Instruction Precomputation with Memoization for Fault Detection

Demid Borodin and B.H.H. (Ben) Juurlink
Delft University of Technology, The Netherlands
Tel: +31 15 2787362, Fax: +31 15 2784898. E-mail: {demid,benj}@ce.et.tudelft.nl

***Abstract*—Fault tolerance (FT) has become a major concern in computing systems. Instruction duplication has been proposed to verify application execution at run time. Two techniques, instruction memoization and precomputation, have been shown to improve the performance and fault coverage of duplication. This work shows that the combination of these two techniques is much more powerful than either one in isolation. In addition to performance, it improves the long-lasting transient and permanent fault coverage upon the memoization scheme. Compared to the precomputation scheme, it reduces the long-lasting transient and permanent fault coverage of 10.6% of the instructions, but covers 2.6 times as many instructions against shorter transient faults. On a system with 2 integer ALUs, the combined scheme reduces the performance degradation due to duplication by on average 27.3% and 22.2% compared to the precomputation and memoization-based techniques, respectively, with similar hardware requirements.**

I. Introduction

Fault tolerance (*FT*) is receiving an increasing attention. FT is often based on some form of expensive redundancy. While acceptable for critical computing systems, it should be avoided in commodity systems. This work addresses the error detection overhead minimization.

Franklin [1] proposed to duplicate instructions to detect errors in functional units (FUs) and (partly) in the dynamic scheduler of superscalar processors. Every decoded instruction is duplicated in the scheduler, and the results are compared after execution. This method incurs a significant performance penalty and covers mostly short transient faults in FUs. Unless redundant instructions happen to execute on different FUs, this scheme does not cover long-lasting transient and permanent faults. Ray et al. [2] also investigated a time-redundant duplication-based FT technique for superscalar processors.

Parashar et al. [3] improved the performance of duplication based on [2] with instruction memoization (*M*). M, also called instruction reuse, avoids redundant computations by reusing the results of previous executions, and is traditionally used in software. Hardware schemes utilizing M have also been proposed, such as [4]. Parashar et al. [3] avoid the execution of duplicate instructions by reusing previously computed results. Every executed instruction stores its input(s) and output in a special hardware buffer (which we call the *M-table*). Subsequent original instructions are always executed normally, while duplicates perform a M-table lookup and reuse the result, if available. The result of the original instruction is compared to the re-executed or reused result of the duplicate instruction.

This work was partially supported by the European Commission in the context of the SARC integrated project #27648 (FP6).

Borodin et al. [5] proposed to use the instruction precomputation (*P*) technique [6] to improve the duplication performance and fault coverage. P involves off-line application profiling. The profiling data (opcodes with input operands and results) is stored with the application binary code. Prior to execution, the profiling data is loaded into the *precomputation table* (*P-table*). The P-table is then used as the M-table in [3].

M as in [3] improves the fault coverage of duplication by covering more long-lasting transient faults. This is due to the time interval which M inserts between the first instruction execution (when the result is memoized) and the subsequent execution (when the result is compared to the memoized one). Long-lasting faults that disappeared or appeared during this interval, and thus did not affect both executions, are covered. P further improves the fault coverage of M by addressing all these faults, because the profiling is done at a different time and most likely even on a different system.

This work combines P and M, exploiting the advantages of each of them to achieve a better overall performance. P serves the globally dominant instructions, leaving more space in the M-table for the locally dominant instructions. Experimental results demonstrate that without duplication, P combined with M (*P+M*) most of the times outperforms both P and M used alone. With duplication (*D+P+M*), the advantage over duplication with M (*D+M*) and with P (*D+P*) is even larger, because duplication increases pressure on the FUs. D+P+M reduces the performance degradation of duplication with either P or M by on average 27.3% and 22.2%, respectively. The total hardware overhead is similar, because we use half-sized M- and P-tables for the P+M schemes. D+P+M slightly reduces the long-lasting fault coverage of D+P, because the P-table size halves, and thus fewer instructions hit it. In total, however, D+P+M protects more instructions by either P or M, improving the coverage of shorter transient faults. Compared to D+M, D+P+M improves the long-lasting and permanent fault coverage, because some instructions are covered by P, and the total instruction coverage is similar.

An additional paper contribution improves the P performance. This is achieved by resorting the instructions in the profiling data, and by changing the M- and P-table structure.

The remainder of this paper gives the organization details of the discussed systems in Section II, presents the experimental results in Section III, and draws conclusions in Section IV.

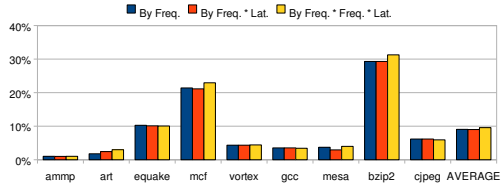

Fig. 1. Performance comparison of instructions sorting methods. Average IPC increase of different P configurations over the execution without P.

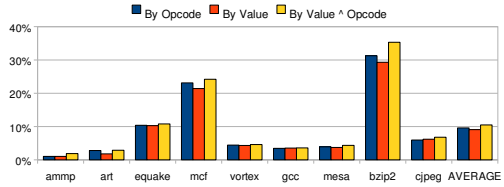

Fig. 2. Performance comparison of P-table indexing strategies: by opcode, by XORed operand values, and by XORed instruction operands and opcode.

II. System Organization

A. Instruction Duplication

The instruction duplication (*D*) scheme used in this work is based on [1], but executes every instruction in RUU twice rather than creating a copy. When both results are available, they are compared at the Writeback stage, and the instruction commits if no errors are detected.

All instructions except memory stores are duplicated. Stores consist of effective address calculation, which is duplicated, and the store itself, which is not duplicated, because it does not produce results that can be compared.

D reliably detects short transient faults in FUs. The further discussed variations exploiting M and P extend the D coverage, including long-lasting transient and permanent faults in FUs.

B. Precomputation and Memoization

Before execution, P loads the profiling data obtained off-line into the P-table. A large P-table is very expensive, but inefficient, because the advantage of storing rare instructions is negligible. The goal is therefore to fill a relatively small and cheap P-table with the most useful instructions.

M is similar to P, but instead of off-line profiling, it populates the M-table dynamically. The M-table is updated at the Writeback stage for every executed instruction, evicting old data if necessary. In our implementation, the Least Recently Used replacement policy is used for evictions.

The straightforward way to sort instructions in the P profiling data is by their frequency of occurrence. Then P-table contains the most frequent instructions. However, a single-cycle instruction with a slightly higher frequency than a multi-cycle instruction can be less useful in the P-table, because every hit will save fewer clock cycles. Thus, it is desirable to also take into account the instruction latency. Fig. 1 compares different sorting methods: frequency-only (by $freq.$), equal priority (by $freq. \times latency$), and frequency-priority (by $freq. \times freq. \times latency$). The bars depict the average IPC increase over the original execution (without P). The frequency-only and equal priority sorting methods never improve the IPC over 4.2% (cjpeg) more than the frequency-priority sorting does. Frequency-priority IPC improvement is by up to 67.3% (art) larger than that of frequency-only sorting, and by up to 35.5% (mesa) larger than equal priority. Thus, the frequency-priority sorting is used in this work.

The structure of the P-/M-table must provide a fast access time (one clock cycle in our organization). Structuring the table as a large array is not feasible, since it requires a sequential access to every instruction. In [5] a rather complex structure was used, which indexed the P-table by instruction opcodes, and made sure that more table space was available for more frequent opcodes. This provides a performance advantage (on average 8.2% more IPC increase, see Fig. 2) over indexing by operand values XORed with each other. The problem of the XORed operand values is that the P-table cannot hold different instructions with the same popular input values. This can be solved by XORing the result with the opcode (taken modulo the number of sets in the table), and thus assigning different P-table sets for these conflicting instructions. Fig. 2 shows that this method achieves better performance than opcode-based indexing. It improves the IPC increase by on average 15.4%, and is further used in this work both in M- and P-tables.

The cache-like M-/P-table consist of sets, each holding instructions with their input operands and output values. The number of instructions per set is defined by the table associativity. A P-table (M-table) with *N* sets and associativity equal *A* will be further referred to as *P(N,A)* (*M(N,A)*).

Only computational instructions (integer and FP) are reused by P and M. Stores cannot be verified, because they do not provide any output values. Loads cannot be reused, because the loaded value may differ at different times.

C. Duplication with Precomputation or Memoization

In D+M, instructions are executed at least once, and executed once more if they do not hit the M-table. The result is then compared to the reused or recomputed result. Successful instructions update the M-table and commit.

Unlike with M, with P the results are computed off-line, only once per application. If the profiling data is obtained on a reliable system and is protected well, it can provide the reliability of the profiling system on the target system. Protecting the P-table with ECC and some control logic redundancy is sufficient for that. The error correcting capability of the P-table would even bring a partial recovery capability into otherwise fail safe-only D-based system. Given a reliable profiling system and P-table, the execution of instructions that hit the P-table can be skipped without any reliability loss. In [5] this scheme was referred to as *duplication+precomputation only* (*D+PO*), and it was shown to significantly outperform D+P with original instruction re-execution (as in D+M). In this work only D+PO is used, and D+P actually refers to D+PO.

D. Precomputation Combined with Memoization

In P+M, P has priority over M. Instructions hitting the P-table are reused and are not written to the M-table. The M-table is reserved only for instructions missing the P-table.

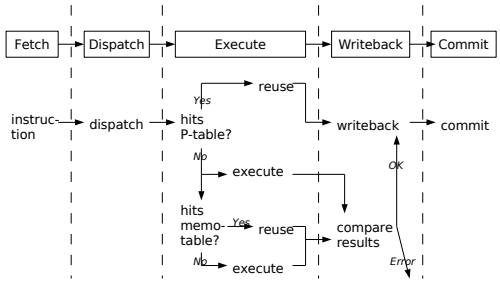

Fig. 3. D+P+M. M-table update at Writeback is skipped.

These instructions perform a M-table lookup. Instructions that hit it are reused, and the others are executed. This organization allows P to serve the globally dominant instructions, and leaves space in the M-table for the locally redundant instructions. For performance reasons, the P- and M-table lookups are performed in parallel.

Fig. 3 presents D+P+M. Instructions hitting the P-table are considered sufficiently reliable. They progress to Writeback and Commit without verification. Other instructions are always executed once and are either reused (from the M-table) or re-executed. The results are compared at the Writeback stage, and an error is signaled on mismatch.

In this work, when comparing P+M-based schemes with the P- and M-based schemes, the size of both the M- and P-table is always halved in P+M. In other words, the *P+M(N,A)* scheme has a M- and a P-table each with $\frac{N}{2}$ sets, of the associativity A. Thus, while *P(N,A)* has a single table with $N \times A$ entries, *P+M(N,A)* has two tables each of the size $\frac{N}{2} \times A$. This ensures that the hardware overhead is similar in the compared schemes.

III. Experimental Results

This section evaluates P+M and D+P+M. Several integer and floating-point benchmarks from the SPEC CPU2000 suite [7] are used, as well as the JPEG encoder. Different inputs are used for P at the profiling and deployment stages.

A. Simulation Platform

The *sim-outorder* simulator from the SimpleScalar tool set [8] has been modified to model the considered schemes. The processor has fetch/decode/issue width of 8, 1 to 4 integer ALUs, L1 data and instruction caches of 32 KB (2-way set associative), L2 unified cache of 512 KB (4-way), 128 instructions fit in the RUU. The benchmarks were run, skipping the first 100 M instructions to bypass the I/O overhead, either until completion or until the next 100 M instructions committed.

To keep the cost low, only small to modestly-sized M- and P-tables are examined. The number of sets in the tables varies from 8 to 1024, and associativity from 1 (direct mapped) to 4. This means that the total M-/P-table size varies from approximately 104 B to 52 KB (assuming that for every instruction the tables hold one 8-bit opcode and two 32-bit input and one output values, without protective information).

B. Fault Coverage

It is difficult to compare (quantitatively) the fault coverage of the considered D-based schemes. Therefore, two indirect methods are used to evaluate the fault coverage: the average M-table instruction lifetime and the table hit rate.

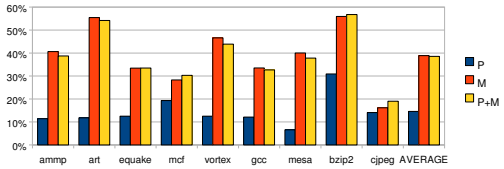

Fig. 4. Average hit rates in P, M and P+M for different table configurations.

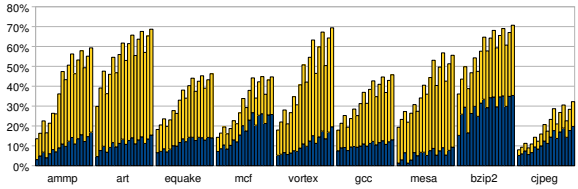

Fig. 5. P+M hit rates for different table configurations.

D, D+P, and D+M differ in the coverage of long-lasting transient and permanent faults. D+M improves the fault coverage of D by introducing a time gap between the execution of the memoized instruction and its subsequent (being verified) execution. This time period (measured by the number of clock cycles) is called the *M-table instruction lifetime* [5]. The larger the M-table instruction lifetime is, the more long-lasting faults are likely to appear or disappear during this time. These faults will be detected, because they affect only one of the compared execution results. Note that D is also likely to insert a certain time gap between the redundant instruction executions (a gap of a few clock cycles can be expected). The advantage of D+M over D depends on how much larger the M-table instruction lifetime is than the D gap. As shown in [5], the M-table instruction lifetime is comparable to the D gap for small M-table sizes, because the instructions are replaced quickly with these configurations. For larger M-tables, the instruction lifetimes of 1.4% to 28.2% of the total application execution time have been observed. Faults lasting longer (including permanent faults) are only covered by D+M if the compared results happen to be calculated on different FUs. D+P covers all long-lasting transient and permanent faults for instructions hitting the P-table, because the profiling data is computed at a different time and likely on a different host system.

D+P+M improves the long-lasting fault coverage of D+M, because some instructions are covered by P instead of M. D+P+M reduces the long-lasting fault coverage of D+P, because with the reduced P-table size, the number of instructions covered by P diminishes. The following hit rate evaluation measures the fault coverage gain and loss of D+P+M compared to D+P and D+M.

Fig. 4 shows the hit rate (average for different table configurations) of P, M and P+M. Fig. 5 presents the M-table (the upper part of every bar) and the P-table (the lower part of every bar) hit rates in the P+M scheme. The bars for every benchmark represent the following configurations from left to right: *(8,1)*, *(8,2)*, *(8,4)*, *(16,1)*, *(16,2)*, *(16,4)*, *(64,1)*, *(64,2)*, *(64,4)*, *(256,1)*, *(256,2)*, *(256,4)*, *(512,1)*, *(512,2)*, *(512,4)*, *(1024,1)*, *(1024,2)*, and *(1024,4)*.

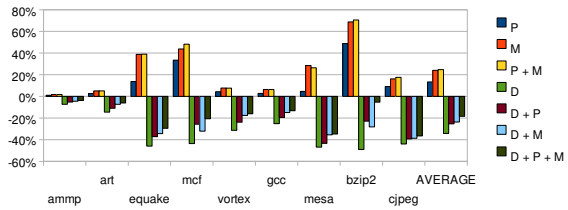

Fig. 6. IPC increase of different P, M, and duplication (D) schemes over the original. 2 integer ALUs. Average for different table configurations.

Fig. 4 shows that the average hit rate of P+M is only 0.9% less than the M hit rate. This means that approximately the same percent of instructions is protected by D+P+M and D+M. From Fig. 5 it follows that on average 36.1% of the instructions in P+M (and thus also in D+P+M) hit the P-table, and the rest hit the M-table. Thus, while about the same number of instructions is protected in D+P+M and D+M, the long-lasting fault coverage of 36.1% of them is improved by P in D+P+M. Fig. 4 also shows that P+M hit rate is on average 2.6 times higher than that of P. Thus, D+P+M protects significantly more executed instructions than D+P, by either P or M. However, D+P+M protects on average 10.6% less instructions by P than D+P does (the others are covered by M). These instructions have a reduced long-lasting fault coverage.

C. Performance

Fig. 6 compares the IPC increase of different schemes (average for different table configurations) over the non-redundant execution on a system with 2 integer ALUs. D degrades performance, and thus has a negative IPC increase. The shown IPC of the D-based schemes takes into account only the original instructions (not duplicates), and thus can be compared to the non-redundant schemes.

P+M for most benchmarks slightly outperforms M (the IPC increase improves by on average 2.6%), and significantly outperforms P. D+P+M always outperforms D+M and D+P, reduces the performance degradation due to D by on average 22.2% and 27.3%, respectively. We explain D+P+M advantage compared to P+M by the doubled FUs requirements of the D-based schemes. In addition, the fact that instructions hitting the P-table are not executed plays an important role: D+PO reduces the IPC by on average 9% less than D+P (which executes hitting instructions) does [5]. Moreover, in a few cases D+P+M even outperforms the original execution.

P and M improve performance when the number of FUs is a system bottleneck. Thus, the improvement decreases when the number of ALUs increases. Fig. 7 shows the average IPC increase (across all the configurations and benchmarks) for varying number of integer ALUs in the system. With 1 ALU D+P+M reduces the IPC decrease, compared to D, by 53.4%, with 2 ALUs by 46.1%, and with 4 ALUs by 22.2%.

Fig. 8 shows how the performance (across all the benchmarks) depends on the M- and/or P-table configuration. To explore the limits, Fig. 8 also includes a very large configuration with 64K sets and an associativity of 16 (corresponding to a table of approximately 13 MB). Fig. 8 demonstrates the importance of the associativity, which significantly improves

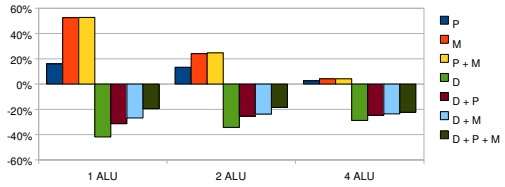

Fig. 7. IPC increase of different schemes with different number of integer ALUs. Average for different table configurations and benchmarks.

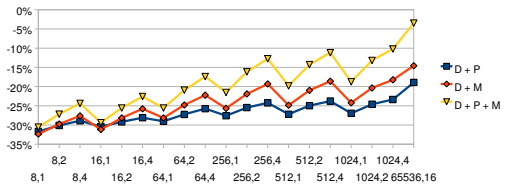

Fig. 8. IPC increase for different table configurations. Average for systems with 1, 2 and 4 integer ALUS.

performance for all the configurations. Fig. 8 also shows that the performance improvement diminishes when enlarging larger tables (with 256 and 512 sets) in comparison with the smaller ones. For example, *D+P+M(1024,4)* doubles the size of *D+P+M(512,4)*, while its IPC increase improves by less than 1%. Thus, *D+P+M(512,4)* might be a good candidate for the optimal D+P+M table configuration from the performance vs. hardware overhead point of view.

IV. Conclusions

This work binds the strengths of P and M to achieve better performance than any of them achieves alone. Applied together, P exempts M from the globally dominant instructions, saving space in the M-table for the locally dominant ones.

D+P+M covers approximately the same number of instructions as D+M, but introduces the permanent fault coverage of on average 36.1% of them. At the same time, D+P+M reduces the IPC penalty of D+M by on average 22.2%. Compared to D+P, D+P+M covers on average 2.6 times as many instructions, but on average 10.6% less instructions are covered against long-lasting and permanent faults. Given the significantly higher total instruction coverage and the performance degradation reduction of 27.3% over D+P, the long-lasting fault coverage loss of 10.6% seems to be acceptable.

References

[1] M. Franklin, "A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors," *DFT*, pp. 207–215, 1995.

[2] J. Ray *et al.*, "Dual Use of Superscalar Datapath for Transient Fault Detection and Recovery," *MICRO-34*, pp. 214–224, 2001.

[3] A. Parashar *et al.*, "A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy," in *ISCA-04*, 2004, pp. 376–386.

[4] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse," in *ISCA-97*, 1997, pp. 194–205.

[5] D. Borodin, B. Juurlink, and S. Kaxiras, "Instruction Precomputation for Fault Detection," in *DSD-2009*, 2009, pp. 91–99.

[6] J. Yi *et al.*, "Increasing Instruction-Level Parallelism with Instruction Precomputation," in *Euro-Par-02*, 2002, pp. 481–485.

[7] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[8] T. Austin *et al.*, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.