# Characterizing the Performance of Value Prediction using Statistical Simulations

Peter Giese,
Dept. of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, Canada T6G 2V4
E-mail: pgiese@trlabs.ca

***Abstract* – While the speedup potential of value prediction (VP) is appealing, value locality, predictor accuracy, and hardware restrictions place practical limits on achievable performance. In this study, a statistical simulation approach is used to characterize the upper limits of VP performance. By emulating the behaviour of value predictors, the performance of VP is studied under best-case operating conditions for a 5-stage pipeline possessor with register forwarding and separate functional units for prediction validation. The simulation program developed in this study uses a synthetically generated instruction stream to model a generic integer application. Experiments are conducted over a wide range of value locality and predictor accuracy values. An upper limit of 10-30% was found to exist when VP is applied to both LOAD and ALU instructions. An upper limit of 80% was found to exist for LOAD only prediction with an average L2/MEM latency of 100 cycles. Statistical simulation results revealed that the upper limit of VP speedup is not significantly higher than full execution-driven simulation results reported by other researchers. The rather low upper limits of VP performance found in this study may explain why there is still no commercial processor that employs VP as a general speculative execution technique.**

## I. INTRODUCTION

The underlying persistence of true data dependencies between instructions is a significant barrier in extracting parallelism out of sequential programs. Value predication (VP) is a relatively new approach used to increase instruction level parallelism (ILP) by overcoming true data dependencies. It is used to predict the result of instructions before they are executed and allows speculative execution to take place. VP techniques rely on value locality, which is defined as the likelihood of a previously seen value recurring. Value locality is closely related to temporal and spatial locality concepts, which are used by address prediction and caching techniques. Early value locality experiments demonstrated that many instructions that write to general-purpose registers do exhibit value locality [1]. The existence of value locality provides an opportunity to decrease data dependency between instructions residing within a pipelined processor.

VP is a hardware-based approach that predicts the outcome of instructions that write to registers. Techniques involving VP are similar to the techniques used for dynamic control speculation (branch prediction) and dynamic address speculation (pre-fetching). While operating separately, the techniques of trace caching, value prediction, and cache pre-fetching each reside in a specific portion of the pipeline and each technique plays a different role in improving ILP. These three techniques all exploit statistical methods to improve performance and are based on different concepts of locality. Figure 1 illustrates where in a pipelined processor VP takes place [2]. Since both the execution and memory stages of a pipelined processor are significant sources of latency they are both prime candidates for VP. Arithmetic operations, typically non-trivial operations, can take several cycles to complete creating stalling conditions for data dependent instructions. At the same time cache misses associated with LOAD instructions can also create stalling conditions.

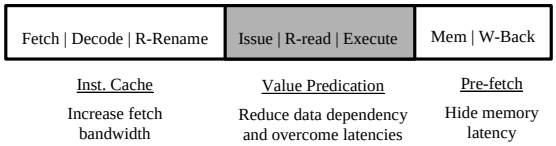



Figure 1 – Placement and role of value prediction

The speedup potential of VP depends on many factors. The main factors influencing the performance of VP are: i) frequency of long latency instructions; ii) level of value locality within instructions; iii) level of data dependency between instructions; iv) accuracy of value predictions, and v) hardware limitations. An established body of literature exists that studies the influence of these factors in different processor architectures. A good introduction and discussion are presented in [3], [4], and [5]. Research into the development of context-based value predictors for LOAD and ALU instructions has shown that one can predict instruction write registers with a high degree of accuracy. The focus of current research activities is on finding new ways of improving the accuracy of VP by detecting data dependency during instruction pre-fetching prior to execution, as well as using VP to overcome longer memory latency present with LOAD instructions. Despite the speedup potential of VP to overcome data dependencies, VP has not been widely implemented in contemporary processors [6].

In this work, the focus is on characterizing the upper limit of speedup when VP is applied to both ALU and LOAD instructions. The objective is to determine the speedup potential of VP by emulating the behaviour of value predictors under best-case operating conditions. Since

the performance of VP is governed by statically behaviours present within the instruction stream a statistical simulation approach was used to evaluate the upper limits of VP performance. Simulation experiments were conducted in this study using a synthetically generated instruction stream that passes through a 5-stage pipeline. By controlling the statistical distribution of instructions types, level of data dependency, level of value locality, and emulating predicator accuracy the overall speedup of VP is characterized. Experimental results are based on a generic integer application that is simulated by 10 million synthetic instructions.

The remainder of this paper is organized as follows: Section II presents a review of statistical simulation. Section III reviews value locality and value prediction. Section IV outlines the simulation approach applied to a classic 5-stage pipeline processor. In Section V, simulation results are presented and discussed. Finally, Section VI concludes the paper and discusses future work.

## II. STATISTICAL SIMULATION

In between detailed functional simulations and analytical model lies statistical simulation. In [7], a comparison of different simulation and modeling techniques is discussed. Statistical simulations can overcome many shortcomings of full trace-driven or execution-driven simulations when studying the performance of microprocessors. Using a statistical simulation approach allows quick and comprehensive analysis of a microprocessor in the early stages of design. It can provide good first-order performance estimates without the time consuming requirement of simulating complex functions residing within a microprocessor. Several research efforts have approached the problem of performance simulation by statistical means, most notable are [8] and [9]. These and others have demonstrated that by using a synthetically generated instruction stream, coupled with functional emulation of pipeline processes, one can produce performance results that are within 4-8% of cycle-by-cycle simulations [8][10]. These findings indicate that VP performance can be accurately characterized by using statistical simulations.

The statistical simulation approach taken in this study is based on the approached taken in [9], where the program trace is replace by a synthetically generated instruction stream governed by a pre-defined set of profiling parameters. As apposed to [8], which uses an execution model to simulate the instruction stream. To ensure a high level of accuracy, several profiling parameters are required to define the synthetically instruction stream, as well as behaviour of the microprocessor. A summary of the instruction profile parameters discussed in [8] and [11] is presented in Figure 2. In this work, profiles are created for all parameters mentioned in Figure 2 with the exception of instruction stream behaviour.

As a first-order design approximation used to simplify the developed of a statistical simulation program, only four generic types of instructions are simulated. The four instruction types are shown in Figure 3A. As shown in Figure 3B, a random number generator Alfa is used to generate the type of instructions and a second random number generator Beta is used to randomly generate register names thereby creating data dependency.

Also illustrated in Figure 3B are other random variables used to generate instruction locality, as well as emulate predictor accuracy, for both LOAD and ALU instructions. A random number generator Gamma is used to generate locality values and a random number generator Delta is used to simulate predicator accuracy. Finally, branch taken events are model using a random number generator Epsilon.

| Profile | Characteristic |
|---|---|
| Instruction Mix | Percentage of operations |
| Instruction Dependency | Register and sequence |
| Instruction Distance | Instructions between |
| Branch Behavior | Branch predictability accuracy |
| Data Stream Behavior | Miss Rate for D-cache |
| Instruction Stream Behavior | Miss Rate for I-cache |

Figure 2 – Type of profiles used in statistical simulations

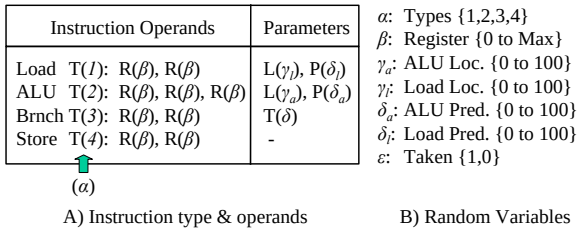



Figure 3 – Simulated instructions, operands and random variables

Literature describing statistical simulation highlights the importance of profiling instruction distance, but it does not provide a universal set of templates for integer and floating-point applications. Ideally, one should include all instruction distances, as defined by the distance matrix shown in Figure 4, when generating a synthetic instruction stream.

| | Load | ALU | Branch | Store |
|---|---|---|---|---|
| Load | | | | |
| ALU | | | | |
| Branch | | | 4-10 | |
| Store | | | | |

Figure 4 – Instruction Distance Matrix

As a minimum, branch distance needs to be defined because it plays a critical role in defining control-based blocks within the instruction stream. By using an average blocks size of 4-10 instructions one is typically modeling integer applications, where as floating-point applications tend to spend most of their time in larger loops [12]. To

ensure a high level of accuracy, distance values for all row-column combinations in the distance matrix should be defined. However, what are the appropriate distance values for different applications and what are the appropriate probability functions remains an open research question. Conducting detailed analysis to determine the appropriate distance matrix for different benchmarks is beyond the intended scope of this work.

A statistical simulation approach was used because it allows the potential of VP to be evaluated over a wide range of operational parameters. This enables performance to be simulated under both oracle and best-case operating conditions without incurring large program development times.

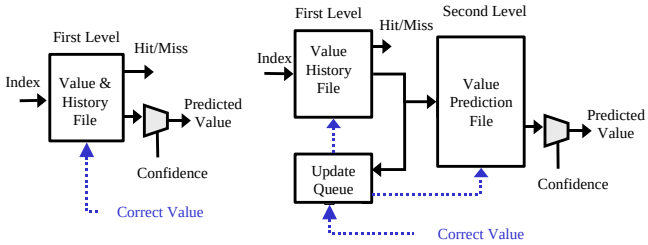
## III. VALUE PREDICTION

VP and instruction reuse (IR) are two techniques that have been proposed to overcome data dependencies. VP is a speculative technique that attempts to predict the results of instructions, whereas IR recognizes whether an instruction has been previously executed and can be committed. VP has been shown to be the more general technique and it possesses the ability to exploit value locality to a higher degree [13].

The role of a value predictor is to predict the result of an instruction based on past history. There are two main categories of predictors: computational and contextual [14]. Computational-based predicators use deterministic approaches to detect strides and computed future values. A context-based predictor uses a particular repeating pattern in a history file to predict the next value in a sequence. Context-based predicators are commonly used because they are accurate and theatrically they can predict any repetitive sequence. Variations of the Finite Context Method (FCM) [15] have been successfully used for predicting LOAD and ALU instructions.

Over the years predictors have evolved from single table architectures to two-level table architectures as illustrated in Figure 5, which is based on material presented in [3] and [16]. Earlier predictor implementations used history value and stride approaches based on a single-stage architecture as shown in Figure 5A. Last value predictors used the last previously seen value to predict the result of an instruction. Stride predictors calculate stride patterns to predict the result of an instruction. These two approaches typically use the instruction address (PC) as the primary index into the history table. In the two-stage architecture shown in Figure 5B, a value predictor hardware unit consists a value history table (VHT) and the value predation table (VPT). The current generation of FCM predators typically use four last seen values, which are stored in the VHF table. These four history values are used to compute an index into VPT using different sequences or patterns found at the indexed VHF location. The best performing predictors in the current literature are hybrid FCMs [17][18]. The accuracy of hybrid FCM predictors varies between about 10 to 90%, depending on the application, type of value locality, and predictor table size.

A VP does not exist for every instruction. One reason is that it may be the first time the instruction is issued and not enough history has been collected. Another reason is that that value locality for an instruction is not high enough to make a prediction. Early experimental results in [1] indicate that LOAD instructions exhibit high levels of register value locality, whereas ALU instructions exhibit lower levels. Value locality depends on the software programming style, the type of application, and on the type of instruction. Typically, value locality for LOAD instructions is in the order of 80%, whereas value locality for ALU instructions is 20-60% (with floating-point instructions exhibiting the lowest levels of value locality). To keep the number of miss-predictions to a minimum almost all value predictors incorporate some form of confidence estimator to identify when no locality is present and when predictions are likely to be incorrect. Blocking predictions with low confidence is important as it avoids unnecessary instruction rollbacks due to bad predictions. Without restricting predictions to those with only high levels of confidence it is possible that VP can decelerate the processor instead of speeding it up.




A) Single stage look-up B) Two stage look-up

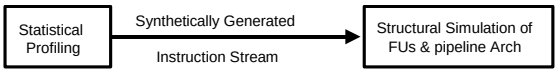
Figure 5 – Predictor architectures

It is worth mentioning that a substantial body of literature exists for hardware-oriented implementations of VP. Semantic-based value prediction, which involve software only or joint hardware-software collaboration, is briefly discussed in [14] and represents an important avenue of future research.

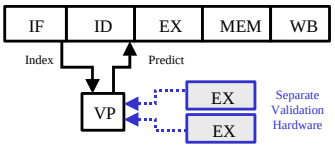
## IV. PIPELINE SIMULATION

A computer program (C-program) was written that simulates the interaction of instructions as they flow through a 5-stage pipeline with forwarding capabilities. The computer program is presented in Appendix A. As illustrated in Figure 6A, profiles are used to generate a synthetic instruction stream. Once an instruction has been randomly generated it progresses through each stage of the 5-stage pipeline, which is simulated using the developed computer program. VP is performed for both ALU and LOAD instructions. As depicted in Figure 6B, VP is performed in the ID-stage of the pipeline and separate ALU/LOAD functional units (FUs) are used to support

speculative execution. This pipeline architecture allows the full potential of VP to be studied in a pipelined processor.




A) Statistical Instruction Stream




B) 5-Stage pipeline with parallel validation

Figure 6 – Simulated pipelined processor

Each stage of the pipeline takes one cycle to perform except when a long-latency ALU event is encountered or when a LOAD instruction causes a cache miss. When a long-latency ALU event occurs the instruction remains in the EX-stage of the pipeline for the defined latency period. A L1 cache miss forces the LOAD instruction to remain in the MEM-stage for the defined latency period. Latency events are randomly generated, while the number of latency cycles are predefined numbers. Data dependences between instructions is simulated by keeping track of write and read dependency (RAW) as instructions pass through the ID-stage of the pipeline. Registers usage is flagged using a register status array. When a RAW dependency is detected, the instruction that is reading a depended register remains in the ID-stage creating a stall condition. Branch decisions are made in the ID-stage of the pipeline and a one-instruction delay slot is simulated. A branch not-taken decision will result in an instruction purge, creating a one-cycle bubble.

Separate EX and MEM FUs perform speculative execution when VP is activated and no limit is placed on the number of FUs. This architecture allows in-order execution to proceed through the 5-staged pipeline when a predication is made. Included in the simulator is an instruction window that stores all instructions residing within the pipeline and all instructions that have been speculative executed as a result of a value prediction. Committed instructions are stored in an instruction window. No limits are placed on the size of the instruction window. When a prediction has been verified to be correct the predicted instruction and all speculatively committed instructions are removed from the instruction window. In the event that an incorrect prediction is made, a rollback operation takes place and the first dependent instruction is placed into the ID-stage of the pipeline, this time with the correct register value. Different rollback delay scenarios are simulated: i) zero cycle delay to simulate ideal hardware, and ii) one cycle delay to simulate realistic hardware behaviour. Instruction stream prefetching and memory LOAD/STORE conflicts were not modelled in the computer program. The various simulation parameters are summarized in Figure 7.

| Profile | Characteristic |
|---|---|
| Inst. Occurrence | Random Distribution |
| Load | 22% |
| ALU | 50% |
| Branch | 17% |
| Store | 11% |
| ALU Latency | 1 Cycle |
| | Mult/Div: 5 Cycles Aver. |
| | Latency Occurrence 15% |
| Inst. Dependency | Register RAW |
| Data Dependency | Variable: 72% to 0% |
| Inst. Distance | 6 Between Branches |
| Branch Behavior | Delay Slot |
| Data Stream Behavior | L1 No Miss:1 Cycle |
| | L2/Mem: 10 Cycles Aver. |
| | Miss Rate 5% |

Figure 7 – Simulation parameters

The parameters Gamma and Delta (see Figure 3) are used to emulate value locality and predictor accuracy, respectively. These two parameters can be set to unique values for ALU and LOAD instructions. As mentioned in Section II, the parameter Beta simulates data dependency. In the case of a single-issue pipeline, which is simulated by the computer program, the level of data dependency can be defined as a percentage involving the number of depended read instructions that immediately following a write instruction divided by the total number of instructions. In all experiments, the number of synthetically generated instructions is 10 million. This is a simplified metric for measuring true data dependencies that applies to a single-issue and in-order pipeline. A more elaborate metric is required to measure data dependency in a multi-issue and out-of-order execution pipeline.

## V. RESULTS AND DISCUSSION

A set of experiments was conducted to characterize the upper limits of VP performance under near-ideal operating conditions. Parameter settings used to emulate data dependency (DD), value locality, and predictor accuracy are all biased towards studying VP performance under best-case conditions. Performance is measured it terms of $VP_{Speedup}$ defined as the ratio of $CPI_{Without\text{-}VP}$ divided by $CPI_{With\text{-}VP}$, where $CPI_{Without\text{-}VP}$ denotes the baseline case without VP and $CPI_{With\text{-}VP}$ denotes the test case with VP.

### *A. Baseline Without VP*

A set of experiments were conducted without VP and with the branch taken parameter set to 90%. The cycles per instruction (CPI) results shown in Figure 8 indicate that the branch delay slot causes a constant number of control hazards across all DD values, which range between a maximum of 72% and a minimum of 0%. Because control hazards can be treated as a constant, they are not included in VP experiments (100% taken is the default setting used in all subsequent experiments). In Figure 8, one can see that at high DD values the number of latency-based hazards is

equal to the number of DD-based hazards. Also evident is a constant number of latency-based hazards.

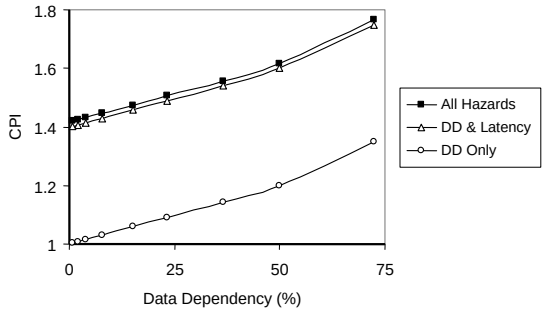



Figure 8 – Control, latency, and data hazards present without VP

### *B. VP With/Without Rollback Penalty*

A series of experiments were conducted under the condition of maximum DD (72%), while varying value locality and predictor accuracy. Differ categories of value locality were simulated: i) perfect locality with 100% locality for both ALU and LOAD instructions; ii) upper limit of locality with 60% for ALU and 90% for LOAD instructions; and finally, iii) low level of locality with 20% for ALU and 70% for LOAD instructions. These difference categories of locality were selected to characterize VP performance under different statistical conditions. Figure 9 plots $VP_{Speedup}$ versus predictor accuracy, which ranges from perfect accuracy (100%) to poor accuracy (50%).

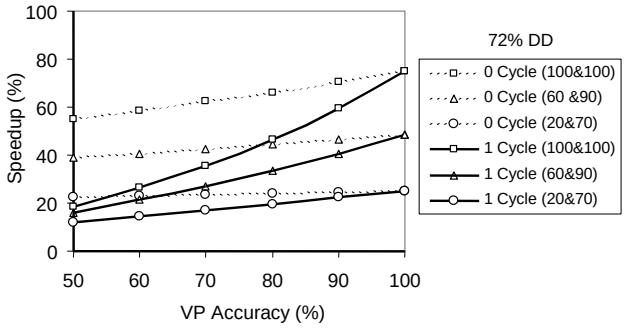



Figure 9 – VP with 0-cycle and 1-cycle rollback delay

While it is possible for a predictor to make a prediction in one cycle, it is unrealistic to assume that prediction validation and rollback can all take place in zero cycles. Shown in Figure 9 is $VP_{Speedup}$ performance with no rollback penalty and with a rollback delay of 1-cycle. Rollback penalties greater than 2 did result in $VP_{Speedup} < 1$ when predictor accuracy is low. As part of a theoretical exercise one can model 100% locality for both ALU and LOAD instructions, however this is also an unrealistic condition for VP. Remaining experiments do not consider 100% locality and include a 1-cycle rollback penalty.

### *C. VP Upper Limit Experiments*

Experiments were conducted to gauge the effect that data dependency has on VP performance, while varying value locality and predictor accuracy. DD values were set to 72%, 50% and 23%. Lower values of DD were not considered to be realistic, as current programming approaches tend to create high rather than low data dependency conditions. Experimental results are show in Figure 10. In this figure the realistic range of predictor accuracy is highlighted (70-90%). Using this figure to characterized the upper limit of VP performance we find that the upper limit of $VP_{Speedup}$ is 40%, where 90% accuracy intersects 72% DD with high levels of locality (60% ALU and 80% LOAD). A more realistic assumption would be to assume 50% DD. With this assumption, VP performance would reside within the shaded area of the graph and can be defined as $10\% < VP_{Speedup} < 30\%$.

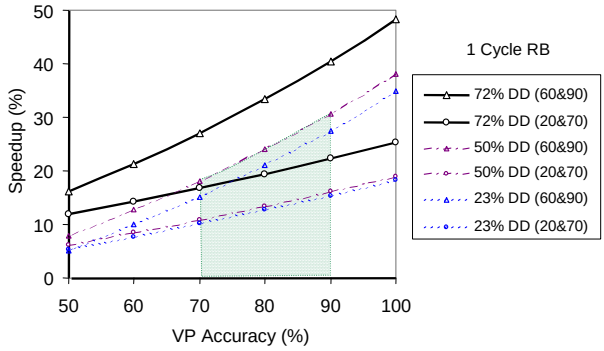



Figure 10 – VP performance versus data dependency

The upper limit prediction of 10-30% is not out of line with other research results that report VP performance improvements to be 5-20% when using full execution-driven situations [6]. It is postulated that $VP_{Speedup}$ results produced by this report are higher than other reported results because: no table limitations exists; near-ideal value locality is modeled; and, no limitations are imposed by control hazards (which would be the case if dynamic branch prediction is used). It was observed that VP performance values reported in the literature are typically for superscalar and multi-threaded processors. Even though these processor architectures are different than what has been simulated in this work the reported average VP speedup values are consistent with the upper limits predicted using a 5-stage processor architecture. This seams to indicate that the upper bound of VP performance found in this study may also apply to other processor architectures. One can postulate that data-driven performance enhancements will always be restricted to a small percentage after ILP has been fully exploited using control-based techniques; no matter what type of processor architect is involved.

### D. *Load only VP*

Overcoming long memory latencies is an attractive scenario for the application of VP. A final set of experiments was conducted to gauge the performance of VP applied only to LOAD instructions. Three memory access latencies where simulated, they are 10, 50 and 100 cycles. It is worth noting these latency values represent the combined latency behaviour of a L2 cache and memory access in the event that a level L1 cache misses. Experimental results are show in Figure 11 for the realistic range of predictor accuracy values. Using this figure to characterize the upper limit of VP performance for LOAD instructions only, we find that the upper limit of $VP_{Speedup}$ is 80%. Once again, assuming a 50% data dependency the upper limit becomes 12% < $VP_{Speedup}$ < 76% for LOAD predictions. Since DD does not play a significant role in LOAD only predictions the upper limit of 80% is used to characterize VP performance.

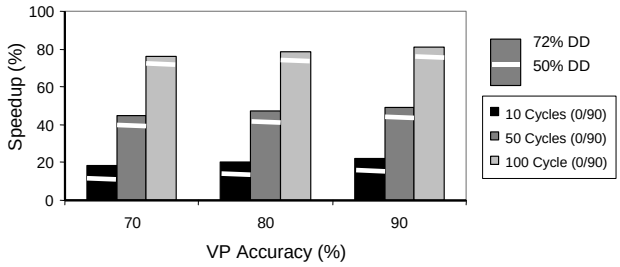



Figure 11 – LOAD only VP for different L2/MEM latencies

In these experiments the instruction widow size was never a limiting factor. In [19], an average speedup of 40% was reported for VP experiments conduced on a multithreaded architecture. While the multithreaded experiments involved larger memory latencies of 1000 cycles (L2/L3/Mem cache model), the authors report that VP performance was ultimately limited by restrictions involving widow table size and/or thread store buffer size. This indicates that the 80% upper limited predicted by this study may be difficult to achieve because of performance restrictions imposed by hardware limitations. Hardware sensitive analysis is required to confirm this conclusion.

## VI. CONCLUDING DISCUSSION

The statistical simulation approached used in this study allowed VP to be characterized over a wide range of data dependencies, value locality, and predictor accuracy values. Simulation of a generic integer application revealed that $VP_{Speedup}$ < 40% under maximum data dependencies conditions in a 5-stage pipeline with parallel prediction validation. Assuming 50% data dependency is more typical, the upper limit of VP performance is predicted to be 10-30%. The experimental results generated in this study are consistent with other research results that report VP performance improvements to be 5-20% for various processor architectures, as well as different integer and floating-point benchmarks. The $VP_{Speedup}$ results generated in this study are higher than other report results because: no limitations are imposed by table sizes; near-ideal value locality and data dependency is modeled; no limitations are imposed by control hazards; and finally, floating-point applications were not studied in the statistical simulation experiments. Even under best-case operating conditions, the upper limit of VP performance is not significantly higher than the results reported by others. The rather low upper limits found in this study may explain why there are still no commercial processors that employ VP as a general speculative execution technique. The simulation results for LOAD only prediction show more promise as 80% improvement in performance can be achieved when the average L2/Mem latency equals 100 cycles.

Statistical simulations conducted under best-case operating conditions enabled the upper limit of VP performance to be quantified, however more work is required to verify the accuracy of the experimental results reported in this study. Benchmark experiments are required to confirm the overall accuracy of the statistical simulation results, which the author estimates to be +/-10%. In addition, further research is required to characterized VP for integer versus floating-point applications. This will also require detailed benchmark analysis in order to fully specify the distance matrix for each type of application. Further research is required to study the application of the statistical simulation approach to superscalar and multithreaded processors. The results presented in this study seem to indicate that the upper limit of VP performance, as characterised in this study, may be representative of behaviour found in other processor architectures. It is postulated that the upper bound of VP performance will not very significantly in superscalar and multithreaded processors.

## REFERENCES


[1] M.H. Lipasti, J.P. Shen, "Exceeding the dataflow limit via value prediction," Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on, pp. 226 – 237, Dec. 1996

[2] C. Cher, "Exploring and Evaluating Control-flow and Thread-level Parallelism," Doctor of Philosophy Thesis, Purdue University, May 2004

[3] F. Gabbay, A. Mendelson, "Using Value Prediction to Increase the power of Speculative Execution Hardware," ACM Transaction on Computer Systems, Vol. 16, No. 3, pp. 234-270, August 1998

[4] B. Calder, G. Reinman, "A comparative Survey of Load Speculation Architectures," Journal of Instruction-Level Parallelism, Vol. 2, pp.1-39, 2000

[5] P. Marcuello, A. Gonzalez, J. Tubella, "Thread Partitioning and Value Prediction for Exploiting Speculative Thread-level Parallelism," IEEE Transactions on Computers, Vol. 23, No.2, February 2004

[6] J. Markovski, M. Gusev, “Why Value Prediction is Limited?” Technical Report PMF-II-01-2004, FNSM, Ss Cyril and Methodius University, Skopje, Republic of Macedonia, January 2004

[7] L. Eeckhout, S. Nussbaum, J.E. Smith, De Bosschere, K., “Statistical Simulation: Adding efficiency to the computer designer's toolbox,” Micro, IEEE, Vol. 23, Issue 5, pp. 26 – 38, Sept.-Oct. 2003

[8] M. Oskin, F.T. Chong, M. Farrens, “Using Statistical and Symbolic Simulation for Microprocessor Evaluation ,” Journal of Instruction-Level Parallelism, Vol. 4, pp.1-39, 2002

[9] L. Eeckhout, D. Stroobandt, K.D. Bosschere, “Efficient Microprocessor Design Space Exploration Through Statistical Simulation,” Proceedings of the 36th Annual Simulation Symposium (ANSS03), IEEE, 2003

[10] S. Nussbaum, J.E. Smith, “Modeling Superscalar Processors via Statistical Simulation,” Proceedings of the 2001 International Conf. on Parallel Architectures and Compilation Techniques pp. 15-24, 2001

[11] L. Eeckhout, R. Sundareswara, Joshua J. Yi, D. J. Lilja, P. Schrater, “Accurate Statistical Approaches for Generating Representative Workload Compositions,” The 2005 IEEE International Symposium on Workload Characterization IISWC, Oct 2005

[12] A. Phansalkar, A. Joshi, L. Eeckhout, L. K. John, “Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites”, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2005), March 2005

[13] A. Sodani, G. S. Sohi, “Understanding the Differences Between Value Prediction and Instruction Reuse,” Proceedings of the 31st Annual International Symposium on Microarchitecture, November 1998

[14] L. N. Vintan, “Value Prediction and Speculation into the next Microprocessors Generation,” Proceedings of the Romanian Academy, Series A, Vol. 5, No. 3, 2004

[15] Y. Sazeides, J. E. Smith, “The Predictability of Data Values,” IEEE, Proceedings of Micro-30, 1997

[16] Y. Sazeides, J. E. Smith, “Implementations of Context Based Value Precitors,” ECE 97-8, University of Wisconsin-Madison, December 1997.

[17] M. Burtscher, B. Zorn, “Hybrid Load-Value Predictors,” IEEE Transaction on Computers, Vol. 51, No.7, July 2002

[18] K. Wang, M. Franklin, “Highly Accurate Value Prediction using Hybrid Predictors,” 30th Annual International Symposium on Microarchitecture, Dec., 1997

[19] N. Tuck, D. M. Tullsen, “Multithreaded Value Prediction”, 11th Int’l Symposium on High-Performance Computer Architecture (HPCA-11), 2005

## APPENDIX A

```
// Value Prediction Simulation
// Peter Giese
// December, 2005

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>
#include <float.h>
#include <conio.h>

#include "valpredv1.h"

void main (int argc, char *argv[]) {

   /* structure for time stamp */

   time_t now_t;
   struct tm now;

   /* Variables related to what input information is provided by the user:
   - usageErr: 1 is program was not properly called
   - logFileNeeded: 1 if the user requires an event log file */

   int usageErr;
   int logFileNeeded = 0;

   /* Simulation variables */

   int SIMSEED;
   int MAXINST;
   int MAXREGISTER;
   int LOADPERC;
   int ALUPERC;
   int BRANCHPERC;
   int STOREPERC;
   int ALUDELPERC;
   int CACHEMISSPERC;
   int BRCHJMPPERC;
   int PREDACC;
   int LOCALITYALU;
   int LOCALITYLD;
   int CycleCnt, InstCnt, WindowCnt, CommitCnt;
   int MaxWindow, MaxWinInst, MaxPipe, WinSize;
   int ALUDelay, CacheDelay, CacheReadCnt, CacheWriteCnt;
   int RanGen, RanAccess, Enhanced, ALUMissCnt;
   int LoadCnt, ALUCnt, BranchCnt, StoreCnt, BrchTakenCnt;
   int LoadPredCnt, ALUPredCnt, PredLdCorrCnt, PredALUCorrCnt;
   int DataDep, BrchHaz, DataHaz, ALULat, MemLat;
   int ExecFlag, OffSet, LoopTest;
   int LdDestPred, LdSrcePred, ALUDestPred, ALUSrcePred, OtherSrcePred;
   int LoadSrceCnt, ALUSrceCnt, OtherSrceCnt;
   int Index, IndexBck, Flag, i, j, s;
   int BrchDisCnt, BrchDisSum;
   int *Pipe, *DataWrite;

   Instruction *Opcode;
   RegStuff *RegisterFile;

   FILE *logfptr;
   FILE *outfptr;

   time(&now_t);
   now = *localtime(&now_t);
   printf("\n VP Simulation  - Successful Execution\n");

   /* Determine possible errors in calling the program */

   usageErr = 0;
   if (argc != 16) usageErr = 1;

   /* Error Message if the program is not called properly */
   if (usageErr) {
      printf (" Usage: valpred.exe ");
      printf ("time [TOPfile] [DEMfile] [PRTfile] [MNDfile]");
      printf (" [OUTfile] [LOGfile]\n\n");
      printf (" Seed: simulation seed (number)\n");
      printf (" MaxInst: Simulation time period (t-units)\n");
      printf (" MaxWindow: Simulation time steps (t-units)\n");
      printf (" LoadPerc: Load Percent (%)\n");
      printf (" ALUPerc: ALU Percent (.%)\n");
      printf (" BranchPerc: Branch Percent (.%)\n");
      printf (" StorePerc: Store Percent (%)\n");
      printf (" ALUDelPerc: Percent ALU Delay (%)\n");
      printf (" CacheMissPerc: Percent Cache Miss (%)\n");
      printf (" BranchJmpPer: Branch Percent Not-Taken (%)\n");
      printf (" LocalityALU: Locality of ALU (%)\n");
      printf (" LocalityLD: Locality of Load (%)\n");
      printf (" PredACC: Prediction Accuracy (%)\n");
      printf (" OUTfile: Simulation Output File (*.out)\n");
      printf (" LOGfile: Events log file (*.log) ");
      printf ("(optional, put '0' if not needed)\n\n");
      exit(1);
   }
   if (strcmp(argv[14], "0") != 0) logFileNeeded = 1;

   /*************************************************************************/
   /* Read in input information                                             */
   /*************************************************************************/

   SIMSEED = atoi(argv[1]);
   MAXINST = atoi(argv[2]);
   MAXREGISTER = atoi(argv[3]);
   LOADPERC = atoi(argv[4]);
   ALUPERC = atoi(argv[5]);
   BRANCHPERC = atoi(argv[6]);
   STOREPERC = atoi(argv[7]);
   ALUDELPERC = atoi(argv[8]);
   CACHEMISSPERC = atoi(argv[9]);
   BRCHJMPPERC = atoi(argv[10]);
   LOCALITYALU = atoi(argv[11]);
   LOCALITYLD = atoi(argv[12]);
   PREDACC = atoi(argv[13]);

   /*************************************************************************/
   /* Prepare output files                                                  */
   /*************************************************************************/

   outfptr = fopen (argv[14], "a");
   fprintf (outfptr, "Value Prediction Simulation by Peter Giese\n");
   fprintf (outfptr, "Simulation report generated on ");
   fprintf (outfptr, "%4d-%02d-%02d",now.tm_year+1900,
   now.tm_mon+1,now.tm_mday);
   fprintf (outfptr, "\n");
   fprintf (outfptr, "Arguments:\n");
   fprintf (outfptr, " - Seed: %s\n", argv[1]);
   fprintf (outfptr, " - MAXINST: %s\n", argv[2]);
   fprintf (outfptr, " - MaxRegister: %s\n", argv[3]);
   fprintf (outfptr, " - LoadPerc: %s\n", argv[4]);
   fprintf (outfptr, " - ALUPerc: %s\n", argv[5]);
   fprintf (outfptr, " - BranchPerc: %s\n", argv[6]);
   fprintf (outfptr, " - StorePerc: %s\n", argv[7]);
   fprintf (outfptr, " - ALUDelPerc: %s\n", argv[8]);
   fprintf (outfptr, " - CacheMissPerc: %s\n", argv[9]);
   fprintf (outfptr, " - BrchNotTakenPerc: %s\n", argv[10]);
```

```
fprintf (outfptr, " - LocalityALU: %s\n", argv[11]);
fprintf (outfptr, " - LocalityLD: %s\n", argv[12]);
fprintf (outfptr, " - PREDACC: %s\n", argv[13]);
fprintf (outfptr, " - OUTfile: %s\n", argv[14]);
if (logFileNeeded) fprintf (outfptr, " - LOGfile: %s\n", argv[15]);
fprintf (outfptr, "\n");

/* Create log file (if requested) */
if (logFileNeeded) {

   logfptr = fopen (argv[15], "w");
   fprintf (logfptr, "Value Prediction Simulation by Peter Giese\n");
   fprintf (logfptr, "Simulation log generated on ");
   fprintf (logfptr, "%4d-%02d-%02d",now.tm_year+1900,
   now.tm_mon+1,now.tm_mday);
   fprintf (logfptr, "\n");
   fprintf (logfptr, "Arguments:\n");
   fprintf (logfptr, " - Seed: %s\n", argv[1]);
   fprintf (logfptr, " - Simulation MAXINST: %s\n", argv[2]);
   fprintf (logfptr, " - MaxRegister: %s\n", argv[3]);
   fprintf (logfptr, " - LoadPerc: %s\n", argv[4]);
   fprintf (logfptr, " - ALUPerc: %s\n", argv[5]);
   fprintf (logfptr, " - BranchPerc: %s\n", argv[6]);
   fprintf (logfptr, " - StorePerc: %s\n", argv[7]);
   fprintf (logfptr, " - ALUDelPerc: %s\n", argv[8]);
   fprintf (logfptr, " - CachePerc: %s\n", argv[9]);
   fprintf (logfptr, " - BrchTakenPerc: %s\n", argv[10]);
   fprintf (logfptr, " - LocalityALU: %s\n", argv[11]);
   fprintf (logfptr, " - LocalityLD: %s\n", argv[12]);
   fprintf (logfptr, " - PREDACC: %s\n", argv[13]);
   fprintf (logfptr, " - OUTfile: %s\n", argv[14]);
   fprintf (logfptr, " - LOGfile: %s\n", argv[15]);
   fprintf (logfptr, "\n");
}

// Initialize simulation values and results arrays

MaxPipe = 5;     //Number of pipeline stages fixed (do not change)
MaxWindow = 32;  //Instruction Window Size fixed
ALUDelay = 5;
CacheDelay = 10;
LdDestPred = 1;  // Load Desitnation VP on = 1 and off = 0
LdSrcePred = 0;  // Load Source VP on = 1 and off = 0
ALUDestPred = 1; // ALU Destination VP on = 1 and off = 0
ALUSrcePred = 0; // ALU source VP on = 1 and off = 0
OtherSrcePred = 0; // Other (Brch&Store) source VP on = 1 and off = 0
Enhanced = 1;    // Enhancement VP on = 1 and off = 0

srand(SIMSEED);

Opcode = (Instruction *) malloc (MaxWindow * sizeof (Instruction));
for (Index = 0; Index < MaxWindow; Index++) {
   Opcode[Index].Id = 0;
   Opcode[Index].Type = 0;
   Opcode[Index].OperA = 0;
   Opcode[Index].OperB = 0;
   Opcode[Index].OperC = 0;
   Opcode[Index].OperD = 0;
   Opcode[Index].Latency = 0;
   Opcode[Index].Active = 0;
   Opcode[Index].Pred = 0;
   Opcode[Index].Exit = 0;
        }
RegisterFile = (RegStuff *) malloc (MAXREGISTER * sizeof (RegStuff));
for (Index = 0; Index < MAXREGISTER; Index++){
   RegisterFile[Index].Usage = 0;
   RegisterFile[Index].Flag = 0;
}
DataWrite = (int*)calloc(2, sizeof(int));
for (Index = 0; Index < 2; Index++){
   DataWrite[Index] = -1;
}
Pipe = (int*)calloc(MaxPipe+1, sizeof(int));
for (Index = 0; Index < MaxPipe+1; Index++){
   Pipe[Index] = 0;
}

CycleCnt = 0;
InstCnt = 0;
MaxWinInst = 0;
WindowCnt = 0;
OffSet = 0;
CommitCnt = 0;
WinSize =0;
LoadCnt = 0;
ALUCnt = 0;
BranchCnt = 0;
StoreCnt = 0;
BrchTakenCnt = 0;
ALUMissCnt = 0;
CacheReadCnt = 0;
CacheWriteCnt = 0;
LoadPredCnt = 0;
LoadSrceCnt = 0;
ALUPredCnt = 0;
ALUSrceCnt = 0;
OtherSrceCnt = 0;
PredLdCorrCnt = 0;
PredALUCorrCnt = 0;
ALULat = 0;
MemLat = 0;
DataDep = 0;
DataHaz = 0;
BrchHaz = 0;
BrchDisCnt = 0;
BrchDisSum = 0;

if (LOADPERC+ALUPERC+BRANCHPERC+STOREPERC != 100){
fprintf(outfptr, "Input Error LOADPERC+ALUPERC+BRANCHPERC+STOREPERC != 100");
exit(1);
}

/*************************************************************************/
/* Main Simulation Loop                                                  */
/*************************************************************************/

while (CommitCnt < MAXINST){
   // Generate next instruction and pipeline
   if (WindowCnt < MaxWindow){
        if (Pipe[0] == 0){  // Fetch next instruction
      if (MaxWinInst < InstCnt){
         //Get next instruction from Opcode window
         MaxWinInst = MaxWinInst + 1;
         Pipe[0] = MaxWinInst;
      }
      else {
         // Generate next instruction
         InstCnt = InstCnt + 1;
         MaxWinInst = MaxWinInst + 1;
         Opcode[WindowCnt].Id = InstCnt;
         Opcode[WindowCnt].Active = 1;
         Opcode[WindowCnt].Pred = 0;
         Opcode[WindowCnt].Exit = 0;
      BrchDisCnt = BrchDisCnt + 1;
      RanGen = (int)101*rand()/(RAND_MAX+1.0); //Random between 0 and 100
      if (RanGen >= 0 && RanGen <= LOADPERC){
          Opcode[WindowCnt].Type = 1; //Load Instruction
          Opcode[WindowCnt].OperA = ((int)MAXREGISTER*rand()/(RAND_MAX+1.0));
          //Random between 0 and MaxReg-1;
          DataWrite[1] = Opcode[WindowCnt].OperA;
          Opcode[1].OperB = ((int)MAXREGISTER*rand()/(RAND_MAX+1.0));
          //Random number between 0 and MaxReg-1;
          RanAccess = ((int)100*rand()/(RAND_MAX+1.0))+1;// Random number
          between 1 and 100;
          if (RanAccess > 0 && RanAccess <= CACHEMISSPERC){
             Opcode[WindowCnt].OperD = CacheDelay; // Cache Latency
             CacheReadCnt = CacheReadCnt + 1;
          }
          else{
             Opcode[WindowCnt].OperD = 1; // No Cache Latency - One Cycle
          }
          Opcode[WindowCnt].Latency = Opcode[WindowCnt].OperD; //Save latency
          in event of rollback
          LoadCnt = LoadCnt + 1;
      }
      if (RanGen > LOADPERC && RanGen <= (LOADPERC+ALUPERC)){
          Opcode[WindowCnt].Type = 2;  //ALU Instruction
          Opcode[WindowCnt].OperA = ((int)MAXREGISTER*rand()/(RAND_MAX+1.0));
          // Random number between 0 and MaxReg-1;
          DataWrite[1] = Opcode[WindowCnt].OperA;
          Opcode[WindowCnt].OperB = ((int)MAXREGISTER*rand()/(RAND_MAX+1.0));
          // Random between 0 and MaxReg-1;
          Opcode[WindowCnt].OperC = ((int)MAXREGISTER*rand()/(RAND_MAX+1.0));
          // Random between 0 and MaxReg-1;
          RanAccess = ((int)100*rand()/(RAND_MAX+1.0))+1; //Random number
          between 1 and 100;
          if (RanAccess > 0 && RanAccess <= ALUDELPERC){
              Opcode[WindowCnt].OperD = ALUDelay; // ALU Latency
              ALUMissCnt = ALUMissCnt + 1;
           }
           else{
              Opcode[WindowCnt].OperD = 1; //No ALU Latency - One Cycle
           }
           Opcode[WindowCnt].Latency = Opcode[WindowCnt].OperD;  //Save
           latency in event of rollback
           ALUCnt = ALUCnt + 1;
         }
       if (RanGen > (LOADPERC+ALUPERC) && RanGen <= (LOAD
           PERC+ALUPERC+BRANCHPERC)){
           Opcode[WindowCnt].Type = 3; //Branch Instruction
           DataWrite[1] = -1;
           Opcode[WindowCnt].OperA =((int)MAXREGISTER*rand()/(RAND_MAX+1.0));
           //Random between 0 and
           MaxReg-1;
           Opcode[WindowCnt].OperB =((int)MAXREGISTER*rand()/(RAND_MAX+1.0));
           //Random between 0 and MaxReg-1;
           RanAccess = ((int)100*rand()/(RAND_MAX+1.0))+1; // Random number
           between 1 and 100;
           if (RanAccess <= BRCHJMPPERC){
              Opcode[WindowCnt].OperC = 1; //Branch Taken (jump)
              BrchTakenCnt = BrchTakenCnt + 1;
           }
           else{
              Opcode[WindowCnt].OperC = 0; //Branch Not Taken (pass through)
           }
           BranchCnt = BranchCnt + 1;
           BrchDisSum = BrchDisSum + BrchDisCnt;
           BrchDisCnt = 0;
       }
      if (RanGen > (LOADPERC+ALUPERC+BRANCHPERC) && RanGen <= (LOAD
           PERC+ALUPERC+BRANCHPERC+STOREPERC)){
           Opcode[WindowCnt].Type = 4; //Store Instruction
           DataWrite[1] = -1;
           Opcode[WindowCnt].OperA =
           ((int)MAXREGISTER*rand()/(RAND_MAX+1.0)); // Random  between 0 and
           MaxReg-1;
           Opcode[WindowCnt].OperB =
           ((int)MAXREGISTER*rand()/(RAND_MAX+1.0)); // Random between 0 and
           MaxReg-1;
           RanAccess = ((int)100*rand()/(RAND_MAX+1.0))+1;// Random between 1
           and 100;
           if (RanAccess > 0 && RanAccess <= CACHEMISSPERC){
               Opcode[WindowCnt].OperD = CacheDelay; // Write Cache Latency
               CacheWriteCnt = CacheWriteCnt + 1;
           }
           else{
               Opcode[WindowCnt].OperD = 1; // No Write Cache Latency - One
               Cycle
           }
           Opcode[WindowCnt].Latency = Opcode[WindowCnt].OperD;  //Save
           latency in event of rollback
           StoreCnt = StoreCnt + 1;
       }
       //Data Dependency Check - START
       if (Opcode[WindowCnt].Type == 1){
           if(Opcode[WindowCnt].OperB == DataWrite[0]){
               DataDep = DataDep+1;
           }
       }
       if (Opcode[WindowCnt].Type == 2){
           Flag = 0;
           if(Opcode[WindowCnt].OperB == DataWrite[0]){
               DataDep = DataDep+1;
               Flag = 1;
           }
           if (Flag == 0){
               if(Opcode[WindowCnt].OperC == DataWrite[0]){
                   DataDep = DataDep+1;
               }
           }
       }
           if (Opcode[WindowCnt].Type == 3){
               Flag = 0;
               if(Opcode[WindowCnt].OperA == DataWrite[0]){
                   DataDep = DataDep+1;
                   Flag = 1;
               }
               if (Flag == 0){
                  if(Opcode[WindowCnt].OperB == DataWrite[0]){
                     DataDep = DataDep+1;
                  }
               }
            }
            if (Opcode[WindowCnt].Type == 4){
               Flag = 0;
               if(Opcode[WindowCnt].OperA == DataWrite[0]){
                   DataDep = DataDep+1;
                   Flag = 1;
```

```
                }
                if (Flag == 0){
                    if(Opcode[WindowCnt].OperB == DataWrite[0]){
                        DataDep = DataDep+1;
                    }
            }
        }
        DataWrite[0] = DataWrite[1];
        //Data Dependency Check - END
        WindowCnt = WindowCnt + 1;
        if (WindowCnt > WinSize) {
            WinSize = WindowCnt; //Keep track of largest window size
        }
        Opcode[WindowCnt].Id = 0;
        Pipe[0] = InstCnt;
    }
    }
}
// Shift through Pipeline Five Stages 0 .. 1,2,3,4,5
for (s = MaxPipe; s>0; s--){
    if (Pipe[s] == 0){
    if (Pipe[s-1] != 0){
       ExecFlag = 1;
       IndexBck = Pipe[s-1]-(OffSet+1);
       if (Opcode[IndexBck].Type == 1){ //Load Instruction
          if (s == 2){  //Entering ID and Source Value Prediction
             if (LdSrcePred == 1){ //Source Prediction Active
             if (RegisterFile[Opcode[IndexBck].OperB].Flag == 0 && Register
                File[Opcode[IndexBck].OperB].Usage > 0){
                Opcode[RegisterFile[Opcode[IndexBck].OperB].Usage-
                (OffSet+1)].Pred = 1;
                RegisterFile[Opcode[IndexBck].OperB].Flag = 1; //Load source
                value has been predicted
                LoadSrceCnt = LoadSrceCnt + 1;
             }
             }
          }
          if (s == 3){  //Entering EXEC and Destination Value Prediction
             if (RegisterFile[Opcode[IndexBck].OperB].Usage > 0){
             if (RegisterFile[Opcode[IndexBck].OperB].Flag == 0){
                ExecFlag = 0; //RAW Hazard - stall
                DataHaz = DataHaz + 1;
             }
             }
            if (ExecFlag == 1){
               if (LdDestPred == 1){ //Desintioan Prediction Active
                   RanAccess = ((int)100*rand()/(RAND_MAX+1.0))+1;   //
                   Random number between 1 and 100;
                   if (RanAccess <= LOCALITYLD){
                       Opcode[IndexBck].Pred = 1; //Load destination value
                       has been predicted
                       RegisterFile[Opcode[IndexBck].OperA].Flag = 1;
                       LoadPredCnt = LoadPredCnt + 1;
                   }
               }
            }
          }
          if (s == 5){ //Entering WB Latency hazard
             if (Enhanced == 1){ // Value predition with latency bypass
                 if (Opcode[IndexBck].Pred != 1){ //Bybass if prediced
                     Opcode[IndexBck].OperD = Opcode[IndexBck].OperD-1;
                     if (Opcode[IndexBck].OperD > 0) {
                        ExecFlag = 0; //Cache Miss cycle - stall
                        MemLat = MemLat + 1;
                     }
                     else {
                        RegisterFile[Opcode[IndexBck].OperA].Usage = 0;
                        //Forward & clear register status
                        RegisterFile[Opcode[IndexBck].OperA].Flag = 0;
                     }
                 }
             }
             else{
                Opcode[IndexBck].OperD = Opcode[IndexBck].OperD-1;
                if (Opcode[IndexBck].OperD > 0) {
                   ExecFlag = 0; //Cache Miss cycle - stall
                   MemLat = MemLat + 1;
                }
                else {
                   RegisterFile[Opcode[IndexBck].OperA].Usage = 0;
                  //Forward & clear register status
                   RegisterFile[Opcode[IndexBck].OperA].Flag = 0;
                }
             }
          }
          if (Opcode[IndexBck].Type == 2){ //ALU Instruction
             if (ALUSrcePred == 1){ //Source Prediction Active
                if (s == 2){  //Entering ID and Source Value Prediction
                   if (RegisterFile[Opcode[IndexBck].OperB].Flag == 0 &&
                      RegisterFile[Opcode[IndexBck].OperB].Usage > 0){
                      Opcode[RegisterFile[Opcode[IndexBck].OperB].Usage-
                      (OffSet+1)].Pred = 1;
                      RegisterFile[Opcode[IndexBck].OperB].Flag = 1; //ALU
                      source value has been predicted
                      LUSrceCnt = ALUSrceCnt + 1;
                   }
                   if (RegisterFile[Opcode[IndexBck].OperC].Flag == 0 &&
                      RegisterFile[Opcode[IndexBck].OperC].Usage > 0){
                      Opcode[RegisterFile[Opcode[IndexBck].OperC].Usage-
                      (OffSet+1)].Pred = 1;
                      RegisterFile[Opcode[IndexBck].OperC].Flag = 1; //ALU
                      source value has been predicted
                      ALUSrceCnt = ALUSrceCnt + 1;
                   }
                }
             }
             if (s == 3){ //Entering EXEC and Destination Value Prediction
                 if (RegisterFile[Opcode[IndexBck].OperB].Usage > 0){
                     if (RegisterFile[Opcode[IndexBck].OperB].Flag == 0){
                        ExecFlag = 0; //RAW Hazard - stall
                        DataHaz = DataHaz + 1;
                     }
                 }
             if (RegisterFile[Opcode[IndexBck].OperC].Usage > 0){
                 if (RegisterFile[Opcode[IndexBck].OperC].Flag == 0){
                     ExecFlag = 0; //RAW Hazard - stall
                     DataHaz = DataHaz + 1;
                 }
             }
             if (ExecFlag == 1){
                 if (ALUDestPred == 1){ //Destination Prediction Active
                     RanAccess = ((int)100*rand()/(RAND_MAX+1.0))+1; //
                     Random number between 1 and 100;
                     if (RanAccess <= LOCALITYALU){
                        Opcode[IndexBck].Pred = 1; //ALU value has been
                        Predicted
                        RegisterFile[Opcode[IndexBck].OperA].Flag = 1;
                        ALUPredCnt = ALUPredCnt + 1;
                     }
                 }
             }
          }
          if (s == 4){ //Entering MEM Latency Hazard
               if (Enhanced == 1){ // Value predition with latency bypass
                  if (Opcode[IndexBck].Pred != 1){ //Bybass if prediced
                     Opcode[IndexBck].OperD = Opcode[IndexBck].OperD-1;
                     if (Opcode[IndexBck].OperD > 0) {
                        ExecFlag = 0; //ALU Latency - stall
                        ALULat = ALULat + 1;
                     }
                     else {
                        RegisterFile[Opcode[IndexBck].OperA].Usage = 0;
                        //Forward & clear register status
                        RegisterFile[Opcode[IndexBck].OperA].Flag = 0;
                     }
                  }
               }
               else {
                  Opcode[IndexBck].OperD = Opcode[IndexBck].OperD-1;
                  if (Opcode[IndexBck].OperD > 0) {
                     ExecFlag = 0; //ALU Latency - stall
                     ALULat = ALULat + 1;
                  }
                  else {
                  RegisterFile[Opcode[IndexBck].OperA].Usage = 0; //Forward &
                  clear register status
                  RegisterFile[Opcode[IndexBck].OperA].Flag = 0;
                  }
               }
           }
       }
       if (Opcode[IndexBck].Type == 3){ //Branch Instruction
          if (s == 2){  //Entering ID Data Hazard
             if (OtherSrcePred == 1){ //Source Prediction Active
                if (RegisterFile[Opcode[IndexBck].OperA].Flag == 0 && Regis
                   terFile[Opcode[IndexBck].OperA].Usage > 0){
                   Opcode[RegisterFile[Opcode[IndexBck].OperA].Usage-
                   (OffSet+1)].Pred = 1;
                   RegisterFile[Opcode[IndexBck].OperA].Flag = 1; //Load
                   source value has been predicted
                   OtherSrceCnt = OtherSrceCnt + 1;
                }
                if (RegisterFile[Opcode[IndexBck].OperB].Flag == 0 && Regis
                   terFile[Opcode[IndexBck].OperB].Usage > 0){
                   Opcode[RegisterFile[Opcode[IndexBck].OperB].Usage-
                   (OffSet+1)].Pred = 1;
                   RegisterFile[Opcode[IndexBck].OperB].Flag = 1; //Load
                   source value has been predicted
                   OtherSrceCnt = OtherSrceCnt + 1;
                }
             }
             if (RegisterFile[Opcode[IndexBck].OperA].Usage > 0){
                if (RegisterFile[Opcode[IndexBck].OperA].Flag == 0){
                   ExecFlag = 0; //RAW Hazard - stall
                   DataHaz = DataHaz + 1;
                }
             }
             if (RegisterFile[Opcode[IndexBck].OperB].Usage > 0){
                 if (RegisterFile[Opcode[IndexBck].OperB].Usage == 0){
                    ExecFlag = 0; //RAW Hazard - stall
                    DataHaz = DataHaz + 1;
                 }
             }
          }
          if (s == 3){  //Branch Hazard
             if (Opcode[IndexBck].OperC == 1){
                if (Pipe[1] > 0) {//Flush last instruction
                   Pipe[0] = Pipe[1];
                   Pipe[1] = 0;
                   InstCnt = InstCnt - 1;
                   MaxWinInst = MaxWinInst - 1;
                   WindowCnt = WindowCnt -1;
                   BrchHaz = BrchHaz + 1; //Record control hazard
                }
             }
          }
       }
       if (Opcode[IndexBck].Type == 4){ //Store Instruction
          if (s == 2){  //Entering EXEC Data Hazard
              if (OtherSrcePred == 1){ //Source Prediction Active
                  if (RegisterFile[Opcode[IndexBck].OperA].Flag == 0 &&
                      RegisterFile[Opcode[IndexBck].OperA].Usage > 0){
                      Opcode[RegisterFile[Opcode[IndexBck].OperA].Usage-
                      (OffSet+1)].Pred = 1;
                      RegisterFile[Opcode[IndexBck].OperA].Flag = 1; //Load
                      source value has been predicted
                      OtherSrceCnt = OtherSrceCnt + 1;
                  }
                  if (RegisterFile[Opcode[IndexBck].OperB].Flag == 0 &&
                      RegisterFile[Opcode[IndexBck].OperB].Usage > 0){
                      Opcode[RegisterFile[Opcode[IndexBck].OperB].Usage-
                      (OffSet+1)].Pred = 1;
                      RegisterFile[Opcode[IndexBck].OperB].Flag = 1; //Load
                      source value has been predicted
                      OtherSrceCnt = OtherSrceCnt + 1;
                  }
              }
          }
          if (s == 3){  //Entering EXEC Data Hazard
              if (RegisterFile[Opcode[IndexBck].OperA].Usage > 0){
                  if (RegisterFile[Opcode[IndexBck].OperA].Flag == 0){
                      ExecFlag = 0; // RAW Hazard - stall
                      DataHaz = DataHaz + 1;
                  }
              }
              if (RegisterFile[Opcode[IndexBck].OperB].Usage > 0){
                  if (RegisterFile[Opcode[IndexBck].OperB].Usage == 0){
                      ExecFlag = 0; // RAW Hazard - stall
                      DataHaz = DataHaz + 1;
                  }
              }
          }
       }
       if (ExecFlag == 1){ //Check to see if instruction can move to next
          pipeline stage
          Pipe[s] = Pipe[s-1];
          Pipe[s-1] = 0;
          Index = Pipe[s]-(OffSet+1);
          if (s == 3){ //entering EXEC Set Register Status to Active
              if (Opcode[Index].Type == 1){ //Load Instruction
                  RegisterFile[Opcode[Index].OperA].Usage = Pipe[s];
              }
              if (Opcode[Index].Type == 2){ //ALU Instruction
                  RegisterFile[Opcode[Index].OperA].Usage = Pipe[s];
              }
          }
       }
```

```
      }
    }
}

 //Prediction Timing and Verification - Start
for (j = 0; j<WindowCnt; j++){
     if (Opcode[j].Pred == 1){
        if (Enhanced == 1){ // Value predition with latency bypass
            Opcode[j].OperD = Opcode[j].OperD-1;
        }
        if (Opcode[j].OperD < 1) {
            Opcode[j].OperD = 1; //Completed latency countdown
            Opcode[j].Pred = 0; //Clear Prediction flag
            RegisterFile[Opcode[j].OperA].Usage = 0; //Clear Register File
            RegisterFile[Opcode[j].OperA].Flag = 0;
            RanAccess = ((int)100*rand()/(RAND_MAX+1.0))+1;// Random between
            1 and 100;
            if (RanAccess <= PREDACC){
                if (Enhanced == 1){ // Value prediction with latency bypass
                    if (Opcode[j].Exit == 1){//Instruction has already exited
                        the pipeline
                        Opcode[j].Active = 0;
                        CommitCnt = CommitCnt + 1;
                     }
                }
                if (Opcode[j].Type == 1){
                    PredLdCorrCnt = PredLdCorrCnt + 1;
                }
                else {
                    PredALUCorrCnt = PredALUCorrCnt + 1;
                }
            }
            else {
            if (Opcode[j].Exit == 1){
                Pipe[MaxPipe] = 0; //Predication is not correct flush and
                reload pipeline
                Pipe[MaxPipe-1] = 0;
                Pipe[MaxPipe-2] = 0;
                Pipe[MaxPipe-3] = 0;
                Pipe[MaxPipe-4] = Opcode[j+1].Id;
                Opcode[j].OperD = Opcode[j].Latency; //Reset latency
                Opcode[j].Active = 0; //Finished with instruction
                Opcode[j].Pred = 0;
                CommitCnt = CommitCnt + 1;
             }
             else{  //Prediction is with the Pipeline stages
                if (Pipe[MaxPipe] == Opcode[j].Id){
                    Pipe[MaxPipe-1] = 0;
                    Pipe[MaxPipe-2] = 0;
                    Pipe[MaxPipe-3] = 0;
                }
                if (Pipe[MaxPipe-1] == Opcode[j].Id){
                    Pipe[MaxPipe-2] = 0;
                    Pipe[MaxPipe-3] = 0;
                }
                  if (Pipe[MaxPipe-2] == Opcode[j].Id){
                    Pipe[MaxPipe-3] = 0;
                  }
                  Pipe[MaxPipe-4] = Opcode[j+1].Id;
                  Opcode[j].OperD = Opcode[j].Latency; //Reset latency
                  Opcode[j].Pred = 0;
               }
               if (j+2 >= WindowCnt){
                  MaxWinInst = InstCnt; //next instruction number (empty
                  pipeline)
                  Pipe[MaxPipe-5] = 0;
               }
               else {
                  MaxWinInst = Opcode[j+2].Id; //next instruction exists
                  within pipeline
                  Pipe[MaxPipe-5] = Opcode[j+2].Id;
               }
               for (i=j+1; i < WindowCnt; i++){//Window Reset - Flush and
                  prepare
                  Opcode[i].OperD = Opcode[i].Latency; //Reset latency
                  Opcode[i].Active = 1;
                  Opcode[i].Pred = 0;
                  Opcode[i].Exit = 0;
               }
               for (i=0; i < MAXREGISTER; i++){
                  RegisterFile[i].Usage = 0; //Clear Register File
                  RegisterFile[i].Flag = 0; //Clear Register File
               }
               j = MaxWindow; //Exit loop
            }
         }
      }
   }
   // Check for completed instruction and Window FIFO - Start
   if(Pipe[MaxPipe] != 0){
      i = 0;
      while (i < WindowCnt){
         if (Opcode[i].Id == Pipe[MaxPipe]){ // Commit Instruction
            if (Opcode[i].Type == 1){ //Load Instruction
                if (Opcode[i].Pred != 1) {
                    Opcode[i].Active = 0;
                    CommitCnt = CommitCnt+1;
                }
            }
            if (Opcode[i].Type == 2){ //ALU Instruction
                if (Opcode[i].Pred != 1) {
                    Opcode[i].Active = 0;
                    CommitCnt = CommitCnt+1;
                }
            }
            if (Opcode[i].Type == 3){ //Branch Instruction
                Opcode[i].Active = 0;
                CommitCnt = CommitCnt+1;
            }
            if (Opcode[i].Type == 4){ //Store Instruction
                Opcode[i].Active = 0;
                CommitCnt = CommitCnt+1;
            }
            Opcode[i].Exit = 1; //Instruction has exited the pipeline
            Pipe[MaxPipe] = 0;
            i = MaxWindow;  //Exit loop
         }
         else{
            i = i+1;
         }
      }
   }
   i = 0;
   LoopTest = WindowCnt;
   while (i < LoopTest){
      if (Opcode[0].Active == 0 ){ //Check to see if window shift is required
          WindowCnt = WindowCnt - 1;
          OffSet = OffSet + 1;
          for (j=0; j < WindowCnt; j++){
              Opcode[j].Id = Opcode[j+1].Id;
              Opcode[j].Type = Opcode[j+1].Type;
              Opcode[j].OperA = Opcode[j+1].OperA;
              Opcode[j].OperB = Opcode[j+1].OperB;
             Opcode[j].OperC = Opcode[j+1].OperC;
             Opcode[j].OperD = Opcode[j+1].OperD;
             Opcode[j].Latency = Opcode[j+1].Latency;
             Opcode[j].Active = Opcode[j+1].Active;
             Opcode[j].Pred = Opcode[j+1].Pred;
             Opcode[j].Exit = Opcode[j+1].Exit;
          }
          i = i + 1;
       }
       else {
          i = LoopTest; //exit Loop
       }
    }
    if (Opcode[0].Active == 0){ //Check for empty pipeline case
        WindowCnt = 0; //Reset window
        OffSet = InstCnt;
    }
    // Shift Simulation Time by one Step unit
    CycleCnt = CycleCnt + 1;
}
// Determine statistics and print results
fprintf(outfptr, "\nSimulation Parameters \n\n");
fprintf(outfptr, "Load Percentage:              %d\n", LOADPERC);
fprintf(outfptr, "ALU Percentage:               %d\n", ALUPERC);
fprintf(outfptr, "Branch Percentage:            %d\n", BRANCHPERC);
fprintf(outfptr, "Store Percentage:             %d\n", STOREPERC);
fprintf(outfptr, "ALU Delay Percentage:         %d\n", ALUDELPERC);
fprintf(outfptr, "Cache Miss(R/W) Percentage:   %d\n", CACHEMISSPERC);
fprintf(outfptr, "Branch Not-Taken Percentage:  %d\n", BRCHJMPPERC);
fprintf(outfptr, "ALU Locality:                 %d\n", LOCALITYALU);
fprintf(outfptr, "Load Locality:                %d\n", LOCALITYLD);
fprintf(outfptr, "Prediction Accuracy:          %d\n", PREDACC);

fprintf(outfptr, "\nCompleted Instruction:      %d", MAXINST);
fprintf(outfptr, "\nExecution Window:           %d", MaxWindow);
fprintf(outfptr, "\nNumber of Registers:        %d", MAXREGISTER);
fprintf(outfptr, "\nALU Latency:                %d", ALUDelay);
fprintf(outfptr, "\nCache Latency:              %d", CacheDelay);
fprintf(outfptr, "\nLoad Destination Predict:   %d", LdDestPred);
fprintf(outfptr, "\nLoad Source Predict:        %d", LdSrcePred);
fprintf(outfptr, "\nALU Destination Predict:    %d", ALUDestPred);
fprintf(outfptr, "\nALU Source Predict:         %d", ALUSrcePred);
fprintf(outfptr, "\nOther Source Predict:       %d", OtherSrcePred);
fprintf(outfptr, "\nEnhanced VP:                %d", Enhanced);

fprintf(outfptr, "\n\nSimulation Results \n");
fprintf(outfptr, "\nTotal Instructions Issued: %d", InstCnt+BrchHaz);
fprintf(outfptr, "\nTotal Instructions Completed: %d", CommitCnt);
fprintf(outfptr, "\nTotal Number of Cycles: %d", CycleCnt);
fprintf(outfptr, "\nCPI: %.5f", (float)CycleCnt/CommitCnt);
LoopTest = LoadCnt + ALUCnt + BranchCnt + StoreCnt;
fprintf(outfptr, "\nInstuction: SUM(%d) Load(%d) ALU(%d) Branch(%d)
Store(%d)", LoopTest, LoadCnt, ALUCnt, BranchCnt, StoreCnt);
fprintf(outfptr, "\nBranch Not-Taken(%d)", BrchTakenCnt);
fprintf(outfptr, "\nBranch Distance(%d)", BrchDisSum/BranchCnt);
fprintf(outfptr, "\nLoad Latency(%d)", CacheReadCnt);
fprintf(outfptr, "\nLoad Designation Preditction(%d)", LoadPredCnt);
fprintf(outfptr, "\nLoad Source Preditction(%d)", LoadSrceCnt);
fprintf(outfptr, "\nALU Latency(%d)", ALUMissCnt);
fprintf(outfptr, "\nALU Destination Prediction(%d)", ALUPredCnt);
fprintf(outfptr, "\nALU Source Prediction(%d)", ALUSrceCnt);
fprintf(outfptr, "\nOther Source Prediction(%d)", OtherSrceCnt);
fprintf(outfptr, "\nStore Latency(%d)", CacheWriteCnt);
fprintf(outfptr, "\nCorrect ALU Prediction(%d)", PredALUCorrCnt);
fprintf(outfptr, "\nCorrect Load Prediction(%d)", PredLdCorrCnt);
fprintf(outfptr, "\n\nInstruction-based Data Dependency(%d)", DataDep);
fprintf(outfptr, "\nHazards: Data(%d) Branch (%d)", DataHaz, BrchHaz);
fprintf(outfptr, "\nLatencys: ALU(%d) Load(%d)", ALULat, MemLat);
fprintf(outfptr, "\nLargest Window Size: %d", WinSize);

fclose (outfptr);
if (logFileNeeded) fclose (logfptr);
}
```

valpredv1.h

```
// Type 1 - Load
// Type 2 - ALU
// Type 3 - Branch
// Type 4 - Store

typedef struct {

   int Id;
   int Type;
   int OperA;
   int OperB;
   int OperC;
   int OperD;
   int Latency;
   int Active;
   int Pred;
   int Exit;

} Instruction;

typedef struct {

   int Usage;
   int Flag;

} RegStuff;
```