500,000 instructions to bypass initialization code. Statistics were gathered by running the program for up to an additional 200,000,000 instructions or until program completion. An infinite instruction cache and a perfect data cache were assumed.

TABLE II

MPS EXECUTION CORE CONFIGURATION

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

TABLE III

CYCLES TO FORM A SCHEDULE: SPECULATIVE SCHEDULING ACROSS AN UNLIMITED NUMBER OF BRANCHES. EACH ENTRY IS AN AVERAGE NUMBER OF CYCLES.

Benchmark	Pipelined	Unpipelined	Stall cycles	Instruction count	Schedule Length
go	50.97	68.72	24.52	17.24	7.32
m88ksim	35.51	40.52	16.98	8.79	4.76
gcc	38.69	43.28	18.25	10.34	5.81
compress	26.50	31.87	10.71	6.77	5.04
li	26.64	32.02	10	7.06	4.69
ijpeg	94.23	62.94	70.23	14.56	5.67
perl	41.28	52.76	20.17	11.94	7.68
vortex	52.15	79.39	25.24	18.76	10.22
Harmonic mean	41.07	46.41	17.73	10.48	5.99

There are two aspects of an MPS scheduler that are interesting: the performance of the code produced by the scheduler and the performance of the scheduler itself. Since the goal is to produce parallel code for multiple-issue processors, speculative scheduling is the focus for the remainder of the paper. Scheduler performance is briefly examined first. The time required to construct a schedule is time that the processor core is not executing

instructions and therefore is an overhead of the MPS paradigm[1]. The performance of pipelined and unpipelined miss path scheduling was measured for speculative scheduling across an unlimited number of branches. (Profiled branch prediction was used. Training inputs were used for the profiling runs.) The results are presented in Table III. The pipelined implementation required 12% fewer clock cycles to form a schedule. A pipelined design was assumed for the remainder of the results in this paper due to its superior performance with little hardware overhead.

Table III also presents the number of stall cycles when scheduling and instruction counts and schedule lengths for the schedules. The time to build a schedule is on average 137% greater than the time to fetch the instructions from the L2 cache/memory. The additional time for scheduling is caused by the time to drain the scheduling pipeline (a fixed quantity dependent on the pipeline depth) and by stalls in the scheduling pipeline. Recall that scheduling stalls are caused by two things: a data hazard (as shown in Figure 6) or changing the memory fetch path to fetch instructions from a non-sequential branch target address. Since the IMPACT compiler performs code positioning, *i.e.*, the likely target of a conditional branch is positioned as the fall-through, the scheduler rarely had to change the fetch path. As Table III shows, the time to build a schedule roughly fits the equation Time to schedule = memory latency + # instructions + # scheduling stall cycles + depth of scheduling pipeline. Although constructing an individual schedule requires considerably more time than simple cache miss repair, the extra cost is amortized over time if the schedule peforms well and is executed frequently.

The instruction counts for schedules varied widely across programs, ranging from six to eighteen. The schedule lengths were invariably shorter than the instruction counts, indicating that every program contains an intrinsic amount of ILP. Additionally, the schedule lengths were moderate, ranging from 4.69 to 10.22. Recall from Section IV-A that the MPS scheduler has to search the reservation table for an open execution slot using a priority encoder. If the range of cycles to search is excessive, the search could take multiple clock cycles or impact the cycle time of the entire machine. The moderate schedule lengths imply that the priority encoder usually has to search a very small range and therefore will

[1]Note that scheduling time is independent of the instruction and data caches. An instruction cache miss triggers schedule formation but does not affect scheduling time. The data cache is not used during scheduling.

not take multiple cycles or stretch the cycle time.

TABLE IV

BRANCH STATISTICS. THE TOTAL NUMBER OF PREDICTIONS AND MISPREDICTIONS AND THE MISPREDICTION RATE ARE SHOWN FOR PROFILED BRANCH PREDICTION AND A FORWARD TAKEN (FT) HEURISTIC. THE AVERAGE NUMBERS OF BRANCHES PER TRACE AND EXIT BRANCH PER TRACE ARE SHOWN FOR PROFILED PREDICTION ONLY.

Benchmark	# predictions	# mispredictions		Misprediction rate (%)		Avg # branches / schedule	Avg off-path branch
		Profiled	FT	Profiled	FT		
go	33879239	6382031	11171576	18.81	32.97	2.78	1.94
m88ksim	43988477	2672101	5339013	6.07	12.14	2.067	2.065
gcc	8608960	943322	1712120	10.96	19.89	3.01	2.25
compress	41059112	6795823	17440778	16.55	42.48	1.79	1.43
li	31985553	2926321	6505103	9.15	20.34	1.94	1.81
ijpeg	20732512	3847910	11133683	18.56	53.70	1.58	1.45
perl	60901007	5687505	8820516	9.34	14.48	4.51	3.35
vortex	32086029	1137045	4744401	3.54	14.79	3.28	3.11
Harmonic mean				11.62	25.34	2.62	2.18

We now focus on the characteristics of the code produced by the scheduler. The quality of an MPS-produced code schedule is heavily influenced by two factors: the instruction count and the accuracy of the branch predictor. Table IV presents statistics on the behavior of conditional branches in the benchmark programs. Two branch prediction strategies were simulated: a static forward-taken (FT) heuristic and profiled prediction. As expected, profiled prediction yielded a much lower misprediction rate than the simple FT heuristic. In fact, profiled prediction provided an accuracy rate of just under 90%, a figure competitive with simple hardware schemes [21]. The number of branches per schedule and the point where control-flow exited a schedule are also presented. The former is a static (schedule-time) measure of the number of basic blocks in a schedule, and the latter is a dynamic (processor core execution-time) measure of how many basic blocks in a schedule are executed and retired. For example, a typical schedule in perl contains 4-5 basic blocks and executes 3-4 of these blocks. The average number of basic blocks per schedule ranges from 1.58 to 4.51 and is significantly greater than the code available from basic-block

TABLE V

SPECULATIVE EXECUTION STATISTICS. THE TOTAL NUMBER OF INSTRUCTIONS SPECULATED BY THE SCHEDULER AND THE TOTAL NUMBER OF SPECULATED INSTRUCTIONS THAT WERE USEFUL (RETIRED THEIR RESULTS) ARE SHOWN, ALONG WITH PER SCHEDULE AVERAGES.

Benchmark	Total # spec. instrs	Total # useful spec. instrs	Avg. # spec. instrs / schedule	Avg. # useful spec. instrs / schedule	Useful yield
go	103266056	72540549	5.92	4.16	70.25
m88ksim	54905198	48201595	2.58	2.26	87.79
gcc	21811840	16598611	5.71	4.35	76.10
compress	54840212	33645027	1.90	1.17	61.35
li	40707387	32775809	2.30	1.85	80.52
ijpeg	17535310	12898932	1.23	0.91	73.56
perl	203523766	167614221	11.19	9.22	82.36
vortex	80256045	73679062	7.79	7.15	91.80
Harmonic mean			2.74	2.19	82.39

TABLE VI

SPECULATIVE EXECUTION STATISTICS FOR FT PREDICTION. THE TOTAL NUMBER OF INSTRUCTIONS SPECULATED BY THE SCHEDULER AND THE TOTAL NUMBER OF SPECULATED INSTRUCTIONS THAT WERE USEFUL (RETIRED THEIR RESULTS) ARE SHOWN, ALONG WITH PER SCHEDULE AVERAGES.

Benchmark	Total # spec. instrs	Total # useful spec. instrs	Avg. # spec. instrs / schedule	Avg. # useful spec. instrs / schedule	Useful yield
go	103266042	72540110	5.92	4.16	70.24
compress	54840202	31605007	1.90	1.10	57.63
li	40707387	32475809	2.30	1.84	79.78
ijpeg	19760873	11784191	1.20	0.70	59.63

scheduling. The dynamic behavior of exits from the schedules shows that 1.45-3.35 basic blocks are executed before a branch is mispredicted and control exits the schedule, off of the predicted path (off-path).

The benefit of scheduling across multiple basic blocks is measured by the number of speculated instructions that retire their results. Table V presents data on code speculation performed by the MPS scheduler. The number of speculatd instructions per schedule ranges from under two to over eleven. The percentage of speculated instructions that retired their results – which we term *useful yield* – ranges from 60% to over 90%. For all

benchmarks, the useful yield is over 60%. This indicates that the MPS scheduler does an effective job of speculation.

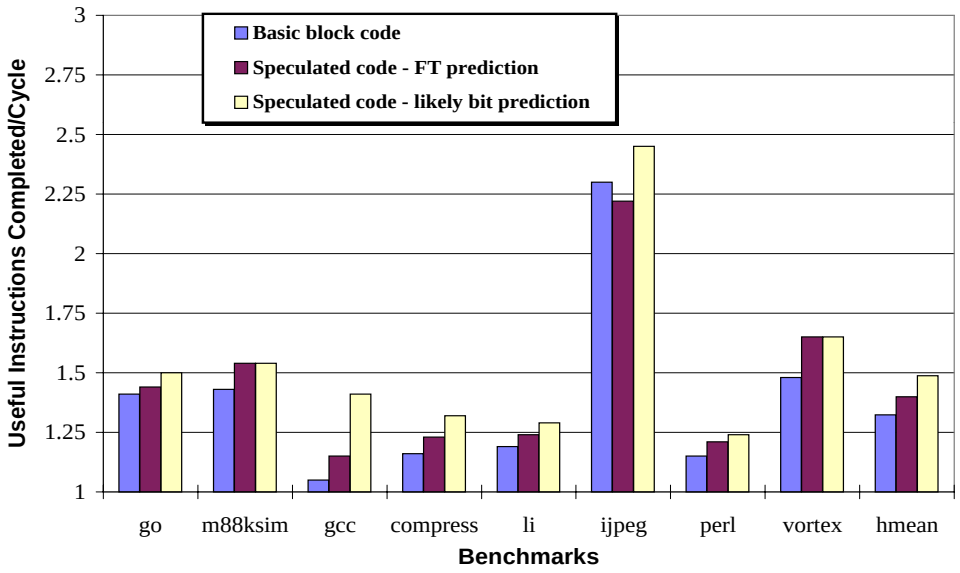

Fig. 7. Performance of basic-block-only and speculative schedules. The speculative scheduling results used two branch prediction strategies: a FT heuristic and a likely-bit predictor. The metric used is useful instructions completed per cycle (IPC). Each individual bar represents the IPC for the indicated program and scheduling scheme. Harmonic means for the various schemes are shown in the rightmost set of bars.

The data in Tables III- V can help intuite which programs will perform well using MPS but the final measure is the execution times of the programs. A metric that captures overall performance is Useful Operations Completed per Cycle (OPC). The MPS simulator keeps precise counts of the number of instructions issued and the number that retire their results. Since scheduling does not introduce any additional instructions, *i.e.*, no code expansion occurs, OPC can be used to compare between different configurations of MPS schedulers as well as with other machine models, such as a superscalar processor. The performance of a speculative MPS scheduler was measured. The scheduler speculated across an arbitrary number of branches, with halt conditions of a backward branch or a branch through a register. (Due to our experimental framework, the scheduler was not able to schedule across library calls, which an actual hardware implementation would be able to do. Therefore, procedure calls were also used as a halt condition.) The branch

prediction strategies used were an FT heuristic and profiled prediction. The results are presented in Figure 7. Basic blocks results are included to measure if speculation proved beneficial to overall performance. In general, speculation yielded better performance than basic block scheduling across all programs. The exception is on ijpeg, where the FT predictor produced a lower OPC than basic block scheduling due to its high misprediction rate (¿ 53%). Profiled prediction consistently outperformed both basic block scheduling and speculation with the FT predictor. (For m88ksim and vortex, the differences between profiled prediction and FT are small.) The results reinforce the intuition that with good branch prediction, speculation can improve performance over basic block scheduling

The results also demonstrate that for some programs, speculation is beneficial even with a poor predictor. Although go, compress, and li suffer from branch misprediction rates in excess of 20% when using an FT predictor, their OPCs are still higher than for basic-block scheduling. The data in Table VI illustrates why. The useful yield for all three programs decreases only marginally even though the misprediction rate increases by 75-156%. Contrast this with ijpeg, for which a 189% increase in the misprediction rate reduces the useful yield by 19%.

VI. Instruction cache support: the Schedule Cache

An i-cache for an MPS-based machine must hold schedules of instructions and therefore is called a *schedule cache (SC)*. The requirements for an SC differ from those for a traditional i-cache and in some respects exceed them. (For the remainder of this paper, a traditional i-cache is referred to as an i-cache and a schedule cache as an SC). This section describes these requirements and suggests two potential organizations for an SC.

A. Requirements for a Schedule Cache

One of the differences between an i-cache and an SC is the unit of placement and replacement. In an i-cache, the unit is a cache line or if sub-blocking is used, a sub-block. In an SC, the unit is a schedule. MPS reorders instructions so that the instructions in a schedule are not necessarily in program order. If any part of a schedule is displaced, the entire schedule is invalid. Therefore, one of the requirements of an SC is invalidation of a complete schedule S when any portion of S is displaced.

The location of a schedule in an SC – the location of the first MultiOp in the schedule – is determined using bit selection on the address of the first instruction in the schedule (this is identical to normal cache addressing.) However, the high-order bits of the address are not sufficient for tagging a schedule. If multiple schedules are formed starting from addresses that have identical high-order bits and map to the same SC location, the tag match cannot determine which of the schedules is cache-resident. However, the offset bits of the address differentiates between instructions. Therefore a schedule is tagged with a concatenation of the high-order bits and the offset bits of the first instruction address[2]. This tagging scheme differs only slightly from that in a traditional i-cache and permits direct-mapped or set-associative organizations.

Each branch instruction in a schedule has associated with it the prediction made for it when it was scheduled. The prediction has to be supplied to the processor core so that it can be checked against the (execution-time) outcome of the branch, to see if the execution remains on-path (within the current schedule) or will go off-path. An SC must store each branch's prediction (only 1 bit per branch is required).

A cache must also store the *next_address* value associated with each schedule.

B. The uncompressed schedule cache design

One potential organization for an SC is a traditional i-cache with additional hardware. The design is similar to that for the uncompressed cache described in Conte, *et al.*,[22] and is called an uncompressed SC. The design is shown in Figure 8. Each cache line holds one MultiOp. The cache index of the first MultiOp in a schedule is determined using bit selection, as described above. The remaining MultiOps for the schedule are located in consecutive cache lines. Single-word sized sub-blocks are used to allow the placement of instructions into arbitrary positions within a cache line. NOPs are used as placeholders for issue slots that do not issue an instruction during a particular cycle. In this respect, the uncompressed SC is a mirror of the reservation table. NOPs used as placeholders within a MultiOp are termed *horizontal NOPs*. In Figure 8, MultiOp B consists of three

[2]This assumes that multiple schedules starting from the same instruction cannot concurrently reside in the SC. Such *path associativity* [9] can be implemented within an SC with minor additional hardware but is not explored here to simplify the discussion.

instructions. The remaining five issue slots are not scheduled to issue an instruction and are horizontal NOPs. An empty cycle in a schedule in which no instructions are issued, *i.e.*, the MultiOp consists entirely of NOPs, is termed a *vertical NOP* and is represented as a cache line consisting entirely of NOPs. In Figure 8, the cache line encircled by the dotted line is a vertical NOP.

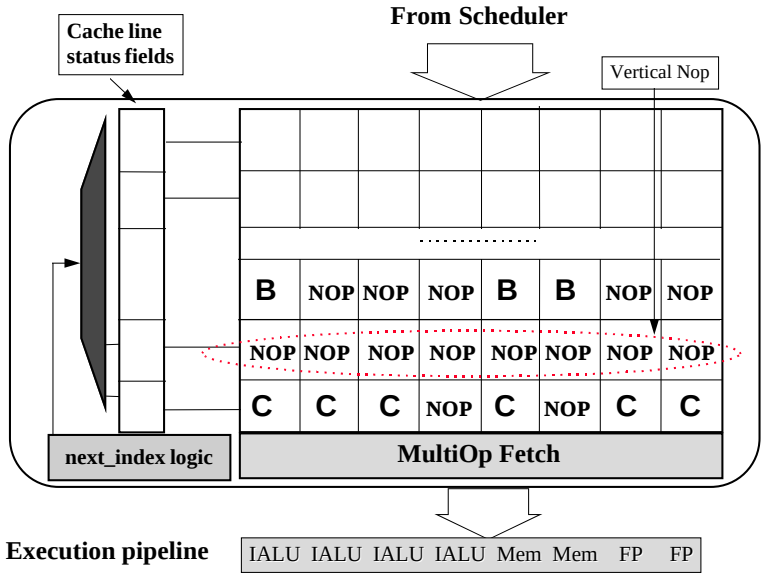

Fig. 8. The uncompressed schedule cache. The design holds NOPs as placeholders for unused issue slots.

Each cache line has several status fields associated with it that aid in cache access:

- Line valid bit: When set, this bit indicates that the cache line is valid.
- Op valid bits: One bit for each issue slot in a cache line. When set, the corresponding Op is valid and is passed on to the functional unit. When the bit is clear, the functional unit is issued a NOP.
- Distance field: This field is set to the number of cache lines between the current cache line and the cache line that holds the first MultiOp in the schedule. Theoretically, a schedule can span the entire cache, so for a cache with n lines, $\log_2 n$ bits are required for the field. In an implementation, however, there will be a restriction on the maximum length of the schedule to m cycles ($m \leq n$), for which the field width is $\log_2 m$ bits.
- Last line bit: When set, the cache line is the last line in a schedule. A vertical NOP

is identified by a cleared line valid bit and a cleared last line bit (this indicates that the line was not written to during schedule formation but is not the last line in the schedule).

- Next address field: This is the address to use for the next cache access if the schedule executes the entire way through (stays on-path). The field is set for only the last cache line in a schedule.

Cache access in an uncompressed SC with n lines proceeds as follows (assume a direct-mapped organization). Bit selection on the PC is used to generate an *index* value. The tag at *index* is matched with the PC to check if the requested schedule is cache-resident. If a tag match occurs and the valid bit is set, this is an SC hit, and the MultiOp is fetched and passed to the execution core. Since subsequent MultiOps are located in consecutive cache lines, the index for the next cache access ($next\_index$) is generated using an increment of the form $next\_index = (index + 1) \bmod n$. A PC value from the processor core is not required for cache access as long as execution remains on-path.

When a cache miss occurs, the scheduler is invoked with the PC and begins constructing a new schedule S starting at cache line *index*. When an instruction for S is placed in a cache line for the first time, the line's distance field is set (the distance value is available from the scheduler's reservation table) and its valid bit is set (when an instruction is placed into an issue slot, the issue slot's Op valid bit is set also). If another schedule S_{old} is resident at *index*, at least part of it will be replaced. Any portion of S_{old} that is not replaced is marked invalid (invalidation is discussed shortly). The new schedule S is constructed and placed into the cache. When scheduling halts, the last line in S has its last line bit and next address fields set, and execution resumes as the MultiOps that comprise S are sent to the processor core.

Schedule invalidation is achieved using the distance field. When a MultiOp M for schedule $S_{invalid}$ is displaced from the cache, the distance field for the cache line indicates the cache location of the first MultiOp, M_{first}, for $S_{invalid}$. The line valid bit for M_{first}'s cache line is cleared. The next cache access for $S_{invalid}$ will result in a cache miss.

C. The compressed schedule cache design

Another organization for an SC is a design similar to the compressed cache described in Conte *et al.*[22] and is called a compressed SC. The design is shown in Figure 9. Unlike the uncompressed SC, the compressed SC does not hold horizontal or vertical NOPs, and multiple MultiOps can occupy a cache line. The compressed SC is organized as two separate banks. When a MultiOp spans two cache lines, banking allows access to all of the instructions within the MultiOp in a single cache access [23]. The cache index of the first MultiOp in a schedule is determined using bit selection. The first MultiOp in a schedule always begins at offset 0 within its cache line.

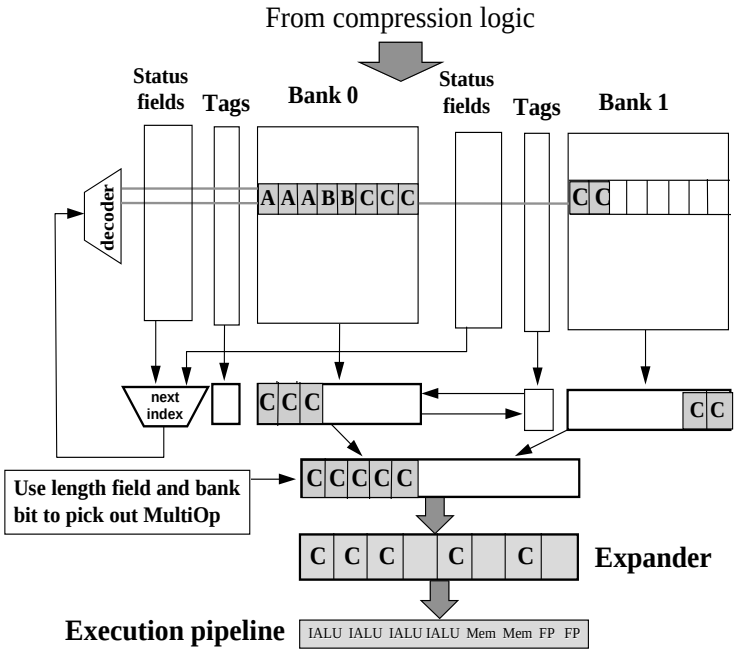

Fig. 9. The compressed schedule cache. The design does not hold NOPs as placeholders for empty issue slots. The cache is written to from a schedule buffer rather than directly during scheduling.

Cache access for successive MultiOps is performed by using the length of the current MultiOp to determine the bank and the offset of the next MultiOp. The generation of *next_index* is similar to a scheme detailed in [24], with minor differences. Since a MultiOp can start anywhere in a cache line, a length field is maintained for every offset within a cache line. The length of each MultiOp is computed during schedule formation and is stored in the length field. The length field is added to *index* to generate *next_index* and

the offset for cache access. A bank bit is associated with every offset in a cache line to direct cache access to the proper bank. As with the uncompressed SC, a PC value from the processor core is needed only when a branch misprediction occurs. Each cache line in the compressed SC also has the same fields as a line in the uncompressed cache (except for the line valid and Op valid bits, which are no longer needed), and they are used in the same manner.

Schedule formation with the compressed SC differs from that with the uncompressed SC. The compressed SC does not mirror the reservation table as the cache does not hold NOPs (except possibly in the last cache line in a schedule, and these are just padding NOPs). This implies that instructions cannot be placed into the cache during scheduling. Rather, instructions are written into a *schedule buffer*. The schedule buffer has the same dimensions as the reservation table and is organized as a small uncompressed SC. When a halt condition is encountered, MultiOps are fetched directly from the schedule buffer and passed to the processor core. As MultiOps are sent to the core, they are placed into the compressed SC. As a MultiOp is placed into the cache, it is processed by compression logic that removes horizontal NOPs. Vertical NOPs are represented by a hardware *pause field* associated with each MultiOp (in hardware, a pause field is implemented for every instruction in a cache line). When an empty MultiOp is encountered during a copy from the schedule buffer, the pause field for the previous MultiOp $M_{previous}$ is set to one. For every consecutive empty MultiOp encountered, $M_{previous}$'s pause field is incremented. A pause field of value m instructs the issue logic to halt MultiOp issue for m cycles after issuing the current MultiOp. If a schedule S goes off-path while it is being fetched from the schedule buffer, S is marked invalid in the cache.

Since a MultiOp is stored in a compressed fashion in the cache, it must be uncompressed so that each individual instruction is aligned with the correct functional unit before the MultiOp is passed to the execution pipeline. The logic to accomplish this is called an expander and resides between the cache and the execution pipeline [22]. The functionality of the expander is directly opposite to that of the compression logic used when placing a MultiOp into the SC. The expander adds an extra stage to the pipeline and increases the branch misprediction penalty by one cycle.

D. Experimental evaluation

Simulations were conducted to gauge the performance of the compressed SC design with the MPS scheduler. (The uncompressed SC was not evaluated because a prior study [22] demonstrated that it does not achieve high performance on a multiple-issue machine similar to the one used for this study.) As with the experiments in Section V, each benchmark was run for 20,500,000 instructions to bypass initialization code. The last 500,000 of these instructions were used for d-cache and SC warmup. Measurements were taken on next 200,000,000 instructions or until the program completed. A 5 cycle latency with a 32 bits/cycle bandwidth to a perfect L2 cache was assumed.

Since the MPS design uses a statically-scheduled processor core, a potential source of performance degradation is data cache (d-cache) misses, because execution of all instructions in a MultiOp stalls on a d-cache miss. The primary concern is d-cache read accesses that can cause a dependent instruction or a chain of instructions to stall. To include the effects of read misses, a d-cache was also modelled. The d-cache was 32KB in size with 32-byte lines. LRU replacement was used with 4-way set associativity.

TABLE VII

PERFORMANCE OF THE COMPRESSED SCHEDULE CACHE. SPECULATION ACROSS AN UNLIMITED NUMBER OF BRANCHES WITH PROFILED BRANCH PREDICTION WAS USED. THE METRIC USED IS OPC.

Benchmark	Perfect	32KB SC			64KB SC,
	SC	1-way SA	2-way SA	4-way SA	4-way SA
go	1.50	0.28	0.33	0.41	1.36
m88ksim	1.72	0.29	0.29	0.76	1.71
gcc	1.41	0.18	0.24	0.34	1.02
compress	1.32	1.32	1.32	1.32	1.32
li	1.31	0.28	0.47	0.91	0.98
ijpeg	2.45	1.24	1.60	2.34	2.44
perl	1.29	0.19	0.21	0.61	0.82
vortex	1.65	0.23	0.31	0.36	1.42

Results are presented in Table VII. For all of the benchmarks except compress, a 32KB direct-mapped SC yielded performance several factors lower than the parallelism the MPS scheduler is able to extract. Two exceptions are compress are compress and ijpeg. compress

achieved performance equivalent to using an ininite SC, since it has a very small footprint of schedules which a 32KB cache is able to capture. ijpeg has a relatively small footprint also and has the most intrinsic parallelism of all of the programs, as evidenced by its OPC of 2.45 with an infinite SC.

The performance of all programs improved as the associativity of the 32KB SC was increased from 1 to 4 but still all programs except for compress and ijpeg attained OPCs of less than 1. A 64KB, 4-way set-associative SC was simulated in an attempt to determine if capacity misses were the major source of the poor behavior. Performance for all programs increased dramatically with the 64KB SC, indicating that indeed capacity is a major issue in schedule cache design. perl and li remained in ¡ 1 OPC range but showed significant improvement over the 32KB design.

Since an SC holds schedules that occupy multiple cache lines, both conflict and capacity misses occur more frequently than in a traditional i-cache. A schedule can reside at indices i through $i+n$ and thus will have a conflict with any other schedule that maps to any cache line in that range. This trait can be mitigated with associativity. Schedule size can also exacerbate capacity issues. Multiple schedules can contain the same basic blocks (they will not *begin* with the same basic block). The code expansion causes capacity contention in the cache and requires a large schedule cache for acceptable performance.

E. A comparison with a superscalar design

As stated in the introduction, MPS is a possible alternative design paradigm to out-of-order superscalar processors. The fundamental difference is that the MPS scheduling hardware is not used in the processor core and does not have to resolve dependencies between a large number of instructions within a central hardware window. MPS does not use register renaming so it does not require potentially complex hardware to allocate and free renaming registers.

We performed a set of experiments to compare the performance of an MPS machine with that of a superscalar machine. The characteristics of the superscalar machine are listed in Table VIII. The superscalar simulator is not within the exact same framework as the MPS simulator, so we were not able to model identical execution cores. For example, the superscalar machine can execute only 1 branch per cycle, whereas the MPS machine

TABLE VIII

SUPERSCALAR EXECUTION CORE CONFIGURATION

General parameters		
Window size	56	
Speculation	Speculate across 16 branches	
I-cache	64KB, 32B lines, 4-way SA, LRU	
D-cache	Same as MPS config	
Execution Core		
Functional Unit	Quantity	Latency
Integer ALU	4	1
Branch	1	1
Load	2	2
Store	2	1
FP Adder	1	1
FP Multiplier	1	3
FP Divider	1	9
FP Branch	1	1

can execute up to 4 branches in parallel. However, the superscalar core has a richer set of functional units, a large central window, and the ability to speculate instructions past up to sixteen unresolved branches.

TABLE IX

PERFORMANCE OF AN MPS MACHINE VERSUS A SUPERSCALAR MACHINE. EACH ENTRY IS AN OPC.

Benchmark	Infinite i-cache/SC		64KB i-cache/SC	
	MPS	Superscalar	MPS	Superscalar
m88ksim	1.71	1.68	1.71	1.68
gcc	1.41	1.50	1.02	1.45
compress	1.32	1.09	1.32	1.09
li	1.31	1.42	0.98	1.42
ijpeg	2.45	1.70	2.34	1.70
perl	1.29	1.45	0.82	1.45

The results are presented in Table IX. (The programs go and vortex are not included as we were unable to simulate them in the superscalar framework.) Data is presented for simulations with infinite caches and with a 64KB i-cache/SC (a data cache was simulated in

all of the experiments). The results are mixed. When using an infinite cache, MPS yields higher OPCs for 3 of the 6 programs. When a 64KB cache is modelled, MPS outperforms the superscalar design on the same 3 programs. Those 3 programs – m88ksim, compress, and ijpeg – have relatively small SC footprints, as confirmed by the data in Table VII. On the other 3 programs, the superscalar outperforms MPS using an infinite cache or a 64KB i-cache/SC. Those 3 programs – gcc, li, and perl – have large SC footprints, and the performance gap between MPS and superscalar widens significantly when a 64KB cache is simulated. This reinforces the earlier observation that the large unit of [re]placement in an SC – a schedule – can cause MPS to be SC-size sensitive.

However, these cycle-count-based number do not reflect the clock speed advantage that an MPS design could have. Taking that into account only increases the performance of an MPS machine, so it is possible that in the final analysis an MPS machine could outperform a superscalar even if producing lower OPCs.

VII. Conclusion

This paper has presented and developed an alternative to out-of-order issue that relegates the complexity of speculative scheduling to the instruction cache miss path. The technique is called miss-path scheduling and has the advantage of a potentially aggressive cycle time for a simplified processor core. The central idea is to schedule sequences of instruction for execution at cache miss time rather than use dynamic scheduling hardware in the processor core. The details of a hardware design for a miss-path scheduler called MPS were presented. The hardware was shown to be practical for implementation, with no more complexity than well-known blocks of logic such as a register file or a priority encoder.

An implementation of an MPS design was outlined, including a method for pipelining the design. The design was extended to schedule instructions speculatively across basic-block boundaries. The performance of MPS was evaluated for basic-block and speculative scheduling using simulations with a realistic machine model. Two branch predictors were modeled for the speculative MPS designs, a forward-taken heuristic and a profile-bit. The requirements of instruction cache design for an MPS-based machine were discussed. An instruction cache for such a machine is labeled a schedule cache (SC), since entire schedules of instructions are stored in the cache. Two designs for schedule caches were

described, the uncompressed SC and the compressed SC. The performance of a compressed SC was measured. It was found that performance varies greatly between 32KB and 64KB SCs, indicating that MPS places additional burdens not present in traditional i-caches. Comparisons between an MPS design and a superscalar design were also made. The performance of neither design proved to be clearly superior. However, the results to date suggest that an MPS-based design provides a viable alternative to traditional out-of-order superscalars, especially when the likely reduction in cycle time is taken into account.

Acknowledgments

This work was supported by the Intel corporation and the National Science Foundation under grants MIP-9696010 and MIP-9625007. The authors would like to express their gratitude to the University of Illinois IMPACT group for the use of the IMPACT compiler system.

References

[1] David B. Papworth, "Tuning the Pentium Pro microarchitecture," *IEEE Micro*, vol. 16, no. 2, pp. 8–15, Apr. 1996.

[2] L. Gwennap, "PA-8000 combines complexity and speed," *Microprocessor Report*, vol. 8, no. 15, Nov. 1994.

[3] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, no. 7, pp. 478–490, July 1981.

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

[5] Manoj Franklin and Mark Smotherman, "A fill-unit approach to multiple instruction issue," In *Proc. 27th Ann. Int'l Symp. Microarchitecture* [25], pp. 162–171.

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

[7] Mark Smotherman and Manoj Franklin, "Improving CISC Instruction Decoding Performance Using a Fill Unit," in *Proc. 28th Ann. Int'l Symp. Microarchitecture*, Ann Arbor, MI, Dec. 1995, pp. 313–323.

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

[9] Eric Rotenberg, Steve Bennet, and Jim Smith, "Trace Cache: a low latency approach to high bandwidth instruction fetching," In *Proc. 29th Ann. Int'l Symp. Microarchitecture* [26].

[10] Ravi Nair and Martin E. Hopkins, "Exploting instruction level parallelism in processors by caching scheduled groups," in *Proc. 24nd Ann. Int'l Symp. Computer Architecture*, Denver, CO, June 1997.

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

[12] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," In *Proc. 27th Ann. Int'l Symp. Microarchitecture* [25].

[13] J. E. Smith and A. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *Proc. 12th Ann. Int'l Symp. 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, MA, Oct. 1992, pp. 248–259.

[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, no. 4, pp. 376–408, Nov. 1993.

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

[17] S. Weiss and J. E. Smith, "Instruction issue logic for pipelined supercomputers," *IEEE Trans. Comput.*, vol. C-33, no. 11, pp. 1013–1022, Nov. 1984.

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

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

[20] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, Toronto, Canada, May 1991, pp. 266–275.

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

[22] 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. Int'l Symp. Microarchitecture* [26].

[23] Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, and Burzin A. Patel, "Optimization of instruction fetch mechanisms for high issue rates," in *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, Santa Margherita Ligure, Italy, May 1995, pp. 333–344.

[24] Sanjeev Banerjia, Kishore N. Menezes, and Thomas M. Conte, "NextPC computation for a banked instruction cache for a VLIW architecture with a compressed encoding," Technical report, Dept. of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695-7911, June 1996.

[25] *Proc. 27th Ann. Int'l Symp. Microarchitecture*, San Jose, CA, Dec. 1994.

[26] *Proc. 29th Ann. Int'l Symp. Microarchitecture*, Paris, France, Dec. 1996.

Appendix

I. Algorithm for miss-path scheduling

II. A pipelined implementation of a miss path scheduler

The steps of the MPS algorithm are listed to make the requirements of the scheduling logic explicit. As explained in Section IV, these steps can be accomplished in a four stage pipeline.

```
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).
}
```

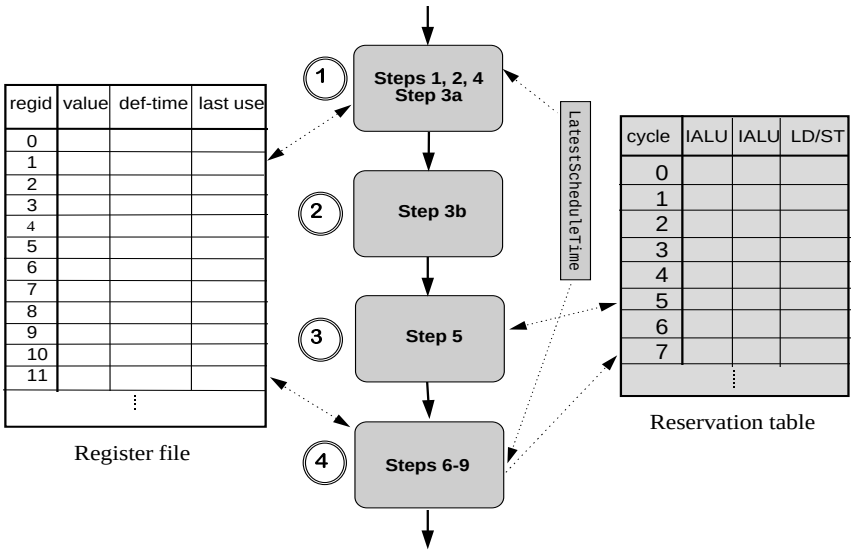

Fig. 10. Pipeline stages for miss path scheduling.

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. 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.
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.
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`.