ABSTRACT

It is normal to expect the processor to produce a correct result. But it may be the case that a software application does not need "strict" correctness, but rather limited correctness. The goal of this project is to search for potential definition of "non-strict" correctness, as well as investigate the improvements in performance and power consumption that can be achieved in this new model.

This project proposes a hardware/software system design for applying the approximate computing idea to the well-known problem of memory latency due to cache misses. We use three techniques to predict the values of cache misses: Last Value, Stride and Adaptive prediction. Because our approach is recovery-free, evaluation of quality of service after the execution is our main way to control the correctness of speculative predictions.

Our evaluations show that the system we propose can successfully cover significant amount of L1/L2 cache misses. On average we can cover 62% of L1 misses with more aggressive, but less precise techniques, and 40% with the more precise, but less aggressive Adaptive technique. In a simple in-order model this corresponds to an average speedup of 12.5% and 10% respectively. For applu benchmark we can successfully cover more than 56% of L2 misses, that leads to 69% performance improvement.

1. INTRODUCTION

Typically people are expecting the processor to produce a correct result. It usually means that the full architectural state should be numerically perfect for every cycle, or, in the worst case, everything that is "visible" to the programmer should be perfect. But it may be the case that the software algorithm does not need such a "strict" correctness, but only limited correctness. If a certain program is not precise by its nature, such as approximation, video/audio encoding, classification application, it most probably does not require all the computations to be precise, and can exhibit even higher than usual fault tolerance [10]. Whether a fault is acceptable or not depends on the definition of the quality of service (QoS) [16] for this application.

The most amenable programs to less precise computations are AI or Biological applications [21], as well as some multimedia programs [12, 26]. Those can be very tolerant to potential "defects" in execution [13, 10] and produce acceptable results even in the presence of significant amount of failures. So, it is possible to exploit those characteristics and get performance improvement by simplifying hardware, changing the algorithm itself with an aggressive compiler optimization, or both. Moreover, such performance improvements may result in significantly less computation, and, hence, power savings [5, 2].

The tradeoffs between quality of service and performance are well known to the programmers, and they typically use either special heuristics or/and code versioning to achieve this. The problem is that these changes are hidden from users, hard to maintain, and may be easily broken by hardware/software changes. Moreover, these changes are very specific for the problem they address, and, as the result, far from being general.

In this project we are proposing an automatic system that can resolve some of these issues by using both software and hardware techniques. The two main questions we are trying to address are:

1. How can we replace the notion of "strict" correctness?
2. What are the aggressive optimizations that can be done in hardware/software to get benefit from the "non-strict" correctness?

In the search for a good definition of "non-strict" correctness we define a QoS on a per application basis with some general rules that are applicable for a wide variety of applications. It is defined on a [0, 1] interval, with 0 meaning full correctness, and 1 meaning totally unacceptable result. The user can specify the QoS limit, i.e. the amount of quality he/she is willing to trade for the increasing performance and decreasing power. We provide several templates on how QoS can be computed, but our system is flexible to use user-defined QoS with some simple rules on how to define it.

Previously, researchers considered the possibility of ignoring
1.2 Contributions

Because only additional prefetches were generated, but their approach did not affect the correctness of the investigated the possibility of recovery-free value prediction [30], free. We do not dynamically handle the effects of potential The technique we are proposing in this project is recovery-free. L2 caches, and returning some value making use of simple value prediction techniques. The true value is fetched when available and only if it is needed. Similar techniques can be applied to branch mispredictions and load-store dependencies.

1.1 Recovery-Free Value Prediction

The technique we are proposing in this project is recovery-free. We do not dynamically handle the effects of potential mispredictions, but instead we evaluate the resulting QoS of the whole execution. Researchers had previously investigated the possibility of recovery-free value prediction [30], but their approach did not affect the correctness of the execution, because only additional prefetches were generated.

1.2 Contributions

This paper makes the following main contributions:

1. Introduces the design and implementation of a simple, flexible and automatic software/hardware system that can decrease the application’s execution time, as well as save power by speculatively predicting values that were missing in the caches.

2. Extension of the existing binary instrumentation tool called Valgrind [20]. A Valgrind sub-tool called Cachegrind [19] was extended to not only collect the information about cache misses in L1 and L2 caches, but also predicting values of those cache misses using different value prediction techniques.

3. System evaluation that shows significant decrease in the amount of both L1 and L2 cache misses on a representative group of benchmarks.

4. Adaptive algorithm that can automatically learn nearly-optimal configuration for an application in our system when given a QoS limit and a QoS metric.

The remainder of this paper is organized as follows. Section 2 describes our motivation and initial experiments. Section 3 and Section 4 discuss the design and implementation of our system. Experimental methodology and evaluation are provided in Section 5 and Section 6. Section 7 gives a brief overview of the related work. Section 8 concludes the work, and Section 9 provides information about shortcomings and future work.

2. MOTIVATION AND INITIAL EXPERIMENTS

To explore opportunities provided by "non-strict" correctness we investigate the possibility of hiding the memory stall latency caused by cache misses. Typical penalties for L1 data cache miss are 10-20 cycles , with as high as 400 cycles for L2 cache misses [19]. Previous works [19, 8] showed that these stalls can severely degrade system performance, and thus removing them can result in significant performance improvement i.e. 23 % initialization misses are responsible for 41 % potential speedup in the work by Lewis et al. [9].

First of all we evaluate the amount of misses, for both L1 and L2 caches, on a group of representative benchmarks: SPEC2000 [26], BioBench [21] and MediaBench [12]. Details of our experimental setup are in Section 5. Table 1 presents the results of our experiment. The notation for columns is 'D1/D2' for data cache level 1 and 2 respectively, 'm' for miss, 'r' for read, 'w' for write and Dr/Dw for total amount of memory reads/writes. The 'Memory Access / 1K' column means the amount of memory accesses per one thousand instruction executed.

There are many observations that can be made based on this data, but we will concentrate on the ones that motivate our project. First of all, there are definitely enough benchmarks that have significant amount of L1 cache misses on read, especially art (59.5%), 002.tiger (59%), mcf (33.3%) and swim (14.9%). These benchmarks can be the source of potential improvement with our system. For swim many of those D1mr will be also D2mr (43.82%); that results in a huge stalling during the execution. A second observation is that in several cases almost all D1mr/w are also D2mr/w: applu, lucas, mgrid, swim, wupwise, fma3d, 001.mummer, 002.tiger, h263enc. Hence, the L2 cache in those cases seems very inefficient. This is a potential for speculation, since L2 misses can result in stalling for 400 cycles or more [19]. Hence, there is certainly room for improvement here.

We see the significance of cache misses for these benchmarks, but we still haven’t addressed a serious question - how these cache misses are distributed across applications. If the cache miss distribution around the entire application is not concentrated in certain points, it is harder to predict them, because we need to analyze significantly more places in the code. Our expectation is that these points of concentration (sometimes called delinquent loads) should exist in our benchmarks, and their amount should be relatively small.

Figure 1(a) and Figure 1(b) present the coverage for L1mr and L2mr for several SPEC benchmarks. For every benchmark on these graphs we observe the amount of cache misses (in %) that corresponds to a certain amount of delinquent loads. As we can see in most of the cases only 10-15 delinquent loads/stores are needed to be considered to cover the vast majority of missed penalty during the execution. Hence, predicting only the values of these loads and stores is good enough to achieve better performance.

3. SYSTEM DESIGN

Figure 2 provides the general overview of our system, more details about the system itself and algorithms used are described in this section.

3.1 Profiling

As we showed in Section 2 cache misses typically occur in very few places in the code. Identifying these cache misses usually require the use of a profiling tool such as gprof, vTune [29], or Valgrind [28]. We propose an automatic system that can identify these cache misses in the code, and
<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>D1mr / Dr %</th>
<th>D2mr / D1mr %</th>
<th>Dr / Dw</th>
<th>D1mw / Dw %</th>
<th>D2mw / D1mw %</th>
<th>Memory Access/1K</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>5.69</td>
<td>4.65</td>
<td>3.02</td>
<td>0.51</td>
<td>19.22</td>
<td>394</td>
</tr>
<tr>
<td>applu</td>
<td>2.17</td>
<td>86.44</td>
<td>3.34</td>
<td>4.02</td>
<td>99.99</td>
<td>356</td>
</tr>
<tr>
<td>apsi</td>
<td>3.16</td>
<td>28.28</td>
<td>2.88</td>
<td>3.17</td>
<td>20.78</td>
<td>332</td>
</tr>
<tr>
<td>art</td>
<td>59.55</td>
<td>0.00</td>
<td>2.35</td>
<td>8.12</td>
<td>0.02</td>
<td>223</td>
</tr>
<tr>
<td>bzip2</td>
<td>2.51</td>
<td>2.74</td>
<td>2.32</td>
<td>1.61</td>
<td>14.37</td>
<td>376</td>
</tr>
<tr>
<td>equake</td>
<td>5.72</td>
<td>88.78</td>
<td>6.80</td>
<td>1.37</td>
<td>50.62</td>
<td>570</td>
</tr>
<tr>
<td>facerec</td>
<td>1.76</td>
<td>13.77</td>
<td>2.72</td>
<td>3.46</td>
<td>44.72</td>
<td>440</td>
</tr>
<tr>
<td>fma3d</td>
<td>2.65</td>
<td>88.39</td>
<td>1.97</td>
<td>1.47</td>
<td>77.01</td>
<td>471</td>
</tr>
<tr>
<td>gzip</td>
<td>9.76</td>
<td>0.64</td>
<td>3.14</td>
<td>1.13</td>
<td>31.79</td>
<td>306</td>
</tr>
<tr>
<td>lucas</td>
<td>1.23</td>
<td>99.68</td>
<td>2.67</td>
<td>3.65</td>
<td>98.48</td>
<td>494</td>
</tr>
<tr>
<td>mcf</td>
<td>33.88</td>
<td>27.43</td>
<td>4.26</td>
<td>18.05</td>
<td>34.13</td>
<td>433</td>
</tr>
<tr>
<td>mesa</td>
<td>0.18</td>
<td>7.17</td>
<td>1.94</td>
<td>0.75</td>
<td>47.36</td>
<td>439</td>
</tr>
<tr>
<td>mgrid</td>
<td>1.78</td>
<td>51.64</td>
<td>24.44</td>
<td>6.48</td>
<td>98.34</td>
<td>490</td>
</tr>
<tr>
<td>parser</td>
<td>4.43</td>
<td>4.50</td>
<td>2.55</td>
<td>1.40</td>
<td>21.52</td>
<td>449</td>
</tr>
<tr>
<td>sixtrack</td>
<td>0.22</td>
<td>0.02</td>
<td>0.94</td>
<td>0.00</td>
<td>17.58</td>
<td>304</td>
</tr>
<tr>
<td>swim</td>
<td>14.84</td>
<td>43.97</td>
<td>2.62</td>
<td>5.51</td>
<td>98.97</td>
<td>372</td>
</tr>
<tr>
<td>twolf</td>
<td>8.11</td>
<td>0.00</td>
<td>3.13</td>
<td>4.52</td>
<td>0.01</td>
<td>366</td>
</tr>
<tr>
<td>wupwise</td>
<td>1.43</td>
<td>83.21</td>
<td>2.61</td>
<td>0.37</td>
<td>100.00</td>
<td>280</td>
</tr>
<tr>
<td>MediaBench</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h263enc</td>
<td>0.03</td>
<td>5.74</td>
<td>18.08</td>
<td>0.20</td>
<td>58.53</td>
<td>243</td>
</tr>
<tr>
<td>h264enc</td>
<td>3.05</td>
<td>0.18</td>
<td>2.34</td>
<td>5.20</td>
<td>0.12</td>
<td>526</td>
</tr>
<tr>
<td>mpeg2enc</td>
<td>0.12</td>
<td>3.20</td>
<td>15.31</td>
<td>0.79</td>
<td>22.20</td>
<td>259</td>
</tr>
<tr>
<td>BioBench</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001.mummer</td>
<td>3.86</td>
<td>93.80</td>
<td>1.73</td>
<td>0.05</td>
<td>58.34</td>
<td>488</td>
</tr>
<tr>
<td>002.liger</td>
<td>59.04</td>
<td>50.45</td>
<td>26.35</td>
<td>2.27</td>
<td>63.20</td>
<td>388</td>
</tr>
<tr>
<td>003.cluster</td>
<td>0.00</td>
<td>0.07</td>
<td>3.39</td>
<td>0.00</td>
<td>13.39</td>
<td>274</td>
</tr>
<tr>
<td>004.hmmer</td>
<td>0.49</td>
<td>0.21</td>
<td>5.37</td>
<td>3.09</td>
<td>11.89</td>
<td>560</td>
</tr>
<tr>
<td>006.phylip</td>
<td>0.00</td>
<td>0.21</td>
<td>2.22</td>
<td>0.00</td>
<td>57.46</td>
<td>367</td>
</tr>
<tr>
<td>007.fasta</td>
<td>0.67</td>
<td>1.96</td>
<td>2.84</td>
<td>0.70</td>
<td>2.88</td>
<td>359</td>
</tr>
</tbody>
</table>

Table 1: Cachegrind results for SPEC2000, MediaBench and BioBench.

![L1 cache miss on read evaluation](image1)

(a) L1 cache miss on read coverage.

![L2 cache miss on read evaluation](image2)

(b) L2 cache miss on read coverage.

Figure 1: Cache miss coverage.
apply speculative optimizations if possible. Hence, we introduce an automatic profiling phase using our extension of a Valgrind tool named Cachegrind [19]. The idea is to provide users with not only flexible and easy to use tools for finding cache misses, but also doing it on a fine-grain scale: program counter instead of source-code line; and feeding this information to the hardware with value prediction support.

3.2 Value Prediction

After the profile information about delinquent loads is collected, it is fed to the next level as a configuration file. This file contains information about the addresses that are most critical to predict in the code. This information is used to predict the values for cache misses. Currently, we tested three potential techniques for value prediction: Last Value [11] prediction, Stride prediction [25], and a modification of stride prediction with confidence counters which we call Adaptive prediction. There are other techniques, such as context-sensitive prediction [25], that can potentially provide better accuracy, but the goal of this work is not to investigate all possible prediction techniques (all of them are imperfect), but rather to show how to get the benefits of the technique using a "non-strict" correctness definition.

Last Value technique predicts the current value to be the same as the last observed value at the same memory address, the stride technique predicts that data is strided, meaning that it has a constant offset. The adaptive scheme uses confidence saturating counters to make its prediction. If the prediction (we use stride) is correct for certain number of times consequently, then we start to believe our prediction, and, hence, generate a prediction. Otherwise, prediction is not generated. The confidence counters can increase or decrease when the prediction is correct or wrong respectively, but they can not exceed certain range: they are always less or equal than 7, and bigger or equal to 0.

Figure 3 shows the structure of our predictor (for adaptive scheme). Since the amount of targets we are interested in is limited, we can expect this predictor to be implemented in hardware, and become a part of memory hierarchy similar to prefetch engine or stream buffer. We want our predictor not to be on the critical path of execution, so we do prediction only on cache misses: L1 or L2 caches. The simplicity of our techniques allows this information to be computed extremely fast. To collect the history for our predictor, we need to save information about the last value accessed for this address, but this can be done in parallel with the main execution, and should not slow down the pipeline.

3.3 Quality of Service Metrics

Since our prediction is speculative by its nature and we do not use any recovery mechanism, it is clear that the results can be wrong, or even the whole application execution can be broken. To address this issue we want to have a good quality of service (QoS) metric for different benchmarks. Even though QoS for different applications is not the same, we want to have a general guideline to define it. First, if the application crashes, goes to infinite cycle, or doesn’t produce any expected output, its QoS should be considered completely unacceptable. We will define it as 1 in this case. On the other hand, complete match with expected output, meaning all expected output values: files, stdout, stderr, return values; is defined to be 0. The variations from those 2 cases lay in a [0..1] interval.

Suppose we have several different outputs: out1, out2, ..., outk, and the correct outputs are corr1, corr2, ..., corrk. Then we define the QoS for them as:

$$QoS = \frac{1}{k} \sum_{i=1}^{k} w_i \times \frac{|out_i - corr_i|}{|out_i| + |corr_i|},$$

where $w_i$ are the weights for every single output, such that $\sum_{i=1}^{k} w_i = 1$. This formula can be considered as a variation of the distortion [22] term.

The user can define his/her own QoS or use the templates made by us, but is required to provide us a $QoS_{limit}$: the value in a [0..1] interval that corresponds to the maximum loss of accuracy he or she accepts. The $QoS$ and $QoS_{limit}$ are used to find the optimal configuration file for this application. More details about $QoS$ for specific applications are in Section 5.

3.4 Finding the Optimal Configuration File

We want to find the biggest subset, in terms of cache misses coverage, that satisfies our QoS requirement. But finding the optimal value leads to a NP-Complete problem [4]. Instead, we are using a greedy algorithm that provides us with a sub-optimal solution. The algorithm consists of several steps:

1. Try the list of all addresses, if passed, then stop.
2. Try every single address separately. If it satisfies the QoS requirement, add it to the list of "passed" addresses.

3. Try the whole list of "passed", if it satisfies the QoS requirement, stop. Otherwise, remove the less valuable address (the one that covers the fewest cache misses) from the "passed" list, and repeat step 3.

Our experiments in Section 6 show that this algorithm provides the results that are either optimal or close to optimal in terms of cache miss coverage.

4. SYSTEM IMPLEMENTATION
In Section 3 we described the overall structure of the system we implemented. This section provides more details on how we achieved this.

4.1 Profile Collection
In order to collect information about the cache misses, we used the tool Valgrind [28] and its subtool called Cachegrind [19]. The basic functionality of Cachegrind allows to collect general information about all type of misses: level 1 data and instruction caches, level 2 universal cache; reads or writes. With some additional debugging tools it is possible to get this information on a per source code line basis. We extended the Cachegrind tool with a possibility to be more fine grain, and collect cache misses information on a per program counter (PC) basis. This allows more precise predictions. The results of such a run can be saved in a file, and then be fed to the next step of our system - value prediction.

4.2 Value Prediction
In Section 3.2 we described three schemes of prediction that we used in our experiments: Last Value, Stride and Adaptive. The general scheme is presented in Figure 2. In order to be efficient, such a scheme should be implemented in hardware. We simulate these schemes by extending Cachegrind, which collects information from the profile run and uses it to predict the outcome of the cache misses. The output information from such a run has information about the scheme used, total cache miss coverage, and the number of instructions that were skipped, because of lack of confidence in prediction (this applies only to the Adaptive technique).

4.3 Target Selection
The target/address selection algorithm described in Section 3.4 is implemented in the form of Perl script and can be easily adapted to any other more sophisticated selection mechanism.

4.4 QoS templates
The QoS templates are the core of this project. We have several different QoS templates in the form of Perl scripts that can:

1. Compute simple match between output files.
2. Use text as a must-match, and numeric values as single inputs for our general QoS formula.

Users are free to construct their own QoS templates; the only restriction is to obey to the input and output parameters rule described in the project documentation.

5. METHODOLOGY
We performed two classes of experiments. First, we show that our simple techniques for value prediction have reasonably good prediction accuracy in certain benchmarks. Second, we evaluate how many cache misses can be covered while the application is still showing acceptable QoS. There are two tradeoffs that we analyze. The first one is the coverage/accuracy tradeoff between last value and stride prediction on one side, and adaptive prediction on the other side. The second one is the tradeoff between the QoS and the better coverage for a particular benchmark. More precisely, we investigate how much greater performance can be achieved by trading some precision.

For both classes of experiments we test a subset of benchmarks presented before: SPEC2000 [26] and Mediabench [12]. We make our choice based on their cache miss ratio and some intuition about their potential tolerance to errors that is especially crucial for the second type of experiments. We have separate experiment for L1 and L2 data caches.

5.1 Experimental Environment
For our experiments we used Intel® Xeon® CPU E5345 with 8 cores @2.33GHz desktop. It has L1 Data cache of 32 KB, 8 ways, 64 bytes/line; L1 Instruction cache of 32 KB, 8 ways, 64 bytes/line; L2 cache of 4096 KB, 16 ways, 64 bytes/line. The cache parameters for simulations were chosen to be the same with LRU policy for replacement.

Because of the nature of our approximated calculations, all benchmarks in our experiments were executed to the full completion.

5.2 Profiling and Simulation
In order to simulate cache behavior and collect profile information we chose Valgrind [28] tool. Valgrind is a dynamic binary instrumentation tool that allows to analyze every executed instruction. Since it is easy, flexible to use, and has the initial version for cache behavior simulation, we chose it as a primary tool for our experiments.

5.3 Benchmarks
In this section we provide high-level information about the benchmarks we chose to test with our system. These benchmarks were used to evaluate QoS and performance tradeoffs. We discuss their inputs, outputs and also the QoS metric that was applied in order to evaluate the correctness of the generated outputs. More benchmarks were used to evaluate the prediction quality; their description can be found in [21, 26, 12].

5.3.1 Artificial Neural Network: Art
Description: 179.art is the Adaptive Resonance Theory 2 (ART 2) neural network and is used to recognize objects in a thermal image.
**Table 2: SPEC2000 and Media Benchmarks Characteristics: L1 and L2 cache.**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>L1 Cache</th>
<th>L2 Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Art</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neural Network</td>
<td>train</td>
<td>ref</td>
</tr>
<tr>
<td></td>
<td>MPKI</td>
<td>Instr. (x10⁶)</td>
</tr>
<tr>
<td>SPEC2000</td>
<td>95.53</td>
<td>2</td>
</tr>
<tr>
<td>L1 cache</td>
<td>29.41</td>
<td>117</td>
</tr>
</tbody>
</table>

| Equake | | |
| Earthquake simulator | train | |
| | MPKI | Instr. (x10⁶) |
| SPEC2000 | 6.55 | 18 |
| L1 cache | 20.22 | 21 |

| MediaBench | L1 cache | |
| | Mpeg2dec | Mpeg2encode |
| | (decode mpeg video) | (decode mpeg video) |
| | original | extended | original | extended |
| | MPKI | Instr. (x10⁶) | MPKI | Instr. (x10⁶) | MPKI | Instr. (x10⁶) | MPKI | Instr. (x10⁶) |
| | 0.40 | 19 | 0.14 | 202 | 5.95 | 1 | 6.11 | 35 |

**Inputs:** In our experiments we used both train and reference data.

**Outputs:** The output data consists of the confidence of a match between the learned image and the windowed field of view. In addition, each F2 neuron’s output is printed. After the entire field of view is scanned the field of view with the highest confidence of being a match is output.

**QoS:** For train input we use "all-or-none" QoS meaning that we only accept the exact output of the non-approximated version. For reference input we use our general QoS formula with text output as a must-match and numeric output as single elements in formula. All elements are weighted equally and the QoS limit that was used for experiments is 0.2.

**5.3.2 Partial Differential Equations: Applu**

**Description:** 173.applu is the Computational Fluid Dynamics and Computational Physics PDE’s (Partial Differential Equations) solver.

**Inputs:** In our experiments we used train data.

**Outputs:** The program is capable of automatically verifying whether a given run conforms to the specification of the benchmark by using internally stored reference solutions. The output indicates whether or not the run was successful in meeting the requirements of the verifications tests. To conform to the specification of the benchmark, a run should successfully pass all three verification tests. Failure in any one or more tests indicates non-conformance with the specifications.

**QoS:** We use our general QoS formula with text output as a must-match and numeric output as single elements in formula. All elements are weighted equally and the QoS limit that was used for experiments is 0.05.

**5.3.3 Earthquake Simulation: Equake**

**Description:** 183.equake simulates the propagation of elastic waves in large, highly heterogeneous valleys.

**Inputs:** In our experiments we used train data.

**Outputs:** The output file contains a case summary with seismic source data and reports the motion (displacements) at both the hypocenter and epicenter of the earthquake for a predetermined number of simulation timesteps. The output file format is ASCII.

**QoS:** We use our general QoS formula. No text presented in the output, only numeric data that is treated as single elements in the formula. All elements are weighted equally and the QoS limit that was used for experiments is 0.2%

**5.3.4 MPEG encoder: Mpeg2encode**

**Description:** Mpeg2encode converts an ordered set of uncompressed input pictures into a compressed bitstream compliant with ISO/IEC 13818-2 DIS [7].

**Inputs:** In our experiments we used the original input provided with the benchmark and also an extended input that consisted of 359 ppm images of 1.1MB each.

**Outputs:** The output of this benchmark is a video in mpeg format and also a text file that contains statistics for the conversion.

**QoS:** We use a special QoS formula that evaluates the difference between the video produced by the "approximate" application and the original video output. The formula that we use corresponds to the Mean Squared Error of two mxn images I, K which is given by the following equality [6]:

\[
QoS = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \left( \frac{I(i,j) - K(i,j)}{max(I)} \right)^2
\]

In order to apply the formula, the output videos are converted to frames and the corresponding images are compared one by one. The final QoS is the average of the individually computed QoS. The video is more heavily weighted than the generated file that contains statistics and in our experiments the QoS limit was set to 0.015%.

**5.3.5 MPEG decoder: Mpeg2decode**

**Description:** Mpeg2decode converts a compressed bitstream compliant with ISO/IEC 13818-2 DIS [7] to an ordered set of uncompressed images.

**Inputs:** In our experiments we used the original input provided with the benchmark and also an extended input; a video of size 1.7MB.

**Outputs:** The output of this benchmark is numbered ppm images that correspond to the frames of the input video.

**QoS:** The QoS formula that was used for the evaluation
of the quality of the output images was the same as in Mpeg2encode. The QoS limit was set to 0.015%.

6. RESULTS

6.1 Value Prediction Quality

The first class of experiments we performed was evaluating the quality of three prediction schemes without feeding our prediction result to the executed application. It showed the potential quality of different schemes in Figure 4(a) as well as coverage in Figure 4(b).

The results in Figure 4(a) and Figure 4(b) are for L1 cache misses. As we can see, Last Value and Stride prediction have similar prediction quality for most of the benchmarks, except for ammp, art and gzip. For these three benchmarks Stride is noticeably better. At the same time the prediction quality of Adaptive scheme is much higher; it can be as good as 100% in case of applu, but it comes with the decreasing coverage, since we only predict instruction addresses with some prior confidence. In general, Last Value prediction quality is in the range 3% - 86%, Stride - 10% - 87%, and Adaptive - 31% - 100%.

The results in Figure 5 are for L2 cache misses.

6.2 Value Prediction Simulations

The second class of experiments show the benefits that our system can provide to every benchmark when applying speculative prediction. The results in Figure 6 show how much L1 cache miss latency can be covered using our QoS definition and search algorithm. Every bar consists of correct prediction percentage, misprediction percentage and cache misses that we do not attempt to predict. The first two together give us the total amount of miss penalty covered, because even if we mispredict the value, we still cover its latency.

Figure 6 has the results for benchmarks with low, average, and extremely high MPKI (misses per one thousand instruction) rate. For art benchmark the MPKI rate is almost 100, and our system was able to cover the latency for 25% of them on reference data, and 40% on train data. What is interesting for art is that we have almost identical coverage for Last Value/Stride, and Adaptive schemes, but we got it with different amount of steps in search algorithm. For Last Value and Stride we need 25 runs to find the optimal configuration, while Adaptive scheme needs only one run, since it can dynamically skip instruction addresses that cannot be predicted well. The amount of correctly predicted misses is always better for Adaptive scheme.

For applu benchmark Last Value and Stride have the same coverage (47%) and quality (20% correct), but Adaptive scheme has significantly lower coverage (11%), although it achieves 100% accuracy. The main reason for this is that mispredictions not always cause problems to the application. Our Adaptive scheme conservatively skipped “badly” predicted addresses, but they did not deteriorate the QoS. As a result, our search algorithm left them in the final configuration for Last Value and Stride predictors.

As expected, media benchmarks are a very good domain for our approach. They have similar impressive coverage: 87% - 99% for Last Value and Stride, and 48% - 82% for Adaptive. The relative quality is better for Adaptive scheme. If we consider only extended inputs, which are more representative, then Adaptive scheme is 15%-17% behind the two more aggressive schemes in coverage. But the search algorithm will only need one step to find the optimal configuration, while other schemes need more than 20 steps.

Covering the L2 cache miss latency can bring even higher improvement, since the typical L2 latency is much higher than L1 latency. Our experiments for L2 cache show that similar results, both in prediction quality and coverage can be achieved for L2 caches.

To show the effectiveness of our system in covering L2 cache misses we took two benchmarks: applu and equake (Figure 7). Media benchmarks are not studied since their MPKI level was relatively low. For applu Last Value and Stride can cover 56% latency of all misses with acceptable QoS, Adaptive can cover 17% of latency with almost perfect prediction accuracy. The reasons for this behavior are similar to L1 cache miss prediction that is presented above.

The equake benchmark proved to be much more sensitive to errors than other benchmarks, so our improvements with it are limited to 12% with Adaptive scheme of prediction, and 7% and 10% for Last Value and Stride predictors(Figure 7).

6.3 QoS Search Algorithm in Action

Our QoS search algorithm can automatically identify and filter instruction addresses that we can not predict well enough to keep an acceptable QoS measurement. Figure 8(a) shows the initial step of this algorithm for Mpeg2encode benchmark and L1 cache with Adaptive scheme of prediction. Every single bar corresponds to a single instruction address in the initial configuration file (the total amount is 20). The more valuable targets, the ones that are responsible for most of the misses, come first. QoS limit allows to filter single addresses that produce unacceptable QoS by itself.

In Figure 8(a) we have one address with unacceptable QoS when QoS limit is $10^{-3}$. If we ask for a better QoS, we can require i.e. $QoS_{limit} = 10^{-4}$, then 2 addresses are going to be excluded at this step. Since the QoS for different addresses is not additive, we can not expect the total remaining error to be the sum of single errors, so we continue our search algorithm with a subset of addresses after initial filtering. In the case of mpeg2encode the resulting configuration file has 18 addresses for $QoS_{limit} = 2 \times 10^{-4}$.

A similar stage of the search algorithm is shown for equake.
Figure 4: Value Prediction Coverage and Quality for Different Schemes, L1 cache.

Figure 6: L1 Cache Miss Prediction Coverage.
benchmark (Figure 8(b)), but on L2 cache and with Last Value prediction algorithm. Here we set the acceptable QoS to $QoS_{limit} = 0.2$, and, as we observe, seven addresses have acceptable QoS, including three with 0 QoS (exact match of the outputs). After merging these addresses back to the configuration file we get unacceptable QoS again, and 3 more iterations are needed to find a subset of four addresses that satisfy the initial requirement. It is interesting to see the initial QoS, before algorithm that finds the optimal configuration file was applied. Figure 9 shows the difference in the initial quality for different prediction schemes on mpeg2encode benchmark (random frame was used). Stride prediction is much less accurate, and has lower QoS (Figure 9(a)) without search algorithm. In contrast, Adaptive scheme (Figure 9(b)) is very close to the original one (Figure 9(c)) even when the default configuration file that is the output of the profiling stage is used.

### 6.4 Performance Improvement

The Valgrind [28] tool we use for simulation does not have a full timing model, so we cannot measure the exact speedup caused by covering cache misses for different benchmarks. However, we can approximately evaluate the speedup by applying a simple model in which every instruction takes one cycle, L1 cache miss takes 20 cycle and L2 cache take 400 cycle to service; CPU is in-order machine. In such a model, art benchmark with reference data that has 25% miss latency covered for L1 and MPKI = 99 (See Table 2), will improve by 20%. The geometric mean of speedup on all benchmarks for L1 is 12.5% for Stride prediction, 12% for Last Value, and 10% for Adaptive. The range is from 1% for mpeg2encode on original input to 37% for art on reference input. For L2 cache on applu we get 69% speedup for Last Value and Stride, 14% for Adaptive; on equake we get 7% for Last Value and Stride, 12.5% for Adaptive scheme.

### 7. RELATED WORK

Researchers have previously studied the possibility of improving different application metrics, such as performance, fault-tolerance and energy consumption, using the approxi-
mate computing idea. But to the best of our knowledge, our system is the first one that is trying to use simple, recovery-free and fully automatic way to improve application performance and energy consumption by speculatively predicting cache miss values.

The closest works are the Green framework system [2] and QoS profiling [16, 5] in the way that we also propose the tradeoff between QoS and performance. However, our approach is significantly different in several aspects. Both Green framework and QoS profiling concentrate their efforts in skipping loop iterations; Green framework is doing this only to the "tail" of the loop, QoS uses perforation that allows to skip almost any loop iteration. We agree that this can lead to a significant speedup for certain applications, but both approaches impose significant restrictions on the loop structure (QoS needs canonical loop in LLVM [27]) or even the return function value (Green only works with numerical values). On the contrary, our approach does not impose any limitations on the loops and functions, and thus it is more generic.

Green framework has a more fine-grain QoS model, so every interesting loop and function can be instrumented, but this comes with the cost of user-needed intrusion to the code and provision of QoS metric for these functions/loops. On the other hand, our approach is as autonomous as possible, measuring the whole application’s QoS and providing several QoS templates. The results from both previously implemented systems showed that they are applicable to very specific domains, most commonly when the main hot routine in the last iterations is just improving its quality/precision, such as in the cases of computing sin, cos, log, exp or doing search. Our system is potentially applicable to a wider domain of applications that have considerable amount of cache misses.

Our approach of measuring the QoS in the case of text outputs can be considered as a slight generalization of the term distortion introduced in Rinard’s previous work [22, 23]. The difference is that we are assigning weights to every output, depending on their significance - an application might produce several different outputs, e.g. a text, a video, a configuration file - and do normalization with the sum of the two values so that the QoS is in the interval [0,1].

Another aspect of our work is related to value prediction mechanism, which has been investigated before [25, 11]. Lipasti et al. introduced value locality [11] which is the likelihood of a previously seen value to be met again in storage. They introduced what we call Last Value prediction scheme, and also classified loads into ”constant”, ”predict” and ”don’t predict” categories. In our approach we are not trying to predict all loads, since the prediction can be on the critical path, but rather only the ones that are missing from the cache, and, hence, are much more beneficial if predicted. Sazeides et al. analyzed the predictability of data values [25]. They were trying to predict not just loads, but also the outcome of all types instructions. That makes their approach more general, but highly complicated to be implemented, because
of the hardware constraints. On the contrary, our approach requires only minor hardware changes. This work also introduces several other techniques for predicting values including stride prediction and context-based prediction. The variation of the first one with confidence counters is used in our work. Again, we only focus on cache misses, since their prediction is much more crucial for the overall application performance than simple operations, such as additions. And value prediction was not a goal, but rather a technique to achieve acceptable approximate computing.

Prefetching both in hardware and software has been proposed many times in the past as a method of decreasing the average cache miss latency [18, 17, 31]. Here we will refer to the works that are the closest to the spirit to our approach. Conte et al. introduced recovery-free value prediction mechanism [30] that improves memory-level parallelism by value prediction and value speculative execution. Their model for prediction is simple stride with confidence counter that is very similar to our scheme, but the key difference is in the mechanism. The authors used value prediction to only generate efficient prefetch requests; thus, their system is recovery-free due to the fact that the application’s architectural state is not changed. On the other hand, our approach is speculative and can cause the program to produce a wrong result or even crash. But the benefits of our approach are that we do not do speculative preexecution, nor consume bus bandwidth with additional prefetch requests. Moreover, our technique is not limited to pointer-based benchmarks and is orthogonal to any prefetching techniques. Hence, it can be easily used in parallel with them.

Lewis et al. [9] addressed the problem of initialization misses to the heap by totally covering them with the Allocation Range Cache hardware mechanism. In this work authors showed the criticality of those misses: 23% of all miss traffic was caused by initialization misses in a chosen subset of SPEC CINT2000 [26]. Covering these misses results in 41% performance improvement. Since our approach also allows to cover most of the initialization misses (their misprediction does not cause any failures), we can expect at least similar performance improvement by our approach. On the other hand, our approach is not limited to the initialization to the heap only; so, we can cover Fortran initialization misses, or C/C++ arrays. In addition to that, we can cover misses that happen not only during initialization, and that makes our approach more general.

Approximate computing is not limited to predicting cache misses or skipping loop iterations [2, 16]. This idea can be applied in branch prediction [13], improving fault tolerance [10], power and energy consumption [14], and automatic parallelization [15]. Ansel et al. [1] introduced a new language and compiler called Petabricks that has the multiple algorithmic implementations as a natural way to generate high-performance code. This activity shows the importance of approximate computing both for research and for industry.

8. CONCLUSIONS

In this paper, we propose a system that allows to get benefits of the "non-strict" correctness definition by speculatively predicting values when cache misses occur. This system can automatically find a sub-optimal configuration file for which the application performs better than originally and also has acceptable QoS. We constructed several QoS templates that can be used for a variety of different benchmarks, and investigated the tolerance of certain applications in the presence of significant amount of mispredictions.

Our evaluation shows that with three simple prediction techniques we can achieve considerable decrease in cache miss latency for L1 and L2 caches, which results in application speedup and energy savings. The experimental results prove that our system is applicable to a variety of benchmarks like AI, image/video processing and PDE solvers. In particular, we can cover up to 99% L1 cache misses for Mediabench, and 56% of L2 cache misses for applu that can lead to as high as 69% speedup.

9. SHORTCOMINGS AND FUTURE WORK

While a significant progress was achieved during this project, there are certainly several items that need more investigation and improvement. First, we would like to increase the quality of our value predictors, because this is the key factor in producing acceptable QoS after speculation. Context-sensitive predictors [25] is definitely one of the options. Second, it is interesting to know in which cases our approach is more or less beneficial. This requires an extensive analysis of several benchmarks with different prediction quality. Third, we would like to test more benchmarks, especially for L2 cache, to prove that our approach is general enough. PARSEC [3] and olden [24] benchmarks, as well as SPEC2006 [26] can be the source of new opportunities. And, last but not the least, an ambitious goal is to find the boundaries of such speculative approaches. Specifically, we would like to get a measure of applications’ tolerance to the speculative predictions without recovery.

10. ACKNOWLEDGMENTS

We would like to thank Dr. Omur Mutlu, Dr. Todd Mowry, Evangelos Vlachos, Vivek Seshadri, Lavanya Subramanian for their permanent help and valuable feedback during this project. We would also like to thank the whole SAFARI research team for their contribution in the form of productive discussions.

11. REFERENCES


York, NY, USA, 2008. ACM.


