Buletinul Stiintific al Universitatii “Politehnica” din Timisoara, ROMANIA
Seria AUTOMATICA si CALCULATOARE
PERIODICA POLITECHNICA, Transactions on AUTOMATIC CONTROL and COMPUTER SCIENCE
Vol.47 (61), 2002, ISSN 1224-600X

# Understanding Value Prediction through Complex Simulations

**Adrian FLOREA, Lucian N. VINTAN, Dorin SIMA**

*University “L. Blaga”, Department of Computer Science, Str. E. Cioran, No. 4, Sibiu-2400, ROMANIA,*
*Tel./Fax: ++40-269-212716, E-mail:* aflorea@vectra.ulbsibiu.ro, *vintan@jupiter.ulbsibiu.ro*

***Abstract* - Through this paper we investigated the value locality degree present in real-worlds program, and extended the value locality concept to all general-purpose registers (MIPS architecture). Also we exploited this concept through a prediction mechanism of load instruction values from two distinct perspectives: producer-centric and memory-centric. To extract the maximum degree of load value locality we performed execution driven simulations on SPEC95 benchmarks with unbounded table sizes. The encouraging obtained results facilitate implementation of much simpler prediction structures (at most 32/64 locations), reducing the hardware cost and complexity.**

**Keywords**: *advanced computer architectures, value locality, value prediction, pipeline, simulation, benchmarking.*

## I. INTRODUCTION

Several studies have pointed out that the values produced by the programs execution are often quite repetitive. There are basically two approaches that have been proposed for exploiting this value locality. *First non-speculatively* consists in reusing the results of a prior execution of an instruction [1], and *second* - *speculatively* - predicts the value that will be produced by the current execution of an instruction based on the previous values it has produced [4]. Existing value reuse and prediction schemes operate at the level of a single instruction or short sequence of instructions so that the caches used to temporarily store previous values are typically indexed using the instruction addresses.

Value locality describes the likelihood of recurrence of previously seen program values within computer storage location [4]. Work has focused on exploiting this property to accelerate the processing of instructions within a superscalar processor, with the goal of exposing greater instruction-level parallelism and improving instruction throughput. Value locality makes it possible to exceed the classical data-flow limit, which is defined as the program performance obtained when machine instructions execute as soon as their operands are available. The value locality of particular static loads in a program can be significantly affected by compiler optimizations: loop unrolling, tail replication, etc., since these types of transformations tend to create multiple instances of a load that may now exclusively target memory locations with high or low value locality [4]. The value locality concept is based on a dynamically affinity between a resource name (register, memory location, I/O ports) and the values stored within. Value prediction has been proposed for the purpose of reducing, average memory latency by predicting *load*, *alu* or *store* instruction outcomes, improving throughput of all register/memory - writing instructions.

An important parameter used in the designing process is the history depth. A history depth of **k** means that the retrieval process is made in the last k distinguishes values encountered. The question that appears is "*How much history should be used in the prediction process?"* The actual tradeoffs are varying between a small history with lower value prediction accuracy and a lower hardware cost, and a rich prediction history with higher prediction accuracy but expensive from hardware cost viewpoint.

We investigated the value locality of loads from both memory-centric (data-address based) and producer-centric (instruction-address based) viewpoints. *Program structure load value locality* (PSLVL) measures the locality of values written by particular static load and *message-passing load value locality* (MPLVL) [3] exhibit the locality of values written to a particular address in data memory. Most of prior work on value locality has focused on program structure-based prediction, since there is very little to be gained (in timing) by predicting load values once their addresses are known [3].

The first implemented technique that effectively exploits and captures the value locality existing in the real world programs is *Load Value Prediction* [4]. This relatively new technique has several characteristics that make it attractive to a CPU designer: the availability of the lookup index in the hardware structures very early (at the beginning of the instruction fetch stage), little or no complexity added to critical delay paths in microarchitecture and reducing the memory bandwidth requirements. Starting by Lipasti's Load Value Predictor Unit [4] we evaluated load value prediction accuracy from two perspectives points of view: memory centric and producer centric, varying structures size and architecture. Also we investigated the correct identification degree of load instruction by LVP unit, to take the full advantage of each case: avoiding the cost of a misprediction by identifying the unpredictable loads and reducing the memory access cost if we can identify and verify loads that are highly predictable.

Recently was proposed some new techniques for improving the value prediction accuracy. These include computational predictors and context-based predictors' [6]. Computational predictors make a prediction by computing some function (algorithms) of previous values (e.g. *last value predictor* [4], *stride predictor*). Context based predictors learn the values that follow a particular context - a finite ordered sequences of values - and predict one of the values when the same context repeats (e.g. *prediction by partial matching* - a set of markovian predictors).

## II. SIMULATION METHODOLOGY

Execution driven simulation was conducted using the *SimpleScalar* tool set [8] (comprises associated compilers, run-time environments, and simulators) for the integer SPEC95 benchmarks (standardized benchmarks that reflect the advances in microprocessor technologies, compilers and application). Integer benchmarks were selected because they tend to have less data parallelism and may therefore benefit more from data prediction [6]. SPEC95 was developed by SPEC's Open Systems Group (OSG), which includes more than 30 computer vendors, systems integrators, publishers and consultants from throughout the world [9]. SPEC95 is the third major version of the SPEC CPU benchmark suites, which in 1989 became the first widely accepted standard for comparing compute-intensive performance across various architectures. The new application-based benchmarks can be used across several versions of UNIX and Microsoft Windows NT.

*TABLE 1. Benchmarks Description*

| **SPEC '95 benchmarks** | | | |
|---|---|---|---|
| ***Benchmarks*** | ***Input*** | ***Characteristics*** | ***Inst. Count executed*** |
| Applu | Applu.in | Solves matrix system with pivoting. | 5000000 |
| Apsi | Apsi.in | Calculates statistics on temperature and pollutants in a grid. | 5000000 |
| Cc1 | 1stmt.i | Compiles pre-processed source into optimized SPARC assembly code. | 5000000 |
| Compress95 | Bigtest.in | Compresses large text files (about 16MB) using adaptive Limpel-Ziv coding. | 5000000 |
| Go | 9stone21.in | Uses several techniques common in the field of artificial intelligence to play the ancient Asian game of Go. | 5000000 |
| Hydro2d | Hydro2d.in | Hydrodynamical Navier Stokes equations are used to compute galactic jets. | 5000000 |
| Ijpeg | Vigo.ppm | Performs jpeg image compression with various parameters. | 5000000 |
| Perl | Scrabbl.pl | Performs text and numeric manipulations (anagrams/prime number factoring). | 5000000 |
| Su2cor | Su2cor.in | Masses of elementary particles are computed in the Quark-Gluon theory. | 5000000 |
| Swim | Swim.in | Solves shallow water equations using finite difference approximations. (The only single precision benchmark in CFP95.) | 5000000 |
| Tomcatv | Tomcatv.in | Generation of a two-dimensional boundary-fitted coordinate system around general geometric domains. | 5000000 |
| Li | *.lsp | Lisp interpreter. | 5000000 |

## III. OBTAINED RESULTS

We performed several experiments to evaluate the value locality exhibited by MIPS CPU integer registers (unstudied until now as far as we know) and different instruction types. Also, we examined the value locality of load instructions from both memory-centric and producer-centric viewpoints. The value locality for each benchmark is measured by counting the number of time each load/alu instruction retrieves a value from memory/destination register that matches a previously seen value for that dynamic load/alu and dividing by the total number of dynamic loads/alus in the benchmark.

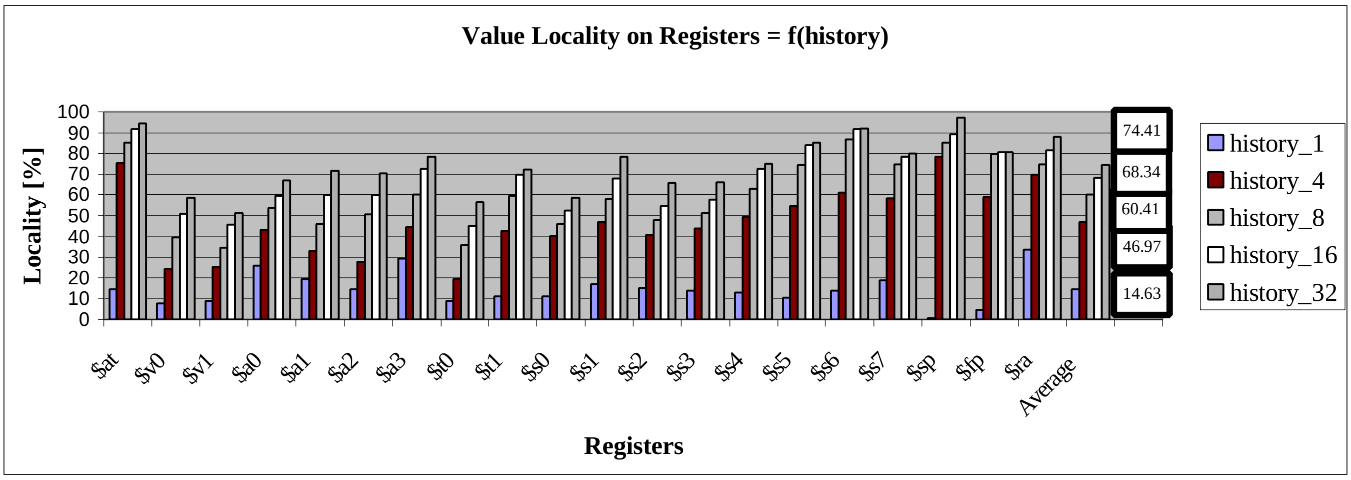

**Figure 1.** Value locality on MIPS's general purpose registers

The *SimpleScalar* architecture is derived from MIPS-IV ISA (Instruction Set Architecture). The MIPS CPU contains 32 general-purpose registers that are numbered 0-31 [7]. Figure 1 exhibits the value locality degrees for some favourable registers belonging to MIPS CPU.

- Register $0 always contains the hardwired value 0.
- Registers $at (1) is reserved for the assembler while $k0 (26) and $k1 (27) are dedicated to operating system and should not be used by user programs or compilers. Register $at is often used in address calculation (**lui** instruction - load upper immediately) and rarely generate more than one unique value (see the memory map at MIPS processor [7]).
- Registers $a0-$a3 (4-7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack). Registers $v0 and $v1 (2, 3) are used to return values from functions. Also, $v0 keeps the system calls code for delivering a small set of operating system-like services.
- Register $gp (28) is a global pointer that points to a middle of a 64k block of memory in the static data segment.
- Register $sp (29) is the stack pointer, which points to the first free location on the stack - varies in a small domain of values. Register $fp (30) is the frame pointer. The *jal* (*jump and link*) instruction writes register $ra (31), the return address from a procedure call. The greater locality exhibited by latest register is due to fewer procedure calls within a program.

The MIPS register-use convention provides *callee*- and *caller*-saved registers because both types of registers are advantageous in different circumstances.

- Callee saved registers $s0-$s7 (16-23) are better used to hold long-lived values that should be preserved across calls, such as variables from a user's program. These registers are only saved during a procedure call if the callee expects to use the register.
- On the other hand, caller-saved registers: $t0-$t9 (8-15, 24, 25) are better used to hold temporary quantities that need not be preserved across a call, such as immediate values in an address calculation. During a call, the callee can also use these registers for short-lived temporaries.

Figure 1 shows clearly that on several registers ($at, $sp, $fp, $ra) the value locality degree is very high (over 90% ! in average). For averaging it was used arithmetic mean. The results obtained are obvious because $at usually contains the same base address (0x1000) for instructions with indexed addressing mode, $sp and $fp are varying in a small domain of values - depending on number of functions formal parameters, and the higher value locality of $ra is due to fewer procedure calls within testing programs. These encouraging results involve the original idea of some value predictors attached to these "special" registers, with a reasonable hardware cost and complexity and, for sure, with important performance benefits. We'll try to develop in a further work this new idea.

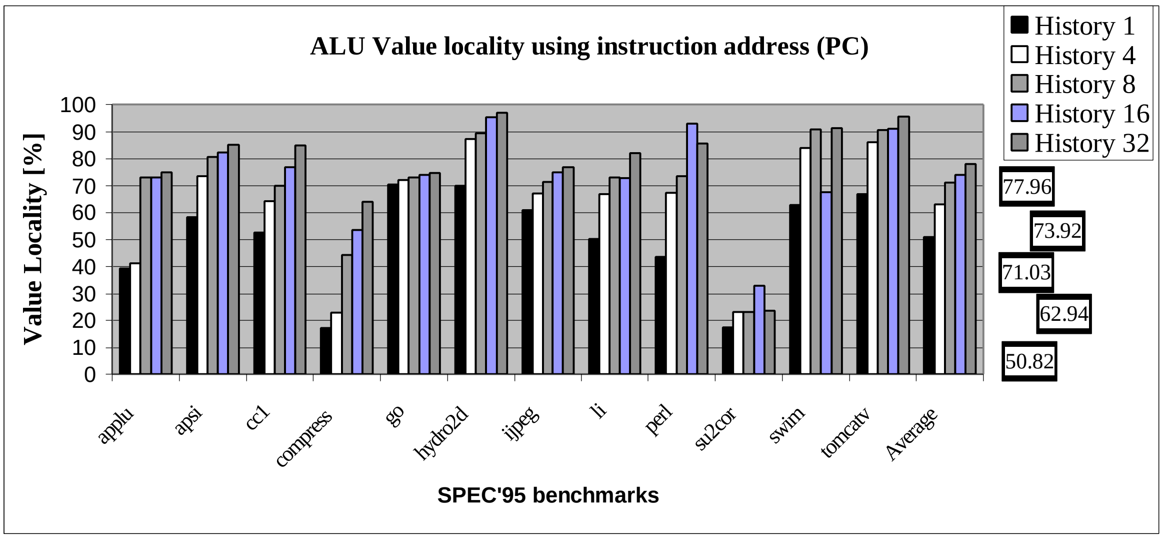

**Figure 2.** ALU Value locality

With a history depth of 1 (thus we check for matches against only the most recently retrieved value) the test programs exhibit value locality in the 51% range (in average) while extending the history depth to 32 can improve that to at most 78% (at average, see Figure 2). A history depth of eight could be an optimum. The simulation results refer to all arithmetical or logical instructions having integer operands (not floating point) excepting MULT and DIV instructions.

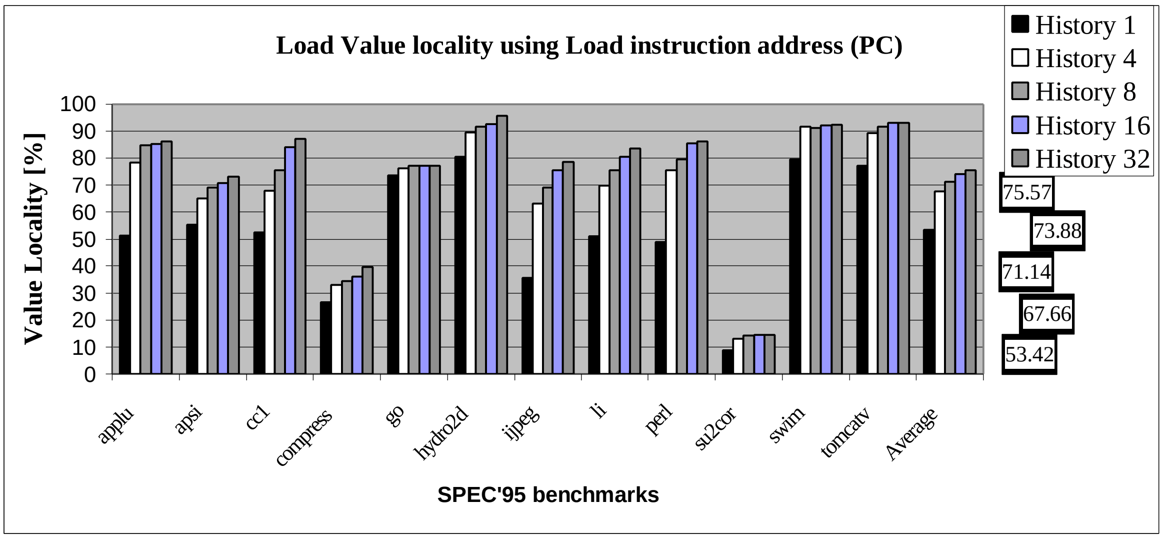

**Figure 3.** Load Value locality from producer-centric point of view

With a history depth of 1 the test programs exhibit value locality in the 53% range (in average) while extending the history depth to 32 can improve that to at most 75% (at average, see Figure 3). This means that the vast majority of static loads express very little dynamic value variations.

Although the value locality degree from memory-centric viewpoint is better than that obtained using load instruction address (69% vs. 53% with history depth of 1 and 86% vs. 75% with history depth of 32 - see Figures 3 and 4), the producer-centric manner is rather preferable. This is due to the very little benefit gained by predicting load values once their addresses are known (Unfortunately these addresses are computed only during ID or ALU pipeline stages, involving thus additional delay compared to the producer-centric approach).

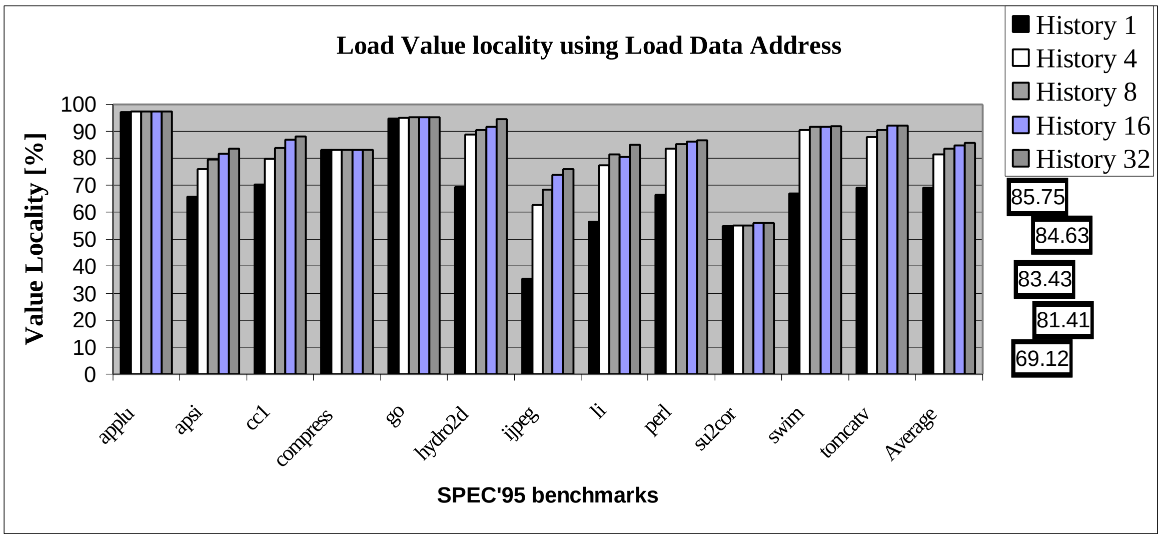

**Figure 4.** Load Value locality from memory-centric point of view

Figure 5 presents some value prediction accuracies, simulating the well-known predictor developed in [4, 5]. For a prediction table of 1024 entries the average accuracy is about 60%, that's quite optimistic taking into account that the predictor is very simple. Our experiments point out that using more sophisticated predictors will involve better performances.

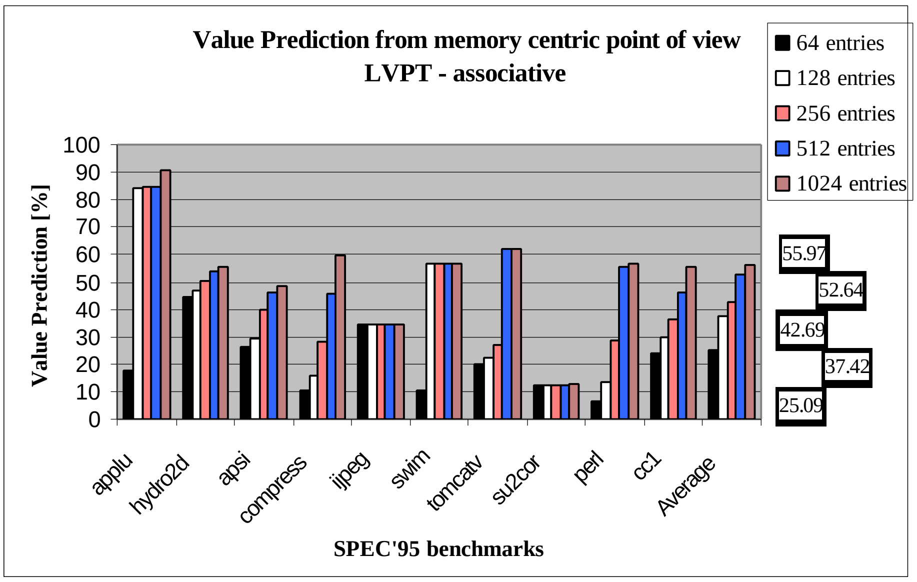

**Figure 5.** Load Value Prediction using data memory address function of Load Value Predictor Table size

To maximise gains when using prediction for speculative execution, it is imperative to have high prediction accuracy and infrequent misspeculation. So, we want to determine **how much confidence** could be offered to prediction automata Load Classification Table by Load Value Predictor. For this purpose, we use a 2-bit saturating counter, having four states (*don't predict*, *don't predict*, *predict*, *predict*) with high confidence to classify the load instructions based on their dynamic behaviour [4]. Table 2 exhibit "*how many times the load value predictor guess the correct value dividing to the number of times when the automata decided that the value is predictable*" and "*how many times the LVP misspredict the value dividing to the number of times when the LCT decided that the value is unpredictable*".

The gap between a real and a perfect classification (100%) of predictable load instructions could be caused by an inefficient replacement mechanism in load value predictor table. Another replacement policy based on histeresys does not change its prediction to a new value until this value doesn't occur a specific number of times in succession. As it can be observed from table 2, the two unpredictable states are not quite semantically covered because the miss-classification rate varied between 13.7% and 24.7%. This means that it could be an important challenge finding other more flexible automata in order to solve this gap.

**Table 2. LCT Hit Rates**. Percentages show fractions of unpredictable and predictable loads identified as such by the LCT. LVPT - associative and *producer centric load value locality*

| SPEC'95 Benchmarks | 64 | | 128 | | 256 | | 512 | | 1024 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **Pred** | **Unpr** | **Pred** | **Unpr** | **Pred** | **Unpr** | **Pred** | **Unpr** | **Pred** | **Unpr** |
| **Applu** | 87.56 | 83.52 | 87.56 | 83.53 | 87.57 | 83.31 | 87.68 | 83.55 | 87.68 | 83.55 |
| **Apsi** | 96.28 | 85.05 | 94.97 | 78.94 | 94.32 | 77.17 | 93.66 | 84.36 | 93.71 | 89.33 |
| **cc1** | 94.13 | 73.83 | 93.93 | 74.78 | 93.52 | 77.71 | 93.64 | 84.18 | 94.35 | 89.34 |
| **Compress** | 89.07 | 93.17 | 89.07 | 93.17 | 89.07 | 93.16 | 89.07 | 93.15 | 89.07 | 93.15 |
| **Fpppp** | 99.41 | 78.23 | 99.17 | 76.29 | 98.3 | 67.12 | 96.01 | 87.06 | 94.96 | 71.1 |
| **Hydro** | 98.86 | 69.06 | 96.41 | 70.68 | 95.88 | 67.26 | 95.32 | 69.01 | 94.91 | 74.74 |
| **Ijpeg** | 86.87 | 98.65 | 87.15 | 99.5 | 87.15 | 99.5 | 87.15 | 99.5 | 87.15 | 99.5 |
| **Perl** | 83.16 | 79.62 | 87.50 | 77.03 | 92.11 | 84.23 | 95.22 | 89.34 | 95.17 | 89.44 |
| **su2cor** | 97.49 | 99.6 | 97.34 | 99.59 | 97.3 | 99.57 | 97.2 | 99.56 | 97.19 | 99.57 |
| **Swim** | 95.78 | 42.02 | 98.55 | 76.62 | 98.55 | 76.41 | 97.28 | 75.56 | 97.28 | 75.56 |
| **Tomcatv** | 95.48 | 63.88 | 95.51 | 61.88 | 95.49 | 40.31 | 97.78 | 88.35 | 97.8 | 91.24 |
| **wave5** | 99.3 | 91.45 | 99.3 | 91.26 | 99.3 | 91.03 | 99.29 | 90.91 | 99.29 | 90.98 |
| **HM** | **93.3** | **75.7** | **93.6** | **80.4** | **93.8** | **75.3** | **93.9** | **86.1** | **93.8** | **86.3** |

## IV. CONCLUSIONS AND FURTHER WORK

The main conclusion is that there are significant (optimistic) amounts of value locality, both from memory-centric (message passing) and producer-centric (program structure) points of view. The simulation results show that all of the instructions within a single application have a relatively small overall result space. That is, while the potential range of values that could be produced is enormous, the actual range of values tends to be much more constrained. Increasing the history depth determines the improvement of value locality. The results are very useful because they express if and in which condition the value prediction is feasible (e.g. the history depth could be a useful indicator in developing the attached predictor). An original evaluation is presented in figure 1, where it is evidenced the value locality concept associated with MIPS's general-purpose registers. The results obtained on some special registers ($at, $sp, $ra) of MIPS architecture are quite remarkable (≈90% value locality degree) leading to conclusion that value prediction might be successful applied at least on these registers. Also we demonstrated that an important challenge is to find some efficient confidence mechanisms.

Encouraged by these first results, we'll try to examine the value locality of all register writing instructions. We intend to exploit store value locality evaluating the potential reduction in multiprocessor data and address traffic. For improving the value prediction accuracy we'll try to develop some new hybrid prediction schemes (composed by a two-level adaptive value predictor and a stride predictor).

## REFERENCES

[1] Sodani A., Sohi G. - "*Dynamic Instruction Reuse*", Proceedings of the 24th International Symposium on Computer Architecture, pp. 194-205, June 1997.

[2] Wang K., Franklin M. - "*Highly Accurate Data Value Prediction using Hybrid Predictors*", Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, December 1997.

[3] Lepak K., Lipasti M. - "*On the Value Locality of Store Instructions*", Proceedings of the 27th Annual International Symposium on Computer Architecture, Vancouver, June 2000.

[4] Lipasti M., Wilkerson C.B., Shen J.P. - "*Value Locality and Load Value Prediction*", Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), pp. 138-147, October 1996.

[5] Lipasti M., Shen J.P. - "*Exceeding the Dataflow Limit via Value Prediction*", Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, December 1996.

[6] Sazeides Y., Smith J.E. - "*The Predictability of Data Values*", Proceedings of Micro-30, December 1-3, 1997 in Research Triangle Park, North Carolina.

[7] Larus J. - "*SPIM S20: A MIPS R2000 Simulator*", Morgan Kaufmann Publishers, 1993.

[8] Burger D., Austin T.M., Bennet S. - "*Evaluating future microprocessors: The SimpleScalar tool set*" - Tech. Rep. CS-TR-96-1308, University of Wisconsin - Madison, July 1996.

[9] http://www.spec.org

[10] Deswet V., Goeman B., Bosschere K. – *Independent hashing as confidence mechanism for value predictors in microprocessors*, Int'l Conf. EuroPar, Augsburg, Germany, 2002

[11] Sazeides Y.T. – *An Analysis of Value Predictibality and its Application to a Superscalar Processor*, PhD Thesis, University of Wisconsin-Madison, SUA, 1999

[12] Vintan L. - *Arhitecturi de procesoare cu paralelism la nivelul instructiunilor*, Editura Academiei Române, Bucuresti, (264 pg.), ISBN 973-27-0734-8, 2000

[13] Vintan L. - *Predictia valorilor instructiunilor*, NET-Report nr. 115, pg.11-17, ISSN 1582-4497, aprilie 2002.