

Instruction Precomputation for Fault Detection

Demid Borodin and B.H.H. (Ben) Juurlink
Computer Engineering Laboratory
Delft University of Technology
Delft, The Netherlands
E-mail: {demid,benj}@ce.et.tudelft.nl

Stefanos Kaxiras
Department of Electrical and Computer Engineering
University of Patras
Patras, Greece
E-mail: kaxiras@ee.upatras.gr

Abstract—Fault tolerance (FT) is becoming increasingly important in computing systems. This work proposes and evaluates the instruction precomputation technique to detect hardware faults. Applications are profiled off-line, and the most frequent instruction instances with their operands and results are loaded into the precomputation table when executing. The precomputation-based error detection technique is used in conjunction with another method that duplicates all instructions and compares the results. In the precomputation-enabled version, whenever possible, the instruction compares its result with a precomputed value, rather than executing twice. Another precomputation-based scheme does not execute the precomputed instructions at all, assuming that precomputation provides sufficient reliability. Precomputation improves the fault coverage (including permanent and some other faults) and performance of the duplication method. The proposed method is compared to an instruction memoization-based technique. The performance improvements of the precomputation- and memoization-based schemes are comparable, while precomputation has a better long-lasting fault coverage and is considerably cheaper.

Keywords—error detection, fault detection, instruction precomputation, fault tolerance, reliability.

I. INTRODUCTION

The importance of fault tolerance (FT) of computing systems is increasing instantly nowadays [1]. This is a consequence of the technology trends which try to follow Moore’s law, increasing chip density by decreasing feature size. Smaller feature size, greater chip density, and minimal power consumption lead to increasing device vulnerability to external disturbances such as radiation, internal problems such as crosstalk, and other reliability problems.

To detect faults, this work proposes to use the *instruction precomputation* technique [2] (further referred to as *pre-computation* or *P*). *P* is a work reuse technique involving off-line application profiling. The profiling data (instruction opcodes with operands and corresponding results) is stored together with the application binary code. Prior to execution, the profiling data is loaded into the *precomputation table* (*P-table*). When an instruction is about to be executed, a *P-table* lookup is performed. In the original (performance-oriented)

P scheme [2], if the instruction with the same input operands is found, the result is reused and the instruction is not executed. In the proposed (FT-oriented) scheme, the instruction is still executed, and the computed result is compared to the result from the *P-table*. This corresponds to the classic duplication with comparison scheme [3]. Alternatively, if the profiling data and the *P-table* are sufficiently reliable, the precomputed result does not need to be re-computed.

P does not provide full fault coverage, because it does not cover all executed instructions. This work uses *P* to supplement another fault detection technique with the purpose of improving its performance and fault coverage. *P* is used in combination with the instruction duplication technique, which executes every instruction twice and compares the results. Whenever possible, instead of re-executing an instruction, its result is compared to the value in the *P-table*. This technique is further called *Duplication+Precomputation*, or *D+P*. *D+P* is compared to an instruction duplication-based technique enhanced by another work reuse technique called memoization (*M*), which is further referred to as *Duplication+Memoization* or *D+M*.

This paper is structured as follows. Section II introduces the instruction precomputation, memoization, and duplication techniques, along with related work. Section III gives the organization details of the discussed systems. Section IV presents experimental results. Finally, Section V draws conclusions and discusses future work.

II. BACKGROUND AND RELATED WORK

This section presents the background and related work on precomputation (Section II-A), its related technique memoization (Section II-B) which will be compared to precomputation, and FT (Section II-C).

A. Instruction Precomputation

P avoids the re-execution of the most frequent instructions by using profiling information collected off-line. It was proposed by Yi et al. [2] to increase instruction-level parallelism. The execution statistics of dynamic instructions are collected at application development time. By *dynamic instruction* we mean the instruction (represented by the opcode) with its unique combination of input operands and

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

the corresponding output result. At run-time, prior to the application execution, the profiling data is loaded into the P-table. Then, when possible, the results from the table are used instead of re-executing instructions. Since the size of the P-table is limited and not all the profiling data can be stored there, the dynamic instructions are sorted by frequency at the profiling stage. This ensures that the most frequent instructions appear in the table.

Another technique targeting redundant computations, instruction memoization, uses the same principle of keeping dynamic instruction results in a special hardware buffer (called the *memo-table*) to avoid re-execution. M differs from P in that instead of off-line profiling, it is dynamic: it uses results of recently executed dynamic instructions to populate the memo-table. The following section presents M in detail.

B. Memoization

Memoization (or memoing), also called instruction reuse, avoids redundant computations by reusing the result(s) of previous executions. The concept was introduced by Michie [4]. M is traditionally used in software, manually or automatically. For example, a programmer can manually reuse the results of a previous function invocation. A look-up table with the function return value(s) corresponding to specific sets of arguments is created at run-time, and is consulted on future invocations. Figure 1 illustrates a possible structure of such a function. Automatic M is

```
function ( arguments )
{
    if the result for the given arguments is available
        reuse the result and quit;
    perform computations;
    save the result and arguments in the look-up table;
}
```

Figure 1. Manual memoization example.

used, for example, for parsing in compiler technologies [5]. M can be applied at different levels. For example, at a processor functional unit (FU) level: the FU reuses its previous results when the inputs match. At higher levels, results of an instruction, a block of instructions, or a high-level programming language function can be reused. This work focuses on instruction-level M, which reuses the results of instructions with matching operand values.

Several works proposed hardware schemes utilizing M. Richardson [6] saved the results of non-trivial floating-point operations (with operands other than 0.0 and 1.0, for example) in a *result cache*. The cache is direct-mapped and indexed by hashed operand values. The result cache is accessed in parallel with an executing floating-point operation. If a hit occurs, the result is reused and the full operation is canceled to save time. Oberman and Flynn [7] addressed reducing the latency of the floating-point division operation. They proposed *division caches*, which are similar

to the result caches except that only the results of divisions are stored, and *reciprocal caches*. By saving the reciprocals of the divisors, the reciprocal caches convert the high-latency division operations to lower-latency multiplication operations. Sodany and Sohi [8] used M for all instructions rather than only for long-latency operations. To increase the benefit for single-cycle operations, they indexed the *reuse buffer* using the program counter (PC) instead of the instruction operand values. This way the reuse buffer can be accessed earlier in the pipeline, even when the operand values are not yet available. Citron et al. [9] used M for multimedia applications and focused only on long-latency instructions.

C. Fault Tolerance

To provide full fault coverage in the proposed scheme (not only of the precomputed instructions), the instruction duplication method [10] is adapted and extended with P support. The original method duplicates every decoded instruction in the dynamic scheduler of a superscalar processor. The instruction and its duplicate are then scheduled and executed in a regular way, and their results are compared. The scheme detects errors in the FUs, the internal buses between the dynamic scheduler and the FUs, and some errors in the dynamic scheduler itself. If redundant instructions execute on the same FU, this scheme in its pure form does not cover long-lasting transient, permanent, and common faults in time. If the copies are executed on different FUs, common faults can still affect both of them in the same way. By long-lasting transient faults we mean faults that are present longer than one clock cycle. By common faults we mean faults affecting two different FUs in the same way, or (in time) the same FU at different times.

Parashar et al. [11] proposed to use M (which they referred to as instruction reuse) to reduce the performance overhead introduced by instruction duplication with comparison. The instruction duplication scheme is based on [12]. When possible, the execution of the duplicate instructions is avoided by reusing a previously computed result. The original instructions are always executed normally, and their results are compared to the re-executed or reused results of the duplicate instructions. We propose a similar organization, using P instead of M. Later Gomaa and Vijaykumar [13] used M to verify the instructions that hit the memo-table, leaving other instructions unprotected or (when it does not degrade performance) protected by another fault detection technique.

D+M improves not only performance of the instruction duplication, but also its fault coverage, by covering more long-lasting and common faults. D+P further improves the fault coverage by addressing more long-lasting and even permanent faults. Section IV-B presents a detailed fault coverage evaluation.

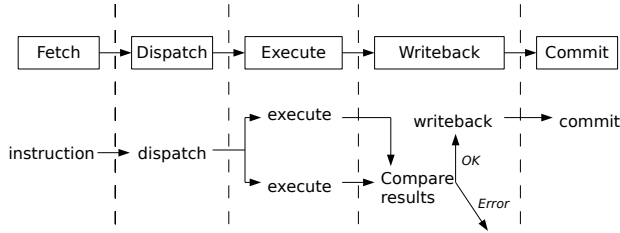


Figure 2. Instruction duplication

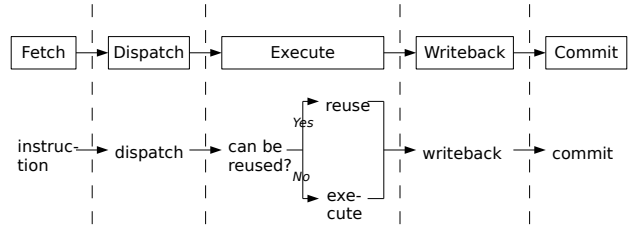


Figure 3. Instruction precomputation

III. SYSTEM ORGANIZATION

This section presents the implementation details of the discussed techniques: instruction duplication (Section III-A), P (Section III-B), M (Section III-C), and the final architecture combining instruction duplication and P or M (Section III-D).

A. Instruction Duplication

Figure 2 presents the instruction execution steps with the corresponding activities performed by an instruction. A fetched instruction proceeds normally until the execution stage. There it is issued to the corresponding FU(s) twice. The results are compared at the Writeback stage and an error is signaled on mismatch. Note that the original instruction stalls at the Writeback stage (stays in the Register Update Unit (RUU)) until its duplicate finishes execution.

What happens if the results do not match, depends on the goals of the system. A system targeting fail-safe operation would signal an error and halt. If FT is required, another copy of the questioned instruction could be created and executed, performing a majority voting on all the obtained results according to the Triple Modular Redundancy (TMR) scheme [3], or a different form of recovery could be initiated. This, however, is outside the scope of this work.

Note that in the considered implementation, the choice of the FUs used to execute the redundant instruction copies is only driven by resource availability. If the system has a single FU required to execute an instruction, both instruction copies will execute on the same FU. Moreover, even if multiple appropriate FUs are present, it can happen that the redundant copies still execute on the same FU. This means that permanent, long-lasting transient, and common faults in the FUs, data buses etc. are not (always) covered.

B. Precomputation

Off-line application profiling is the initial step in P. The executed dynamic instructions are stored. At run time, before the application runs, the profiling data is loaded into the P-table.

The P-table is a hardware buffer of a limited size. It most probably cannot store all dynamic instructions seen in a relatively large application. A very large P-table would

be extremely expensive, but not very efficient, because the advantage of storing infrequent dynamic instructions can be expected to be negligible. The goal is therefore to fill a relatively small and cheap P-table with the most frequently used dynamic instructions. Thus, to reduce the application storage requirements, only a part of the collected profiling information can be actually stored with the program text. It is only needed to guarantee that there are sufficient dynamic instructions to fill the P-table as much as possible. This can be achieved by sorting the instructions in the profiling data by the frequency of occurrence, and filling the P-table in that order.

Figure 3 visualizes the P scheme at run time. An instruction proceeds regularly until the Execute stage. Then a P-table lookup is performed. If the necessary result is found, it is reused, and the instruction writes back on the next clock cycle. Otherwise, the instruction is issued to a corresponding FU, when it is available.

Note that unfortunately the P-table lookup cannot be performed before the Execute stage. This is because the instruction operand values are needed, and they are not guaranteed to be available at the earlier stages. This means that every instruction that hits the P-table has a single-cycle execution latency. For this reason, multi-cycle operations benefit from P (and also M) more than single-cycle operations [9]. But in some cases even single-cycle operations benefit from P and M. This happens when resources become a constraint and the issue of a single-cycle operation stalls because no appropriate FU is available. Result reuse eliminates the need of FUs for instructions that hit the table, reducing the resource pressure.

One of the important design considerations is the structure of the P-table. It needs a fast access time (one clock cycle in our organization). Unfortunately, structuring the table as a single large array with dynamic instructions as they appear in the sorted profiling list is not feasible. It would require a sequential access to every dynamic instruction in the table and comparison against the requested instruction, which is very time-consuming for relatively large tables.

A possible solution is to arrange (index) the table by instruction opcodes. We prefer this to indexing by a subset of input value bits (as some proposed M schemes do),

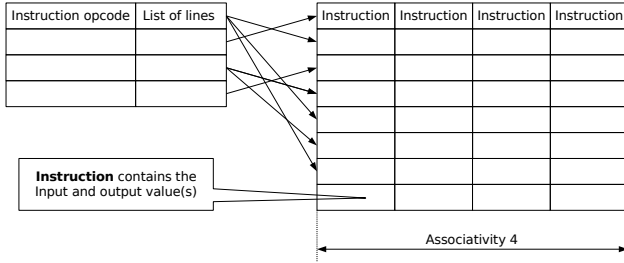


Figure 4. Precomputation table structure

because the opcode is often available earlier than the input values. The access to a table indexed by the opcodes can be started at the instruction decode stage, and finished when the input values are known. Indexing by the opcode rather than the program counter as in [8] also allows different static instructions to reuse each other’s results [9].

Our proposed P-table contains cache-like *sets* holding a number of *entries* with instruction input and output values. The number of entries per set is determined by the associativity. All the entries in a set are assigned the same opcode. Our experimental results show that among the most frequent instructions in the profiling data, some opcodes significantly dominate others. Thus, the table structure should allow uneven instruction distribution. This can be achieved by assigning multiple P-table sets to an opcode. The proposed P-table structure is shown in Figure 4. The *opcodes table* holds instruction opcodes and a list of the table sets assigned to it. When accessing the P-table, the opcodes table is consulted for the list of sets holding dynamic instructions with the required opcode. Then these sets are searched (preferably in parallel to reduce the access time) for the required input values combination. Note that for commutative arithmetic operations different combinations of the input operands can be checked to improve the hit rate and/or reduce the table size. Separate P-tables (or sets) might be used for instructions with operands of different size to optimize the resource usage.

The possible number of P-table sets assigned to a single opcode is limited. This is because the space in the opcodes table allocated for the list of used sets is limited. In addition, the requirement to provide a fast (fixed-latency) P-table access, which searches in all the sets (preferably in parallel), limits this number. On the other hand, the more sets per opcode are allowed, the more dynamic instructions with the most frequent opcodes are likely to be stored. Reducing this limit severely increases the risk to fill the P-table with less useful instructions, and thus to decrease the overall benefit in performance and fault coverage.

Note that the P-table lookup performed before issuing an instruction to a FU could appear on the critical path and affect the cycle time. If this happens, certain techniques

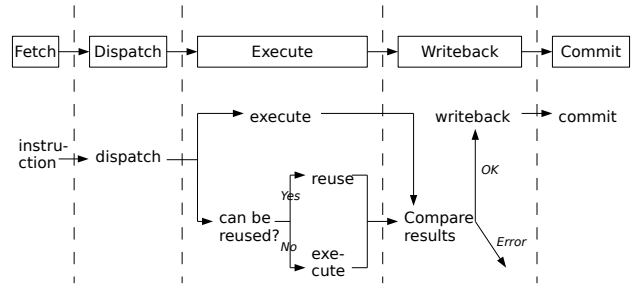


Figure 5. Instruction duplication with precomputation (D+P)

can diminish or solve the problem. For example, the P-table access can be started at earlier stages, because the instruction opcode is already available after decoding. Then, the lookup is finished when the operand values are available. Moreover, the instructions producing the input dependencies could initiate the P-table lookup as soon as the values are available. In [8], the authors also argue that memo-table accesses (which are similar to P-table accesses) are unlikely to be a serious problem.

C. Memoization

M is similar to the P scheme discussed in Section III-B, with a memo-table in place of the P-table. The only difference is that at the Writeback stage in M, the instruction updates the memo-table with its result. P never updates the P-table after it is filled by the program loader.

In our implementation the memo-table is a cache-like structure similar to the P-table (see Section III-B). However, the memo-table differs in that it does not have the opcodes table, and thus, only one set per opcode is available. This is because unlike the P-table, the memo-table needs to be updated very often (about every cycle, possibly for multiple instructions). This would require complex logic, increasing the access time and cost. We do not consider it feasible to update the memo-table with a complex structure including the opcodes table so frequently. The memo-table uses the Least Recently Used (LRU) replacement policy for the entries and the sets.

D. Duplication with Precomputation or Memoization

Figure 5 shows the proposed final scheme employing both instruction duplication and P (D+P). An instruction is always executed in the regular way, and executed once more if the result is not found in the P-table. The computed result is then compared to the reused or recomputed result at the Writeback stage. The D+M scheme is similar to Figure 5, with the addition of the memo-table update at the Writeback stage.

To improve the reliability of the D+P scheme, a highly reliable system which performs profiling can be used. This does not increase the total cost significantly, because the

profiling needs to be performed only once per application, and is done off-line. A sufficiently reliable P-table on the cheaper target computing system can then provide the reliability of the expensive profiling system at a low cost. Protecting the P-table with Error Correcting/Detecting Codes (ECC) [14] and some control logic protection would be sufficient for that. If the profiling system and the P-table are sufficiently reliable, the execution of the dynamic instructions that hit the P-table can be skipped altogether without any reliability loss. This scheme will be referred to as *duplication+precomputation only (D+PO)*. D+P always executes the instructions at least once, while D+PO does not execute instructions that hit the P-table at all.

IV. EXPERIMENTAL RESULTS

The experiments evaluate the fault coverage and performance of P used for fault detection (D+P and D+PO schemes), and compare these with M (D+M scheme). The experiments consider the following schemes: (1) original (without fault detection), (2) original with precomputation (without fault detection, as described in Section III-B) and (3) with memoization (Section III-C), (4) instruction duplication (Section III-A), (5) D+P, (6) D+PO and (7) D+M (Section III-D). Several integer and floating-point benchmarks from the SPEC CPU2000 suite [15] are used, as well as the JPEG encoder multimedia application. Since when using P applications are unlikely to have the same input at the profiling and deployment stages, we use different inputs for profiling and actual execution of every application.

Section IV-A describes the used simulation platform. Section IV-B discusses the fault coverage of the considered methods. Section IV-C evaluates how P improves the performance of the instruction duplication scheme, and compares with M. Performance improvement can be expected to correspond to a proportional energy reduction, because the execution time shortens. This, however, depends on how much additional energy is consumed by the P- and memo-tables. A detailed experimental energy consumption evaluation is outside the scope of this work. Intuitively it is obvious that for similarly configured tables, the P-table should consume less energy than the memo-table, because it does not need to be updated at every clock cycle. Finally, Section IV-D analyzes the hit rate of both P and M and identifies the bottlenecks.

A. Simulation Platform

The experiments are performed using modified versions of the SimpleScalar simulator tool set [16]. The produced simulators (based on *sim-outorder*) support all the considered schemes. The processor configuration used is presented in Table I. In Section IV-C the number of integer ALUs is varied to demonstrate how instruction reuse affects the throughput. The configuration (number of sets and associativity) of the P- and memo-tables is also varied. The

Table I
PROCESSOR CONFIGURATION

Fetch/Dec./Issue Width	8
Fetch/Decode/Issue/Commit Width	8
Branch Predictor	Combined, 8K meta-table
BTB Size	1 K, 2-way associative
Return Address Stack Size	8
Branch Misprediction Latency	7 cycles
RUU Size	128
LSQ Size	64
# of Int. ALUs	1 to 4
# of Int. Mult./Div.	1
# of FP ALUs	1
# of FP Mult./Div.	1
Memory Latency	112 cycles (first chunk), 2 cycles (subsequent chunks)
L1 Data Cache	32 KB, 2-way set associative
L1 Instruction Cache	32 KB, 2-way set associative
L2 Unified Cache	512 KB, 4-way set associative

benchmark applications were run either until completion or until the first 100 million instructions committed.

To minimize the resource overhead and keep the cost low, only small to modestly-sized P- and memo-tables are examined in our experiments. The number of sets in the tables varies from 2 to 128, and associativity from 4 to 8. This means that the total P-/memo-table size varies from approximately 192 B to 24 KB (assuming that the tables hold two 64-bit input and one output values, without protective information). Further in the text $P(8,4)$ means a 4-way set associative P-table with 8 sets. A similarly configured memo-table is denoted as $M(8,4)$. $P(8,4)$ and $M(8,4)$ can keep 8 different opcodes, and 4 dynamic instructions for every opcode. Only computational instructions (integer and floating-point) are reused, other instructions such as memory accesses are not. Note, however, that a memory access instruction is split in two parts: address computation and the memory request. The address computation part, which is an integer operation, can be reused.

B. Fault Coverage

It is difficult to compare (quantitatively) the fault coverage of the D+P and D+M schemes. The Architectural Vulnerability Factor (AVF) [17], for example, cannot be used because it measures the bits vulnerability of the instructions residing in the memo- and P-tables. This does not take into account the types of faults (long-lasting, permanent etc.) covered by the considered schemes, and how many instructions are actually protected. Therefore, two indirect methods are used to evaluate the fault coverage: the average memo-table instruction lifetime and the hit rate.

M improves the fault coverage of pure duplication by targeting some long-lasting transient, permanent, and common faults in FUs, data buses etc. M achieves this due to the time gap which it inserts between the execution of the memoized instruction and its subsequent (being verified)

Table II
AVERAGE MEMO-TABLE INSTRUCTION LIFETIME FOR DIFFERENT D+M CONFIGURATIONS (% OF THE TOTAL EXECUTION TIME).

D+M config.	ammp	art	equake	mcf	vortex	gcc	mesa	bzip2	cjpeg
(2,4) to (8,8)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(64,4)	6.0	0.9	8.3	4.6	2.3	11.0	8.6	28.2	17.0
(64,8)	4.7	0.8	6.2	3.2	1.4	8.1	8.6	24.1	12.8
(128,4)	6.0	0.9	8.3	4.6	2.3	11.0	8.6	28.2	17.0
(128,8)	4.7	0.8	6.2	3.2	1.4	8.1	8.6	24.1	12.8

execution. Assume that an instruction result is memoized at time T_1 . N clock cycles later it is compared to the recomputed result at time T_2 . M covers the long-lasting faults that appeared before T_1 and disappeared between T_1 and T_2 , or appeared between T_1 and T_2 . Other long-lasting and permanent faults cannot be detected by M , because they affect both the memoized and recomputed results in the same way (if executed on the same FU). Hence, the number N determines how efficient M is in covering long-lasting faults. The greater N is, the more long-lasting faults are likely to appear or disappear during this time. We call N the *memo-table instruction lifetime*, because it determines how long an instruction resides in the memo-table before it is reused. Note that pure duplication 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 M over the pure duplication depends on how much greater the memo-table instruction lifetime is than the duplication gap. In addition, $D+M$ simplifies the fault location/recovery of the duplication scheme. When two results do not match in the duplication scheme, either FU involved in the computation could be faulty. With $D+M$, the result held in the memo-table has already been verified by redundant computation before it was saved. Thus, if the newly computed and the memoized results do not match, the FU on which the last computation was performed is most likely to be faulty. The situation corresponds to Triple Modular Redundancy (TMR) [3]. This is, however, only true if the storage elements in the memo-table provide sufficient reliability, guaranteeing that the saved result is not corrupted when being reused. This can be achieved, for example, by protecting the memo-table with ECC.

$D+P$ has all the advantages of $D+M$, and further improves the fault coverage by protecting against all the long-lasting transient, permanent, and common faults. This is due to the fact that the application profiling is performed at a different time than the actual execution, and most likely even on different hardware (host machine). Thus, permanent, long-lasting transient, and common faults cannot affect both the profiling and the actual execution results.

Table II presents the average memo-table instruction

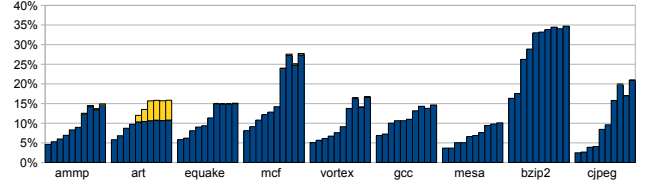


Figure 6. P-table hit rates. The bars from left to right: P(2,4), P(2,8), P(4,4), P(4,8), P(8,4), P(8,8), P(64,4), P(64,8), P(128,4), P(128,8).

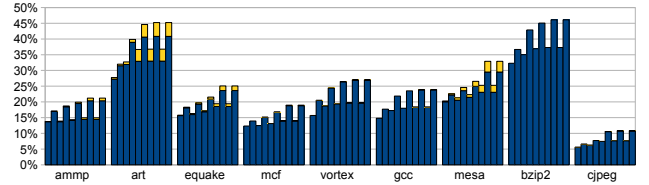


Figure 7. Memo-table hit rates. The bars from left to right: M(2,4), M(2,8), M(4,4), M(4,8), M(8,4), M(8,8), M(64,4), M(64,8), M(128,4), M(128,8).

lifetime for different $D+M$ configurations. The instruction lifetime is measured in clock cycles, and presented as a percentage of the total execution time. For the smallest configuration $D+M(2,4)$ the average instruction lifetime is 3.5 (ammp) to 8.2 (gcc) clock cycles. This means that the long-lasting fault coverage of $D+M(2,4)$ is comparable to that of pure instruction duplication, because a gap of 3 to 8 clock cycles can be expected between the execution of two redundant instruction copies in the duplication scheme. For all configurations from $D+M(2,4)$ to $D+M(8,8)$ the average instruction lifetime is up to 0.02% of the total application execution time. With larger memo-table sizes, the average instruction lifetime grows significantly, improving the long-lasting fault coverage of $D+M$. However, as Table II shows, it is between 1.4% and 28.2% of the total execution time even for the largest memo-tables. $D+P$ covers 100% of the long-lasting and permanent faults in all configurations.

Another indirect fault coverage measure is the memo- and P-table hit rate. The hit rate determines how many instructions are protected by P or M in the $D+P$ and $D+M$ schemes, and how many are only duplicated. Unlike performance, the hit rate does not vary significantly when the number of FUs in the system changes. The factors on which the hit rate depends are the table size and configuration. Thus, only the results for systems with 1 integer ALU are presented.

Figure 6 presents hit rates for P-tables of different configurations. For every benchmark, the bars represent the following P-table configurations: P(2,4), P(2,8), P(4,4), P(4,8), P(8,4), P(8,8), P(64,4), P(64,8), P(128,4), P(128,8). Figure 7 shows hit rates for similarly configured memo-tables. The upper part of every bar in the hit rate graphs represents the fraction of reused multi-cycle instructions, and the lower

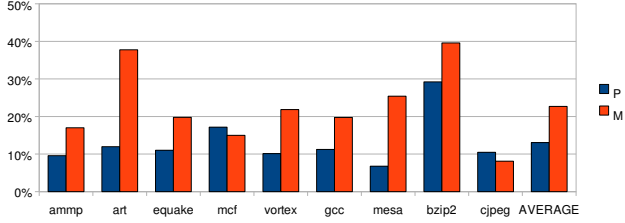


Figure 8. Comparison of the average precomputation and memoization hit rates.

part the fraction of reused single-cycle instructions. In most cases the single-cycle part dominates, and often the multi-cycle part is not even visible.

Figure 8 compares the average hit rate of different P- and memo-table configurations. For 2 out of 9 benchmarks, P has a lower hit rate than M, which covers almost twice as many instructions on average. However, from Figure 6 and Figure 7 we can see that the P hit rate grows more rapidly with the increasing table size than the M hit rate does. The hit rate advantage of M over P is most significant for small table sizes. However, as shown by the average memo-table instruction lifetime, for small table sizes, the long-lasting fault coverage of D+M is comparable to that of duplication. Thus, the hit rate advantage does not mean an actual fault coverage advantage for smaller table sizes.

C. Performance

Figure 9 compares the Instructions Per Cycle (IPC) decrease of the considered duplication-based schemes over the original (non-redundant) execution on a system with 2 integer ALUs. It presents the average IPC decrease over the different table configurations. Note that for the duplication-based schemes, the number of executed instructions does not double, but every instruction executes twice. Thus, the IPC of the duplication-based schemes is computed for the original instructions, and can be compared to the IPC of the none-redundant schemes. Figure 9 demonstrates that on average, the D+M scheme outperforms D+P by 4.6%, while D+PO outperforms D+M by 9%. However, there are exceptions: for mcf, D+P outperforms D+M by 7.6%, and for art, D+M outperforms D+PO by 12.6%.

Figure 10 shows how the performance overhead of the different duplication schemes (across all benchmarks) depends on the number of integer ALUs. The more ALUs are available, the smaller the IPC increase due to P and M. This is because P and M decrease the pressure on the FUs. Thus, their advantage is most prominent when the number of available FUs is the system bottleneck. To analyze how using P compares to the addition of FUs, Figure 11 shows the IPC increase over the full duplication scheme that can be achieved by increasing the number of integer ALUs from 1 to 2, and by different D+PO configurations. The figure shows

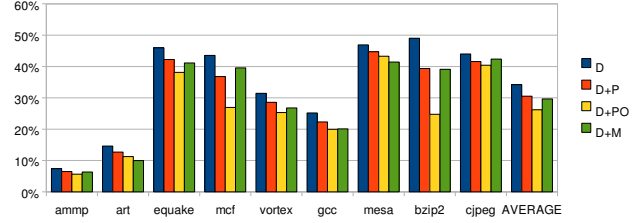


Figure 9. Performance overhead (IPC decrease) of different duplication schemes over the original (non-redundant) scheme on a system with 2 integer ALUs. Average over the different table configurations.

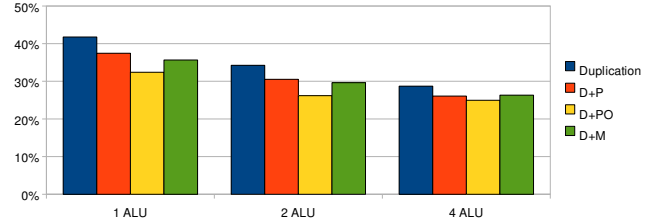


Figure 10. Average IPC decrease of different schemes with different number of integer ALUs.

that for some applications (such as ammp, mcf, and bzip2), D+PO approaches the speedup of an additional integer ALU. For example, for mcf, D+PO(64,8) applied to a system with 1 integer ALU performs comparably to the duplication on a system with 2 integer ALUs. In other words, for some applications P with larger tables can function as a substitute to an additional FU.

Figure 12 analyzes the average IPC decrease (across all the benchmarks) for different memo- and P-table configurations. The graph shows that for D+P, (64,4) and (64,8) are the most optimal configurations from the performance vs. cost trade-off point of view. D+P(128,4) doubles the table size of D+P(64,4), but performs only 6.4% better. As shown in Section IV-B, the P hit rate grows more rapidly when the table size increases than M hit rate does. This explains why in Figure 12 the performance of the precomputation-based schemes improves more than that of D+M when the table size increases.

D. Analysis

Figure 6 and Figure 7 show that the hit rate varies significantly for different applications, depending on how many redundant computations the benchmark performs. The hit rate also depends on the table configuration. In general, larger tables increase the hit rate for all the applications. However, both P and M demonstrate more significant hit rate improvements when enlarging smaller tables than larger ones. For example, for art, the hit rate almost stops increasing after the configurations P(64,4) and M(64,8) are reached. For P (Figure 6), we explain this phenomenon by the fact that the instructions in the profiling data are sorted by their

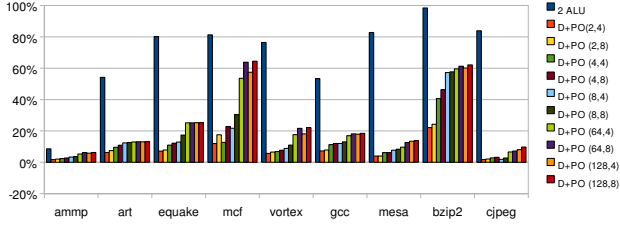


Figure 11. Average IPC increase over duplication on a system with 1 integer ALU.

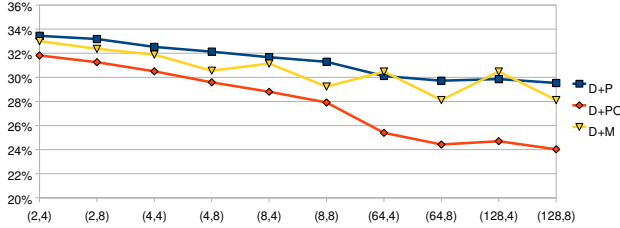


Figure 12. Average IPC decrease for different table configurations.

frequency of occurrence. Thus, the first entries in the P-table are filled with the most frequent instructions. The larger the table is, the more infrequent instructions it holds. We have observed in the profiling data that within the 50 most frequent instructions, the first one is 14 to 100 times more frequent than the last one. This suggests that the P-table size should be limited to achieve the maximum effectiveness at the minimum cost (to store only frequent instructions in the table). Our experiments demonstrate that for the considered applications, the configurations P(64,4) and P(64,8) could be recommended. Thereafter, the hit rate hardly increases.

Another metric to explain the results is the table occupation. Unfortunately, less than 100% of the P-table space is usually used. This happens because every table set can only hold instructions with the same opcode, and some opcodes appear rarely. Figure 13 shows the P-table occupation for different configurations. For most applications larger tables provide a better utilization percentage, because the same number of empty table slots represent a smaller part of the whole table. For all applications smaller associativity leads to a better table occupation, because fewer dynamic instructions with rare opcodes are needed to fill all the entries in the set. The recommended configuration P(64,4) performs among the best ones also from the table occupation point of view.

For M (Figure 7), we explain the saturation of the hit rate growth by the structure of the memo-table. We have observed (from the P profiling data, for example) that some opcodes seriously dominate others during execution. However, in our memo-table, all opcodes receive equal space (one memo-table set). As a result, in large memo-tables, many sets are occupied by infrequent opcodes. For

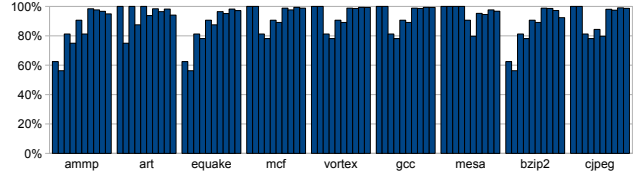


Figure 13. Precomputation table utilization (% of the whole table).

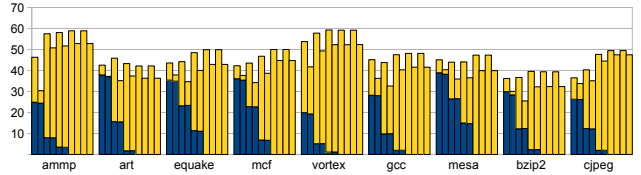


Figure 14. Memo-table evictions ($\times 1$ million).

an optimal memo-table space utilization, more space would be available for frequent opcodes than for others. Then, dynamic instances of frequent instructions would need to be evicted less often. Figure 14 depicts how many evictions happen in the memo-table. The upper part of every bar represents the number of entry evictions, and the lower part the fraction of set evictions. An entry eviction is the replacement of a dynamic instruction. A set eviction is the replacement of a whole set in the memo-table. It happens when a new opcode has to be stored, but no free sets are available. For small memo-tables, set evictions are frequent, because not enough sets are present for the opcodes. This should certainly be avoided, because set eviction is an expensive operation both in terms of performance (multiple instructions are lost from the memo-table at the same time) and resource/energy consumption (the whole set is updated). When the number of sets increases, the number of set evictions reduces, until they completely disappear.

Figure 6 and Figure 7 show that the hit rate achieved by P and M can differ significantly for the same application. We explain this by the differences in the distribution of redundant dynamic instructions in the applications. P focuses on the globally dominant dynamic instructions, while M focuses on the local redundancy. P performs better when the dominant dynamic instructions are distributed across the whole application, in which case in M they can be evicted before being reused. M performs better when many redundant dynamic instructions (which do not have to be globally dominant) execute close to each other. This is also the reason why for most considered benchmarks M reuses more multi-cycle instructions than P does. These multi-cycle dynamic instructions introduce a certain local redundancy, but single-cycle operations dominate in the whole application.

V. CONCLUSIONS

This work proposes and evaluates instruction precomputation for error detection purposes. Instruction precomputation is compared to memoization playing the same role. A system organization based on the instruction duplication scheme adapted from [10] is proposed (see Section III-D). Whenever possible, the results obtained from the P- (or memo-) table are used instead of being recomputed. The D+P scheme improves the fault coverage of the duplication and D+M schemes, by covering more of long-lasting transient, permanent, and common faults. If the profiling is performed on a highly reliable system and the P-table is protected with ECC, D+P also improves the reliability of the results stored in the table at almost no additional cost, because the profiling needs to be done only once per application. From the performance point of view, we have observed both D+P and D+M outperforming each other. The D+P schemes perform worse than the D+M schemes for most benchmarks, while the D+PO schemes usually outperform them. The performance benefits of instruction reuse (both P and M) is most significant in systems with a limited number of FUs. Because instruction reuse reduces the pressure on the FUs, a P- or memo-table may in some cases serve as a replacement for additional FUs. This is especially important for complex FUs such as floating point units. Such a unit may require significantly more area than a modestly-sized P- or memo-table, and, furthermore, a table hit may take less time than a long-latency floating point operation.

For future work we plan to investigate whether a combination of P and M can achieve better results with reduced cost. The P-table would hold the most frequent instructions, and the rest of instructions would utilize M. This way, the most frequent instructions can be protected the most (by a highly reliable profiling and P-table). In addition, we plan to further reduce the number of instructions appearing in both the P- and memo-tables by using trivial computation detection. Furthermore, we plan to experiment with different P-table structures to optimize the cost and performance.

REFERENCES

- [1] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic," in *DSN-02: Proc. 2002 Int. Conf. on Dependable Systems and Networks*, Washington, DC, USA, 2002, pp. 389–398.
- [2] J. Yi, R. Sendag, and D. Lilja, "Increasing Instruction-Level Parallelism with Instruction Precomputation," in *Euro-Par-02: Proc. 8th Int. Euro-Par Conf. on Parallel Processing*. London, UK: Springer-Verlag, Aug 2002, pp. 481–485.
- [3] B. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley, Jan 1989.
- [4] D. Michie, "Memo Functions and Machine Learning," *Nature* 218, pp. 19–22, 1968.
- [5] P. Norvig, "Techniques for Automatic Memoization with Applications to Context-Free Parsing," *Computational Linguistics*, vol. 17, no. 1, pp. 91–98, Mar 1991.
- [6] S. Richardson, "Exploiting Trivial and Redundant Computation," in *Proc. 11th Symp. on Computer Arithmetic*, July 1993, pp. 220–227.
- [7] S. F. Oberman and M. J. Flynn, "Reducing Division Latency with Reciprocal Caches," *Reliable Computing*, vol. 2, no. 2, pp. 147–153, Apr 1996.
- [8] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse," in *ISCA-97: Proc. 24th Annual Int. Symp. on Computer Architecture*. New York, NY, USA: ACM, 1997, pp. 194–205.
- [9] D. Citron, D. Feitelson, and L. Rudolph, "Accelerating Multi-Media Processing by Implementing Memoing in Multiplication and Division Units," *ASPLOS-VIII: Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, vol. 32, no. 5, pp. 252–261, 1998.
- [10] M. Franklin, "A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors," *IEEE Int. Workshop on Defect and Fault Tolerance in VLSI Systems*, pp. 207–215, Nov 1995.
- [11] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam, "A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy," in *ISCA-04: Proc. of the 31st Annual Int. Symp. on Computer Architecture*. Washington, DC, USA: IEEE Computer Society Press, 2004, pp. 376–386.
- [12] J. Ray, J. Hoe, and B. Falsafi, "Dual Use of Superscalar Datapath for Transient Fault Detection and Recovery," *MICRO-34*, pp. 214–224, Dec 2001.
- [13] M. Goma and T. Vijaykumar, "Opportunistic Transient Fault Detection," *ISCA-05: Proc. 32nd Annual Int. Symp. on Computer Architecture*, pp. 172–183, Jun 2005.
- [14] T. Rao and E. Fujiwara, *Error-Control Coding for Computer Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.
- [15] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.
- [16] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.
- [17] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "Measuring Architectural Vulnerability Factors," *IEEE Micro*, vol. 23, no. 6, pp. 70–75, 2003.