ELIMINATING OPERAND READ LATENCY[1]

Larry Widigen Elliot Sowadsky Kevin McGrath

Advanced Micro Devices Inc.
California Microprocessor Division
1623 Buckeye Drive, Milpitas, CA. 95035-7423

{larry.widigen, elliot.sowadsky, kevin.mcgrath}@amd.com

Abstract

Programs generally exhibit load or memory operand read latencies that account for a significant portion of pipeline interlocks or stalls. In this paper we present an approach for the prediction of operand read data during the instruction fetch stage of a pipelined processor. For the X86 programs studied many have a significant percentage of such operand data that can be predicted with a high accuracy.

1 Introduction

The ability to increase throughput via pipelining is limited by pipeline hazards which may be caused by resource or data dependencies. Data hazards can be minimized by either reducing the latencies of functional units or hiding their effects. The hiding technique is the method used when out of order execution allows otherwise unused pipeline stages or functional units to be used by independent instructions. Latency reduction is the approach which data caches support.

One approach called fast address calculation [APS95] works to reduce load latency by allowing effective address calculation to proceed in parallel with cache access. [GM93] describe a cache accessed by instruction address containing operand address and 'stride' values for computing operand addresses in order to allow early cache access for load instructions.

[P94] discusses the technique of compressing operand address traces by maintaining an initial value of the operand's address together with offset values that show how the address changes. He notes that ". . . in a trace of 4 billion references, the actual number of unique addresses may be less than 10,000." Taking this observation as a starting point we discovered that for many instructions the described offset value for an operand's address is zero and is also invariant over a number of accesses. This suggested that such a complete address can be retrieved from a special dedicated cache without further requirements such as for a specialized addition.

Further investigation revealed that for many instructions the read operand data are also similarly invariant over a number of accesses and can be similarly cached. This allows the read latency for an out of order machine to be completely eliminated for a significant number of instructions on a speculative basis.

In this paper we present a cacheing approach that eliminates many memory operand read latencies with a high probability of correctness.

[1]Advanced Micro Devices Inc. has filed for a patent covering the ideas presented in this paper.

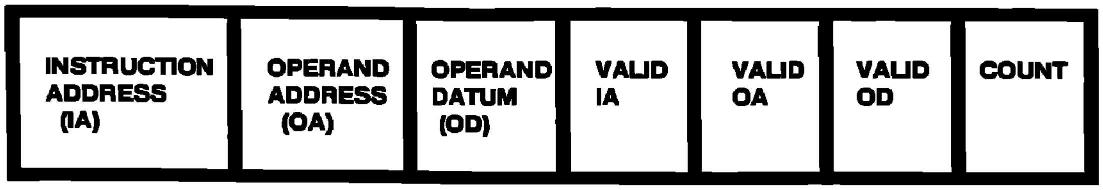

Figure 1. OPC Entry Format

2 Operand Prefetch Cache

The Operand Prefetch Cache (OPC) consists of a first fully associative Content Addressable Memory (CAM) structure that holds the address of an instruction which makes a single memory read access, a second fully associative CAM structure that holds the address of the operand associated with the instruction, and a data RAM structure that holds the associated operand's datum and control fields. Figure 1 shows the fields of an entry in the OPC.

The OPC will predict the operand memory read results for non-stack references when the address of the instruction matches an entry's IA field, all three Valid bits are set, and the COUNT field exceeds a specific limit. The COUNT field provides a criterion of correctness in much the same manner that a saturating counter does for branch prediction. When these conditions have all been met the OPC entry is called a HIT in the OPC. Stack references have been excluded because the data on a stack are typically quite volatile. (This is easily done for the X86 architecture since there is a dedicated stack pointer register and stack instructions are readily identifiable.) Only instructions with a single memory operand were considered in the study.

When a HIT occurs the Operand Address (OA) and Operand Datum (OD) fields are read from the OPC for the hit entry. Then the OA is passed to an Address Generation Unit to be validated when the address is generated for the operand of this instruction. The OD is passed to an Execution Unit to be used to execute the instruction and again to validate the OD against the operand datum returned from the cache lookup.

When the OA and OD respectively match the generated operand address and retrieved operand datum the COUNT field is incremented with a saturating adder. Should either or both fields not match, the mismatched field is updated with the correct value and the COUNT field is decremented with the saturating adder.

Making a new entry in the OPC occurs when an instruction that will make a memory read access does not have an entry whose IA field matches the instruction's prefetch address. In our study an OPC location is selected for a new entry by choosing any location where the Valid IA bit is clear or by choosing the location whose COUNT field is the smallest. Other algorithms (such as random replacement) would be less costly to implement, but they were not studied.

The selected location has the Valid OA and Valid OD bits cleared, the Valid IA bit set, and the prefetch address written into the IA field. Later when the AP generates the operand address it is written into the OA field, the Valid OA bit is set, and when the MCS returns the operand read datum that datum is written into the OD field, the Valid OD bit is set, and the COUNT field is set to an initial value (such as zero).

3 Experimental Evaluation

The effectiveness of this approach was evaluated by examining how well correct operand data are predicted for 12 programs.

3.1 Trace Extraction Methodology

Trace data were extracted using NexTracer for actual PC applications executing under Microsoft's Windows 3.1™ as part of Ziff-Davis' Winstone™ test suite. NexTracer is an engineering model Nx586™ processor that was modified so that each instruction was trapped before its execution to a special address space called hyperspace where hypercode was executed to extract relevant data concerning the instruction and its operand(s). Hypercode uses a superset of the X86 instruction set that permits it to access all X86 registers plus many Nx586 implementation specific registers. In addition to collecting available data, hypercode computes the operand addresses of the instruction for inclusion in the trace data. Table 1 lists the data extracted for each instruction. The extracted data was stored in a 160 MB trace buffer in hyperspace. Following the extraction of data hypercode would return the processor to the start of the instuction in the original address space and then have the processor execute it before again trapping the start of the next instruction.

Table 1. Extracted Data

Register Set
  EAX, EBX, ECX, EDX,
  ESP, EBP, ESI, EDI
  EFLAGS, EIP

Control Registers
  CR0, CR2, CR3
  Internal (including Bbit, Dbit, CPL)

Selector, base, and limit registers:
  CS, SS, GS, FS, DS,
  ES, TR, LDTR, GDTR

Instruction bytes

Operand Address, size, and type

3.2 Benchmark Characteristics

Traces were generally about 5 million instructions in length. Statistics were not gathered until after processing the first million instructions by the model in order to remove transient startup effects. Table 2 lists the programs traced.

Table 2. Programs Traced

Program	Description
alpage	Aldus's PageMaker™
clfmak	Claris's FileMaker™
crdraw	Corel Systems' CorelDraw™
lotus	Lotus's 123™
msacc	Microsoft's Access™
msexcel	Microsoft's Excel™
msword	Microsoft's Word™
mswrk	Microsoft's Works™
paradox	Borland's Paradox™
pwrpnt	Microsoft's Powerpoint™
quattro	Borland's Quattro Pro™
wdpfct	WordPerfect's WordPerfect™

3.3 The Model

The OPC was modeled in order to evaluate the benchmark traces for various cache sizes with a COUNT threshold of > 3. To simplify the effects of pipeline stage interactions some simplifying assumptions were made that bias the results in a conservative manner. The OPC is updated with write data when writes are sent to the MCS; but, in order to account for the possibility of a write to a prefetched operand being pending in one of the pipeline stages, any write to a prefetched operand within a 50 instruction window was considered to result in a miss for the prediction. An actual pipeline would typically be more forgiving in that it would be shorter or would have bypass logic to update a prefetch during instruction execution.

The cache was organized as 8-way set associative in order to provide an approximation of a fully associative organization while minimizing computation requirements.

The replacement algorithm modeled was to first use empty or invalid entries, then to re-use an entry based upon an aged COUNT. A second AGE counter was used for each entry. Whenever a successful prediction was made for an entry all other sets for that line each had their AGE incremented. The AGE counter for the set which had a successful prediction is then cleared. The replacement algorithm then becomes: to first use empty or invalid entries, then to select the set with the lowest COUNT value less the AGE count after shifting the AGE count right 6 bits. This algorithm was a rough attempt to balance the age of an entry versus its value as a predictor. Although considering age in this manner is not mandatory, thrashing of entries that predicted well was sometimes observed without it.

4 Results

Tables 3—5 show the results for the 12 programs. The first column is the percentage of correct predictions for those operand reads that were predicted, the second column is the percentage of operand reads that were predicted, and the third column is the percentage of mispredictions of all operand reads (predicted or not). This last column is important as it indicates how often the speculative execution must be aborted and re-executed with the correct operand datum.

For a 512 entry OPC Table 4 indicates an average 35.6% of the operand reads were predicted, and those prediction were correct 95.3% of the time.

Table 3 presents the results for a smaller cache and Table 5 for a larger cache.

5 Areas for Further Study

Further work in replacement algorithms and in selection criteria

Table 3. 32 X 8 OPC

%corr/ pred	%pred/ totrds	%mispr/ totrds	Program
96.04	28.42	1.13	alpage
93.74	21.79	1.37	clfmak
94.91	26.43	1.34	crdraw
97.92	41.55	0.86	lotus
89.31	27.79	2.97	msacc
93.43	20.40	1.34	msexcel
97.87	47.99	1.02	msword
92.74	22.50	1.63	mswrk
94.30	19.31	1.10	paradox
96.09	23.19	0.91	pwrpnt
98.84	45.86	0.53	quattro
95.35	31.42	1.46	wdpfct
95.05	29.72	1.31	AVERAGES

Table 4. 64 X 8 OPC

%corr/ pred	%pred/ totrds	%mispr/ totrds	Program
96.02	31.82	1.27	alpage
93.72	26.27	1.65	clfmak
95.06	28.60	1.41	crdraw
97.90	47.70	1.00	lotus
89.69	33.30	3.43	msacc
93.69	28.81	1.82	msexcel
98.46	68.06	1.05	msword
93.45	30.78	2.01	mswrk
94.67	24.21	1.29	paradox
96.07	27.37	1.07	pwrpnt
98.83	46.60	0.54	quattro
96.08	33.56	1.32	wdpfct
95.30	35.59	1.49	AVERAGES

Table 5. 128 X 8 OPC

%corr/ pred	%pred/ totrds	%mispr/ totrds	Program
96.19	35.99	1.37	alpage
93.91	30.36	1.85	clfmak
95.07	29.74	1.46	crdraw
98.09	51.22	0.98	lotus
90.46	38.23	3.65	msacc
94.05	35.47	2.11	msexcel
98.43	68.00	1.07	msword
93.83	38.18	2.36	mswrk
94.99	27.53	1.38	paradox
96.14	32.19	1.24	pwrpnt
98.73	47.33	0.60	quattro
96.05	34.98	1.38	wdpfct
95.50	39.10	1.62	AVERAGES

are areas that should receive additional study. In addition to variations of COUNT threshold, one might consider the global effect of not predicting once a misprediction occurs until a correct prediction would have been made.

Another question is whether RISC architectures would see the same benefits of the cache as are seen for the X86 architecture. It seems unlikely that modern architectures would benefit as much since they typically have a larger number of general registers available to reduce the number of memory read accesses.

A possible variation would be to relax the maintenance of near coherency with memory of the cache, possibly by using less than full address fields in order to reduce the size of the cache.

An important area to study would be to determine the impact of the OPC on performance in both in-order and out-of-order processors with various numbers of pipelines.

6 Conclusion

For the X86 architecture an Operand Prefetch Cache of approximately 12K bytes (64 x 8 organization) can achieve a high degree of correctness (over 95%) for operand predictions and such predictions occur for approximately 35% of the memory read operands of applications typically run on a personal computer. For those operands that are correctly predicted their memory read latency is eliminated.

REFERENCES

[APS95] Todd M. Austin, Dionisios N. Pnevmatikatos, and Gurindar S. Sohi, "Streamlining Data Cache Access with Fast Address Calculation," *Proceedings 22nd International Symposium on Computer Architecture,* June 22-24, 1995, pp. 369-380.

[GM93] Michael Golden and Trevor Mudge, "Hardware Support for Hiding Cache Latency," *U. of Michigan, EECS Dept., CSE-TR-152-93.*

[P94] Andrew R. Pleszkun, "Techniques for Compressing Program Address Traces," *MICRO-27,* 11/94, pp. 32-39.