Exploiting Partial Operand Knowledge

Brian R. Mestan
IBM Microelectronics
IBM Corporation - Austin, TX
bmestan@us.ibm.com

Mikko H. Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
mikko@ece.wisc.edu

Abstract

Conventional microprocessor designs treat register operands as atomic units. In such designs, no portion of an operand may be consumed until the entire operand has been produced. In practice, logic circuits and arithmetic units that generate some portion of an operand in advance of the remaining portions are both feasible and desirable, and have been employed in several existing designs. This paper examines existing and new approaches for exploiting early partial knowledge of an instruction's input operands for overlapping the execution of dependent instructions and resolving unknown dependences.

In particular, we study three applications of partial operand knowledge: disambiguating loads from earlier stores, performing partial tag matching in set-associative caches, and resolving mispredicted conditional branches. We find that each of these is feasible with partial input operands. With the goal of fully exploiting this characteristic, we propose and evaluate a bit-sliced microarchitecture that decomposes a processor's data path into 16- and 8-bit slices. We find that a bit-slice design using two 16-bit slices achieves IPC within 1% of an ideal design and attains a 16% speed-up over a conventional pipelined design not using partial operands.

1. Introduction

The degree of pipelining utilized in current microprocessor implementations has sharply increased over previous generation designs. Recent proposals have been made to further increase pipeline depth, and this trend is likely to continue as designers pursue higher clock frequencies [8,10,19]. Decode, issue, and register file logic that was able to evaluate in a single cycle in the past, now is typically divided across several cycles in order to meet aggressive frequency goals. We observe, accordingly, that as clock frequency increases, the number of cascaded logic stages able to evaluate in a single cycle decreases. Traditionally, the number of logic stages needed to produce a complete 32- or 64-bit result in the execution stage, whether that be the evaluation of an adder or address generation for a primary data cache access, has been one limiter on clock frequency [17]. Pipelining the execution stage can help enable a higher clock frequency, however, it can negatively impact performance much more so than deeper pipelining in the front-end of a design since the extra stages lengthen the scheduler loop between dependent instructions [2,15]. Furthermore, additional execution stages for address or condition flag generation can delay the resolution of various types of pipeline hazards, including read-after-write (RAW) hazards for load and store instructions, control hazards for mispredicted branch instructions, and hit/miss detection for cache accesses.

The decrease in performance is in essence a result of dependent or potentially dependent instructions not being able to benefit from the increased throughput of the pipeline since they still observe the end-to-end latency of an earlier instruction's entire execution stage. This reduction in throughput negates the effects of an increase in clock frequency. Nevertheless, the continuing demand for increased frequency makes pipelining of the execute stage appear inevitable. Solutions that focus on particular computations only, such as redundant representations that can avoid carry-propagation delays for arithmetic operations [4], can mitigate this problem. However, a more general solution that also avoids the conversion problems caused by redundant representations appears worthy of consideration.

In this paper, we propose such a design, which mitigates the effect that deeper pipelining has on dependent operations by shortening the effective length of dependency loops. The key observation we exploit is that dependent instructions can often begin their execution without entire knowledge of their operands, and that *partial* operand knowledge can be used to guide their execution. This exposes concurrency between dependent instructions allowing their execution to be overlapped in a pipelined design. We show that partial operand knowledge can not only speed up simple ALU dependency chains, as studied briefly in the past and implemented in the Intel Pentium 4 [11], but that when treated more generally, it can be used throughout a processor core to expose greater concurrency. In particular, we demonstrate that the following operations can proceed with only portions of their input register operands: disambiguating loads from earlier stores, accessing set-associative caches, and resolving mispredicted conditional branches. With the goal of fully exploiting these techniques, we propose and evaluate a bit-slice-pipelined design that decomposes a processor's data path into 16- and 8-bit slices. In a bit-slice design, register operands are no longer treated as atomic units; instead, we divide them into *slices*, which are used to independently compute portions of an instruction's full-width result.

2. Partial Operand Knowledge

The data flow of a program is communicated through register operands that are managed as atomic units. In doing so, scheduling logic assumes that all bits of a register are generated in parallel and are of equal importance. As pipeline stages are added to the execution of an instruction, this assumption may no be longer valid. In designs which pipeline the execution stage certain bits of a result are produced before others, and by exposing this knowledge to the scheduler it may be possible for dependent instructions to begin useful work while their producers remain in execution. We refer to the partial results produced during an instruction's pipelined execution as *partial operand knowledge*.

Conceptually, if we treat each bit of an operand as an independent unit, a dependent instruction can begin its execution as soon as a single bit of each of its operands has been computed. In this manner, dependent operations are chained to their producers, similar to vector chaining in vector processors [9]. Since functional units are designed to compute groups of bits in parallel (referred to as slices), it is more efficient to chain together slices of instructions. This abstraction fits well into a pipeline implementation since portions of a result are naturally produced before others as an instruction proceeds in its execution. Figure 1 presents a high-level overview of pipelined execution using partial operand knowledge. Conventional pipelining in the execution stage can lead to decreased IPC if partial results are not exposed since dependent instructions do not benefit from the increased throughput of the pipeline. When partial operand knowledge is exposed, as shown in (c), portions of a dependency chain can be overlapped.

3. Partial Operand Bypassing

Recent designs have exploited partial operand knowledge exclusively through the technique of *partial operand bypassing*. In these designs, rather than waiting for an entire result to be produced in execution, partial results are forwarded to consuming instructions. The TIDBITS design was one of the first to demonstrate that integer instructions did not have to wait for their entire operands to be produced before beginning execution [12]. In this design, a 32-bit adder is pipelined into four 8-bit adders, each writing its result into an 8-bit slice of the global register file. Dependency chains of simple integer instructions are

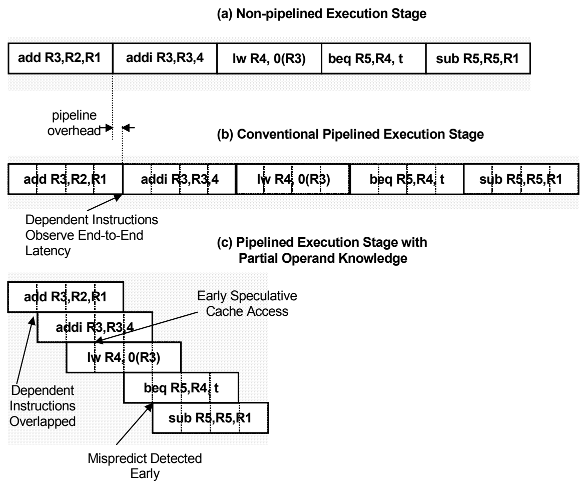

(a) Non-pipelined Execution Stage
add R3,R2,R1
addi R3,R3,4
lw R4, 0(R3)
beq R5,R4, t
sub R5,R5,R1
pipeline overhead
(b) Conventional Pipelined Execution Stage
add R3,R2,R1
addi R3,R3,4
lw R4, 0(R3)
beq R5,R4, t
sub R5,R5,R1
Dependent Instructions Observe End-to-End Latency
(c) Pipelined Execution Stage with Partial Operand Knowledge
add R3,R2,R1
addi R3,R3,4
lw R4, 0(R3)
beq R5,R4, t
sub R5,R5,R1
Early Speculative Cache Access
Dependent Instructions Overlapped
Mispredict Detected Early

FIGURE 1. Pipelining with Partial Operand Knowledge.

efficiently processed since each instruction only waits for the first 8 bits of its operands to become available before it is issued.

More recently, a similar design similar was implemented in the Intel Pentium 4 microprocessor. In the Pentium 4 simple integer instructions are issued to an ALU that is clocked at twice the frequency of the other pipeline stages [11]. This low-latency ALU is pipelined to produce the low-order 16-bits of a result in the first stage, which can then be bypassed to a dependent instruction in the next fast clock cycle. In this manner, the execution of two dependent instructions can be overlapped since dependences are resolved on 16-bit boundaries.

Similar techniques to partial operand bypassing are common for improving timing critical data paths in non-pipelined functional unit implementations. For example, in IBM's Star series microprocessors, the adder for effective address generation uses dual-rail dynamic logic to produce the low-order 24-bits faster than the remaining 40-bits (implemented in slower single-rail logic) in order to overlap the access to the TLB and level 1 data cache with the generation of the high-order address bits [1].

Partial operand bypassing is useful for efficiently processing long chains of simple integer instructions. However, other instruction types, such as loads and branches, traditionally require entire input operands to be available before execution. In the next sections, we show that opportunity exists for using partial operand knowledge to reduce the latency of these instructions as well.

4. Experimental Framework

In this study, we use a benchmark suite consisting of 11 programs randomly chosen from SPECint2000 and SPECint95. These are shown in Table 1 with their baseline

Table 1: Benchmark Programs Simulated

Benchmark	Simulated Instr (char / timer)	IPC	% Loads	Branch Accuracy
bzip	1 B / 500 M	1.29	33%	93%
gcc	1 B / 500 M	1.28	29%	90%
go	1 B / 500 M	1.20	22%	84%
gzip	1 B / 500 M	1.41	23%	93%
ijpeg	1 B / 500 M	2.13	18%	93%
li	1 B / 500 M	1.42	28%	95%
mcf	1 B / 500 M	1.42	22%	98%
parser	1 B / 500 M	1.00	40%	87%
twolf	1 B / 500M	1.40	36%	93%
vortex	1 B / 500 M	1.43	33%	89%
vpr	1 B / 500 M	1.81	28%	96%

Table 2: Machine Configuration

Out-of-order Execution	4-wide fetch/issue/commit, 64-entry RUU, 32-entry LSQ, speculative scheduling for loads, 15-stage pipeline, no speculative load-store disambiguation: load waits for prior store if store addr unknown in LSQ
Branch Prediction	64K-entry gshare, 8-entry RAS, 4-way 512-entry BTB
Memory System	L1 I$: 64KB (2-way, 64B line size), 1-cycle L1 D$: 64KB (4-way, 64B line size), 1-cycle L2 Unified: 1MB (4-way, 64B line size), 6-cycle Main Memory: 100-cycle latency
Functional Units	4 integer ALU's (1-cycle), 1 integer mult/div (3/20 -cycle), 4 floating-pt ALU's (2-cycle), 1 floating-pt mult/div/sqrt (4/12/24 -cycle)

characteristics in our simulation model. The benchmarks were compiled to the SimpleScalar PISA instruction set with optimization level `-O3`, and are run with the full reference input sets.

We use a trace driven simulator for our characterization work and a detailed execution driven model for timing analysis that are each modified versions of SimpleScalar [5] with machine parameters as shown in Table 2. We model a 15-stage out-of-order core similar to the pipeline used in the Intel Pentium 4 [11]. Our model supports speculative scheduling with selective recovery; instructions that are data dependent on loads are scheduled as if the load instruction hits in the level 1 cache, and then replayed if the load in fact misses.

5. Partial Operand Applications

We now propose and characterize three new applications for partial operand knowledge: disambiguating loads from earlier stores, performing partial tag matching in set-associative caches, and resolving mispredicted conditional branches. These three applications represent new opportunity for further condensing dependence chains.

5.1. Load-Store Disambiguation

For a load instruction to issue into the memory system its address must be compared to all outstanding stores to

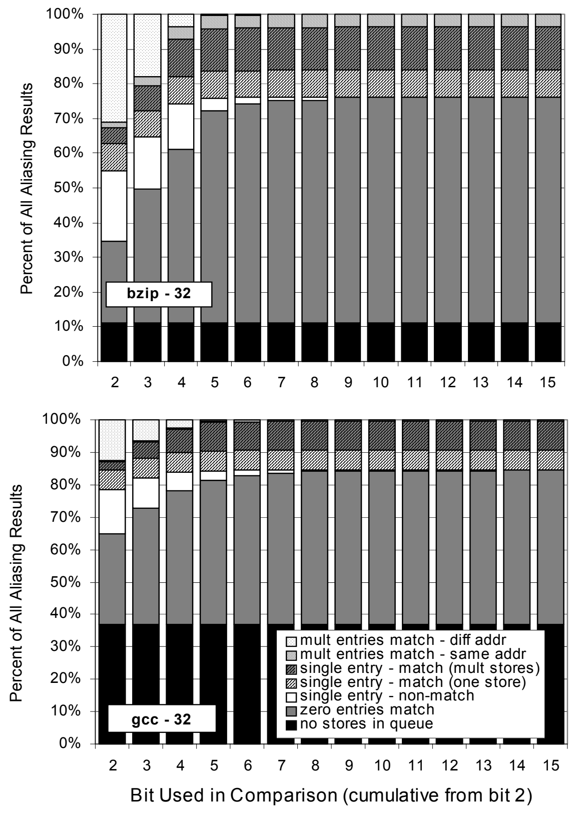


FIGURE 2. Early Load-Store Disambiguation Results. After examining the first 9 bits of the addresses in the LSQ, a unique forwarding address is found or all addresses are ruled out allowing a load to pass ahead of a prior store.

ensure no data dependency exists. Partial knowledge of the memory address can allow addresses in the load/store queue to be disambiguated before their address generation has fully completed. Furthermore, this disambiguation can proceed even before a virtual to physical translation has taken place by focusing solely on the index bits of the addresses.

Figure 2 characterizes how early a load address can be disambiguated against a store address in the load/store queue at the time a load is placed in the queue. We start from bit 2 and serially compare each bit of the load address to all prior stores in the queue. At each step, more bits are added to the comparison until we reach the 31st bit of the address, which represents the conventional comparison of the full addresses. The results are shown for two representative benchmarks, *bzip* and *gcc*, with a 32-entry unified LSQ.

There are five cases that occur as we compare the addresses: (1) zero entries in the LSQ match allowing the load to immediately be issued to the memory system; (2) a single entry is found, but as more bits are compared, this entry will actually not match; (3) a single entry is found, and when the entire address is compared this is an exact match of the load data address; (4) multiple entries match the load data address thus far; (5) multiple entries match the load data address thus far, but these multiple entries are all stores to the same address. (3) and (5) represent conditions in which the store should forward its data to the load instruction. In particular, in the case of (5), the store data should be taken from the latest entry in the queue that matched. To further enhance the characterization, we distinguish when there are no stores in the LSQ (this is a subset of the *zero entries match* case), and separate the *single entry-hit* case to show when we were able to disambiguate between multiple store addresses or just a single address when only one prior store is in the queue. The bars in Figure 2 converge to show the percent of time that a load address matches a prior store address in the LSQ. For this characterization we assume perfect knowledge of prior store addresses. If there is an unknown store address in the LSQ at the time the load enters, we determine its value first and place it in the appropriate category (2-5).

Given this characterization, a partial address comparison could be used for allowing a load to bypass a known store non-speculatively. After 9 bits have been compared, we have either (1) ruled out all prior stores due to a non-match in the low-order bits (*zero entries match* + *no stores in queue*), or (2) found a single store address in the queue which matches the address bits thus far (*single entry-match* + *mult-entries match-same addr*). In the case where a partial match is found, the load must wait until the entire address comparison is completed. However, notice that this address that partially matches ends up being an exact match of the load address when all bits are eventually compared since the *single entry-non-match* category has reached zero at this point. Therefore, we could speculatively forward the store data in this case with very high accuracy. Rather than using a partial comparison we could chose to speculate that the load does not match the prior store address. In this case, the *single entry-match* and *mult-entries match-same addr* categories represent the miss rate of this prediction. Using the partial information available can result in a much more accurate prediction.

The early load-store disambiguation technique enables a partially unknown load address to safely issue ahead of prior *known* store addresses. In this study, we do not allow a load to issue ahead of an *unknown* store address. Such optimizations have been studied in the past [7], and we note that they could be combined with early load-store disambiguation for further performance benefits.

5.2. Partial Tag Matching in Set-Associative Caches

One of the most performance-critical data paths in a microprocessor is the access to the level 1 data cache. Reducing the load-to-use latency can lead to higher performance since instructions that are data dependent on a load can be issued earlier and the load shadow can be shortened, resulting in fewer instructions being flushed on a mis-schedule [2]. Partial operand knowledge can be used to shorten the load-to-use latency by overlapping access of the level 1 data cache with effective address generation.

As an effective address is being computed, the low-order bits, which are naturally produced early in a fast adder circuit, can be used to index into the cache. If we consider a pipelined design, which generates a 32-bit address in two 16-bit adder stages, enough address bits will be available in the first stage to completely index into a large cache. Any bits that are available beyond the index can be used to perform a *partial tag match* to select a member speculatively, or signal a miss in the cache early non-speculatively.

Figure 3 shows an example of a partial tag cache access. After 16-bits of the effective address are generated, the exact index of the cache is known and 3 partial tag bits are available. These are used to perform a partial tag match to select a member in the selected equivalence class. In this case, we can immediately rule out the member in way 1

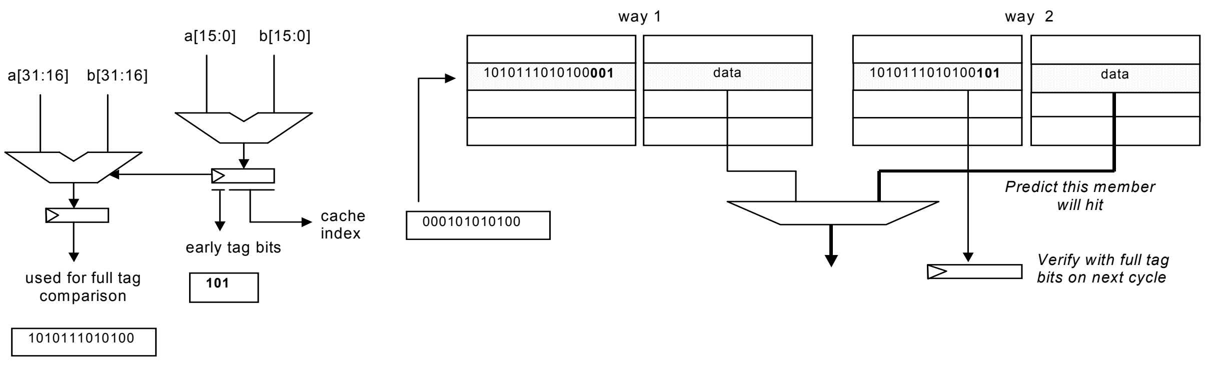


FIGURE 3. Partial Tag Cache Access.

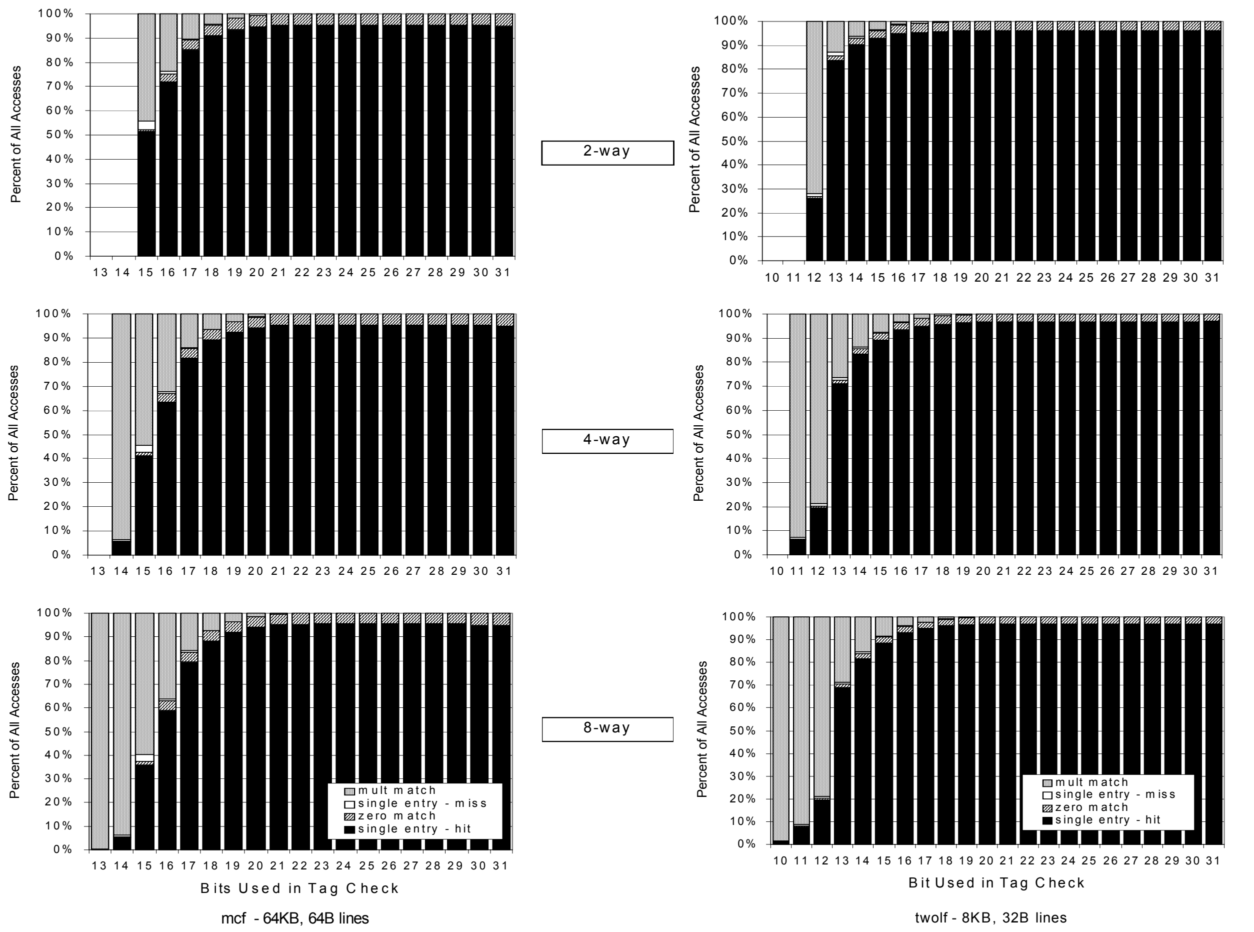


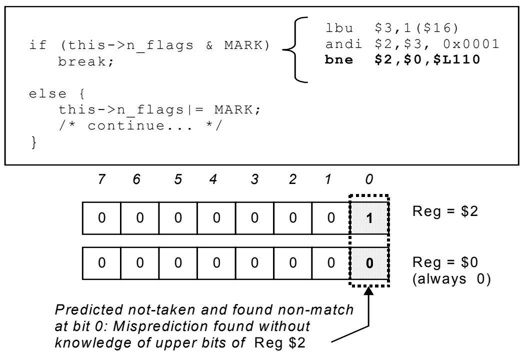
FIGURE 4. Partial Tag Matching Results. As more tag bits are used the graphs converge to the *single entry-hit* and *zero entries match* cases which represent the hit and miss rates of the cache respectively.

since its low-order tag bits do not match. Since the tag bits of the member in way 2 match, and the hit rates of most level 1 data caches are relatively high, we can speculate that this entry will indeed be a hit when the full tag bits are compared. This speculation allows the data to be returned before the address generation is completed, saving one cycle of load-to-use latency.

Partial tag matching has been explored in the past as an enabler of large associative caches [14], and as a method for reducing the access time of large cache arrays[16]. Our characterization is similar to that in [16] although their goal was to use partial tag matches even after full address generation has occurred. In our case, we use partial tag matching as a technique for allowing a cache access to be done in parallel with address generation. Sum-addressed caches take a different approach to reducing the load-to-use latency by performing the address calculation (base+offset) in the cache array decoder [18]. Partial tag matching and sum-addressed indexing are orthogonal, and both could be combined in a single design.

Figure 4 characterizes the number of bits of the tag needed in a set-associative cache to either find a unique member that matches the full address, or to signal a miss in the cache if no members match. The results are presented for two representative benchmarks, *mcf* and *twolf*. All of the benchmarks simulated had similar behavior. Two different cache sizes are shown (a 64KB, 64B line cache and a 8KB, 32B line cache) for three different associativities (2-way, 4-way, 8-way).

Tag bits are compared serially starting from the first tag bit available. Notice that as associativity grows, the tag bits start earlier in the address. At each step, more bits are added until all of the bits in the tag have been compared. This represents the conventional full tag comparison. As the address bits are compared, there are four cases that can occur: (1) a single entry matches the partial tag bits thus far, and this entry will match when the full tag bits are compared;



```
if (this->n_flags & MARK)        lbu   $3,1($16)
   break;                        andi  $2,$3, 0x0001
                                 bne   $2,$0,$L110
else {
   this->n_flags|= MARK;
   /* continue... */
}
```

7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 1
Reg = $2
0 0 0 0 0 0 0 0
Reg = $0 (always 0)
Predicted not-taken and found non-match at bit 0: Misprediction found without knowledge of upper bits of Reg $2

FIGURE 5. Example of Early Branch Misprediction Detection.

(2) a single entry matches the partial tag bits thus far, but this entry will not match when the full tag bits are compared; (3) zero entries match, revealing a miss in the cache; (4) multiple entries match the tag bits thus far, therefore a unique member cannot be determined. Cases (2) and (3) represent cache misses.

Ideally, we want the bars to converge early to the *single entry-hit* and *zero entries match* categories as they represent the hit rate and miss rate respectively. Notice that after 16 bits of the address have been generated (bit 15 in the figures), both the 64KB and 8KB caches still show a significant number of accesses that have multiple entries that match the tag bits thus far. However, most of these converge to the *single entry-hit* category. In other words, more importantly, the *single entry-miss* category is quite small at this point. Therefore, a policy such as Most-Recently-Used (MRU) could be used as a way-predictor to speculatively select one of the cache ways that match. This speculation would then be verified on the next clock cycle when the full address bits become available. Implementing such a way-predictor would reduce the load-to-use latency at the cost of modifying the load replay mechanism typical in most out-of-order processors to account for the cases in which the speculation was incorrect.

5.3. Early Resolution of Conditional Branches

In this section we characterize how early conditional branch mispredictions can be detected with the goal of reducing the effective length of the branch misprediction pipeline. The more stages a branch must pass through to verify a prediction, the more active wrong-path instructions enter the pipeline, and the longer the latency to redirect the fetch engine. We find that partial results can be used to overlap the redirection of fetch with the resolution of a branch.

An example of a branch that contributes to a significant amount (18%) of the mispredictions in the program *li* is shown in Figure 5. A majority of these mispredictions occur when the `bne` instruction (branch not equal to zero) is

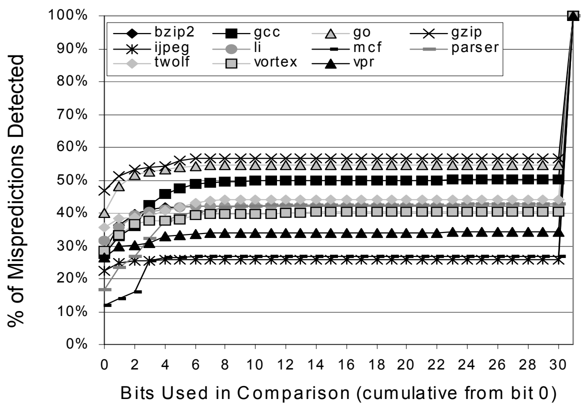


FIGURE 6. Early Branch Misprediction Detection Results. On average, 40% of all branch mispredictions can be detected after analyzing 8 bits of the branch comparison.

predicted as not-taken. In making this prediction, the processor speculates that register `$2` equals zero. Thus, when this misprediction is detected, the execute stage reveals that in fact register `$2` did not equal zero. Notice that the `andi` instruction feeding the branch clears all the bits of register `$2` except the low-order bit. Since the branch is compared against zero, as soon as a non-zero bit is detected the branch misprediction can be signaled to the front-end. In this case, the branch is entirely dependent on the status of the first bit in register `$2`.

In general, only a subset of conditional branch types can detect mispredictions early in our bit-sliced execution model. Branch types that perform a subtraction and test the sign-bit must wait for the full result to be produced. Furthermore, even though some branches, like the example shown in Figure 5, are capable of being detected early, this holds true only if the branch was originally predicted a specific direction. In our simulation model, we use the SimpleScalar PISA instruction set which has six conditional branch types: branch equal to zero (`beq`), branch not equal to zero (`bne`), branch less than or equal to zero (`blez`), branch greater than zero (`bgtz`), branch less than zero (`bltz`), and branch greater than or equal to zero (`bgez`). Of these six types only two (`beq`, `bne`) have the ability to be detected early since they do not require knowledge of the sign bit. However, `beq` and `bne` account for 61% of all dynamic branches and 48% of all mispredictions averaged across our benchmark suite.

In order to determine the effectiveness of using partial operand knowledge for resolving conditional branch instructions early, we characterize the number of bits needed to detect a misprediction using a very large 64k-entry gshare predictor. The results are shown in Figure 6. On average 40% of all conditional branch mispredictions can be resolved by examining only the first 8-bits of their operands. By examining the first bit in isolation, 28% of mispredictions can be detected on average. The large spike at bit position 31 is due to the need of the sign-bit for many branch types, and that some branches need all bit positions to determine that a misprediction occurred. For example, if a misprediction occurs when a `beq` instruction was predicted not-taken, in order to detect the misprediction we must show that the two registers feeding the branch are both equal. This requires all bits to be used in the branch comparison.

6. A Bit-Sliced Microarchitecture

Motivated by the results of the prior section, we propose a bit-sliced microarchitecture that directly exposes concurrency across operand bit slices and exploits this concurrency to pipeline execution of dependent instructions, accelerates load-store disambiguation, performs partial tag matching in the primary data cache, and resolves conditional branches early. The bit-sliced microarchitecture relaxes the conventional atomicity requirement of register operand reads and writes, instead enabling independent reads and writes to each partial operand, as delineated by bit-slice boundaries. Dependences are tracked and instruction scheduling operates at this finer level of granularity. In effect, we extend bit-slice pipelining of functional units to include the full data path and most major components of the control path. The proposed microarchitecture is illustrated in Figure 7. In this design, the issue queue and wake-up logic, register file, and functional units are each split into multiple units that work on a slice of the data path (16 bits if slicing by 2, 8 bits if slicing by 4). This is reminiscent of board-level ALU designs of the past that connect several bit-slice discrete parts together to compute a wider result.

In our bit-slice design, an instruction is divided into multiple slices at dispatch and placed into each slice's issue queue. In this study, we explore slicing by 2, in which an instruction's execution is divided into 2 stages each of which compute 16 bits, and slicing by 4, in which an instruction's execution is divided into 4 stages, each of which compute 8 bits at a time. This is similar to pipelining the execution stage into multiple stages in that instructions now take several cycles to execute. However, with bit-slice pipelining, dependences are resolved on slice boundaries, results are written into a slice of the global register file, and instruction slices can execute out of order. The high-order bit-slice of an instruction is allowed to execute before the low-order slice if no *inter-slice dependency* exists. Whereas conventional data dependences force the serial execution of a pair of instructions, inter-slice dependences force the serial execution of slices of an instruction.

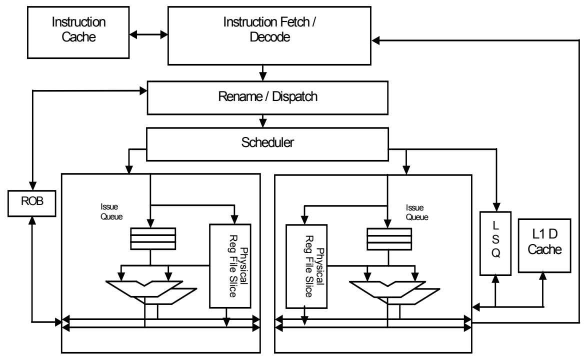


FIGURE 7. Bit-Sliced Microarchitecture.

Figure 8 shows how dependences are scheduled in a bit-slice pipeline when using 4 slices. An instruction dependent on RD must observe the dependency edges shown in each case. Case (a) in the figure, corresponds to a traditional pipelined ALU, in which a dependent instruction must wait until all slices of its operands have computed. In a bit-slice design, partial operand knowledge is exploited so that these dependences can be relaxed. Inter-slice dependences are only required when slices need to communicate with each other. For example, in arithmetic (b), the carry-out bit needs to be communicated across slices. This dependency is scheduled via an inter-slice dependence. Logic instructions (c), however, do not have any serial communication between slices and can execute out of order. Shift instructions require that more than just a single bit be communicated across slices. An example of the scheduling of slice dependences for a code segment from *vortex* is shown in Figure 9.

Not all instruction types easily fit into a bit-slice pipelined design. Prior work has shown that multiplication can proceed in a bit-serial fashion [13]. However, division and floating-point instructions require all bits to be produced before starting their execution. For these cases, a full 32-bit unit is needed. These units would collect slices of their operands and perform the computation once all slices have arrived. Our model accounts for all such difficult corner cases; however, they are not relevant to the performance of the applications we study.

Our bit-slice microarchitecture expands upon the integer ALU design used in TIDBITS [12], and is similar to the byte-serial and byte-parallel skewed microarchitecture targeted for low power presented in [6]. We focus on performance in this work. The low power optimizations proposed previously to exploit *narrow-width operands* could be used to enhance our design [3, 6]. For example, if an instruction is known to use narrow-width operands, inter-slice dependences could be relaxed further since the high-order register operand would be a known value of either all 0's or 1's. Such optimizations could be employed for both higher performance and lower power.

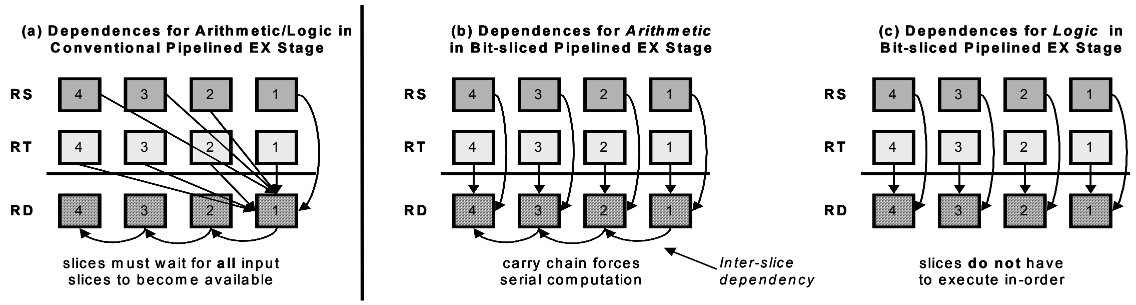
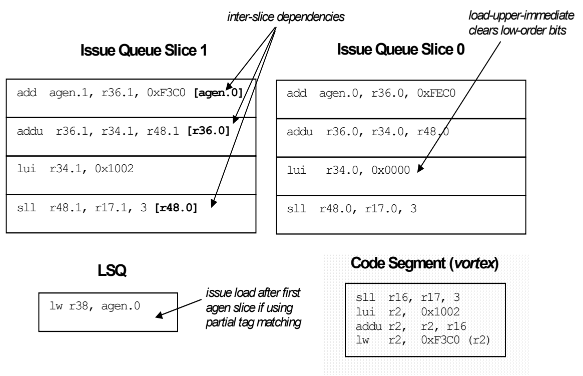

(a) Dependences for Arithmetic/Logic in Conventional Pipelined EX Stage
(b) Dependences for Arithmetic in Bit-sliced Pipelined EX Stage
(c) Dependences for Logic in Bit-sliced Pipelined EX Stage
RS
RT
RD
4
3
2
1
slices must wait for all input slices to become available
carry chain forces serial computation
Inter-slice dependency
slices do not have to execute in-order

FIGURE 8. Register Slice Dependences.


inter-slice dependencies
load-upper-immediate clears low-order bits
Issue Queue Slice 1
Issue Queue Slice 0
LSQ
issue load after first agen slice if using partial tag matching
Code Segment (vortex)

```
add   agen.1, r36.1, 0xF3C0 [agen.0]
addu  r36.1, r34.1, r48.1 [r36.0]
lui   r34.1, 0x1002
sll   r48.1, r17.1, 3 [r48.0]
```

```
add   agen.0, r36.0, 0xFEC0
addu  r36.0, r34.0, r48.0
lui   r34.0, 0x0000
sll   r48.0, r17.0, 3
```

```
lw r38, agen.0
```

```
sll  r16, r17, 3
lui  r2,  0x1002
addu r2,  r2, r16
lw   r2,  0xF3C0 (r2)
```

FIGURE 9. Issue Queue Example for Slice by 2 Configuration.

7. Implementation and Evaluation

In this section, we present an implementation of a bit-slice microarchitecture and evaluate its performance against a best-case design that does not pipeline its functional units yet runs at the same clock frequency. We study both the *slice by 2* and *slice by 4* configurations. Our machine model is the same as described earlier in Section 4, but the selective recovery mechanism is extended to replay loads that were incorrectly matched in the data cache as a result of partial tag matching. We use an MRU policy for way prediction to select an equivalence class member when multiple entries match the partial tag in the data cache. After 16 bits of an address are computed, we begin the cache index and partially match the virtual address tag bits. We assume a virtually indexed-virtually tagged cache, although this could be avoided by page coloring the low-order bits of the tag such that they do not need to go through address translation. In this case, when the full address is generated, the TLB would be accessed, and the physical address used to verify the partial tag match.

Since clock frequency is held constant in our study, slicing the functional units adds a full clock cycle of latency with each additional pipeline stage. This allows us to study the effect on IPC without assuming any increase in clock frequency due to the narrow-width functional units. Our goal is then to achieve an IPC comparable to that of a design that does not pipeline its functional units yet runs at the same clock frequency. Figure 10 summarizes the pipelines for the three configurations studied.

7.1. Performance Results

The IPC results for both slice configurations are shown below in Figure 11. The thin bars at the top of each IPC stack mark the base IPC of the benchmark when the execution stage is not pipelined; this is the IPC for an ideal machine. The bottom-most bar in the stack corresponds to the IPC attained with a standard pipelined execution stage that does not use partial operand bypassing or any of the partial operand knowledge techniques. Register operands are therefore treated as atomic units and dependences are resolved at the granularity of an entire operand, causing dependent instructions to observe the end-to-end latency of the execution stage. The results presented in the figure were obtained by running a series of simulations in which each optimization was applied one by one. Therefore, note that the order in which the optimizations were added matters to the impact shown for each specific optimization. Specifically, the optimizations added last benefit from optimizations added earlier.

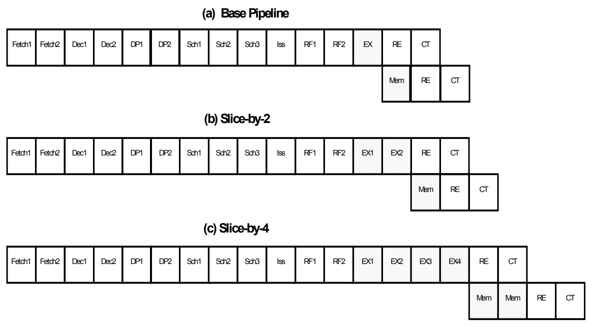


FIGURE 10. Simulated Pipeline Configurations. Frequency is held constant across configurations. The number of execution stages is increased by 2 and by 4. The performance goal is to recover the lost IPC that results from the increase in pipeline depth.

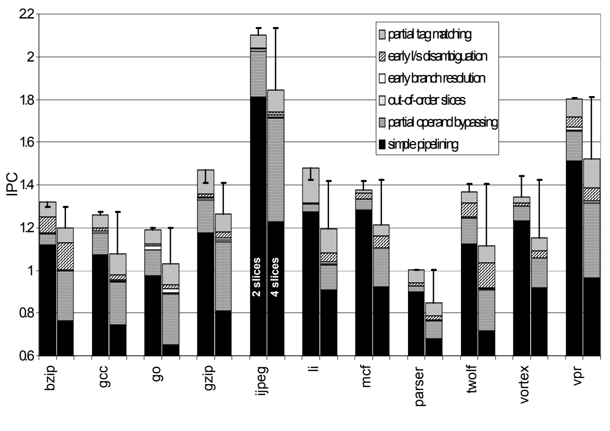


FIGURE 11. IPC Results for Bit-Sliced Microarchitecture. The thin bars at the top of each stack indicate the IPC of a best case design. The bottom-most bar of each stack corresponds to a simple pipeline that does not utilize any partial operand techniques.

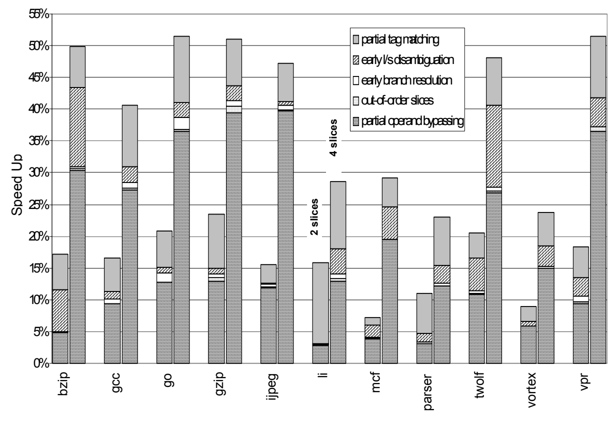


FIGURE 12. Speed-Up of Bit-Slice Pipelining over Simple Pipelining. On average, the new partial operand techniques proposed contribute an additional 8% (*slice by 2*) and 13% (*slice by 4*) speed up.

Figure 11 shows that if partial operand knowledge is exposed to dependent instructions, the IPC achievable approaches the best-case non-pipelined design. On average across the benchmarks simulated, when using 2 slices there is only a 0.01% slowdown compared to the ideal base machine. This is a 16% speedup compared to simple pipelining when no partial operand knowledge is utilized. In *bzip*, *gzip*, and *li*, the bit-slice design is able to exceed the IPC of the best case where the execution stage is still a single cycle. This slight improvement is due to second-order effects caused by wrong-path instructions, as well as increased scheduling freedom that can reduce the performance impact of structural hazards.

When using 4 slices, the bit-slice design has an 18% reduction in IPC on average compared to the best case model; this is a 44% speedup over simple pipelining. It is much harder to attain the base IPC in the *slice by 4* case since the execution latency of all single-cycle integer instructions is increased to 4 cycles. Note that in our simulation model, when slicing by 4 we also increase the cache access time for the level 1 cache to be 2 cycles. Although the execution latencies are 4 times that in the base model, the bit-slice design is able to recover a significant amount of the IPC by utilizing partial operand techniques. A bit-slice design is likely to support a much higher clock frequency than a standard pipeline since fewer cascaded logic stages are needed per pipeline stage now that only partial results are computed each cycle. Of course, other stages may need to be balanced to this higher frequency.

A detailed view of the speed-up achieved with the bit-slice design over simple pipelining is shown in Figure 12. This shows the techniques that are able to recover the lost IPC due to the longer execution pipeline. The existing technique of partial operand bypassing provides roughly half of the benefit for most benchmarks. However, the additional techniques described in Section 5 provide substantial additional benefit, and should be considered for future designs. Specifically, partial tag matching accounts for much of the speed-up over simple pipelining. The simulated L1 data cache size is 64KB, 4-way, which leaves only two bits beyond the index when the first 16-bits of the address are used for partial tag matching. Although just two bits are used, we found the accuracy of partial tag matching to be very high. There is only a 2% miss rate on average across our benchmarks for the slice by 2 configuration, and a 1% miss rate for slice by 4. Thus, while there are often multiple entries that match these two partial tag bits, the way-predictor (with MRU selection policy) is able to find the correct member in the cache.

In summary, partial operand knowledge can be used to recover much of the IPC loss due to deeper pipelining in the execution stage. It is important that a bit-sliced pipeline expose partial results to all instructions, and not simply to integer dependence chains as in previous designs. Early load-store disambiguation, partial tag matching, early branch resolution, and out-of-order slice execution can lead to an additional 8% and 13% speedup in IPC on average when slicing by 2 and 4 respectively. Since a bit-slice design only computes a portion of a result in a clock cycle, we believe execution units will be able to utilize a higher clock frequency. If clock frequency is instead held constant when moving to a bit-sliced design, the reduction in logic per pipeline stage can help ease critical path constraints by distributing these paths across several cycles while still allowing back-to-back execution of dependent instructions.

8. Conclusion

This paper revisits the concept of partial operand knowledge by relaxing the atomicity of register operand reads and writes. In effect, this eliminates the need to perform a pipeline's execute stage atomically. We extend the previously proposed technique of partial operand bypassing, utilized by proposed and existing designs, to enable three new applications: disambiguating loads from earlier stores, performing partial tag matching in set-associative caches, and resolving mispredicted conditional branch instructions. We propose and evaluate a bit-slice microarchitecture which divides atomic register operands into slices and exploits partial operand knowledge for exposing concurrency between dependent instructions. A bit-slice design is able to recover much of the IPC loss that results from pipelining the execution stage of a microprocessor. Our simulation results show that naive pipelining of the execution stage can lead to dramatic reduction in IPC; however, existing techniques as well as the new ones we propose can recover much if not all of this performance loss.

9. Acknowledgements

This work was supported in part by the National Science Foundation with grants CCR-0073440, CCR-0083126, EIA-0103670, and CCR-0133437, and generous financial support and equipment donations from IBM and Intel. We would also like to thank the anonymous reviewers for their many helpful comments.

References

[1] D. H. Allen, S. H. Dhong, H. P. Hofstee, J. Leenstra, K. J. Nowka, D. L. Stasiak, and D. F. Wendel. Custom Circuit Design as a Driver of Microprocessor Performance. *IBM Journal of Research & Development*, vol. 44, no. 6, November 2000.

[2] E. Borch, E. Tune, S. Manne, and J. Emer. Loose Loops Sink Chips, In *Proceedings of the 8th Annual International Symposium on High-Performance Computer Architecture*, February 2002.

[3] D. Brooks and M. Martonosi, Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance, In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, January 1999.

[4] M. D. Brown and Y. N. Patt. Using Internal Redundant Representations and Limited Bypass to Support Pipelined Adders and Register Files, In *Proceedings of the 8th Annual International Symposium on High-Performance Computer Architecture*, February 2002.

[5] D. C. Burger and T. M. Austin, The SimpleScalar Tool Set, Version 2.0, Technical Report CS-1342, Computer Sciences Dept., University of Wisconsin-Madison, 1997.

[6] R. Canal, A. Gonzalez, and J. E. Smith. Very Low Power Pipelines Using Significance Compression, In *Proceedings of the 33rd Annual Symposium on Microarchitecture*, December 2000.

[7] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic Memory Disambiguation using the Memory Conflict Buffer. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[8] A. Hartstein and T. R. Puzak. The Optimum Pipeline Depth for a Microprocessor, In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[9] J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach, Morgan Kaufman, San Mateo, CA, 1994.

[10] M. S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, P. Shivakumar, The Optimal Logic *Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays, In Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[11] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, P. Roussel. The Microarchitecture of the Pentium 4 Processor, *Intel Technology Journal Q1,* 2001.

[12] P. Y.-T. Hsu, J. T. Rahmeh, E. S. Davidson, and J. A. Abraham. TIDBITS: Speedup Via Time-Delay Bit-Slicing in ALU Design for VLSI Technology, In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, June 1985.

[13] P. Ienne and M. A. Viredaz. Bit-Serial Multipliers and Squarers, *IEEE Transactions on Computers*, 43 (12), December 1994.

[14] R. E. Kessler, R. Jooss, A. R. Lebeck, and M. D. Hill. Inexpensive Implementations of Set-Associativity, In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, June 1989.

[15] I. Kim and M. H. Lipasti. Implementing Optimizations at Decode Time, To Appear In *Proceedings of the 29th Annual Symposium on Computer Architecture*, June 2002.

[16] L. Liu. Cache Designs with Partial Address Matching, In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, December 1994.

[17] T. Liu and S.-L. L. Performance Improvement with Circuit-Level Speculation, In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, December 2000.

[18] W. L. Lynch, G. Lauterbach, J. I. Chamdani. Low Load Latency Through Sum-Addressed Memory (SAM), In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[19] E. Sprangle, D. Carmean, Increasing Processor Performance by Implementing Deeper Pipelines, In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.