

University of California, Riverside
Department of Computer Science & Engineering

CS 203A, Advanced Computer Architecture Project

Play With Instructions

Yannis Drougas, Kyriakos Karenos, Thomas Repantis

12.08.2003

1 Part I: Base

1.1 An Empirical Analysis of Instruction Repetition [8] (Yannis Drougas)

Instruction repetition is a common characteristic of programs. Many methods have been proposed in order to exploit this phenomenon. But, before exploitation, the properties and reasons of instruction repetition should be studied. This is done in [8].

A program consists of some *static instructions*. Upon execution, a processor executes one or more *instances* of some of these static instructions. These instances are called *dynamic instructions*. An instruction is repeated when one of its instances (meaning, the instruction with both its inputs and output being the same) appears more than once during the execution of the program.

Statistical analysis on data collected from simulations shows that not all of the static instructions have repeated instances. In fact, processing of the results revealed these facts: (i) Only a few of the static instructions (about 25% on average) of a program are actually executed. (ii) Most of the instructions executed (about 80% on average) are also repeated. (iii) Only a small fraction of these instructions that are repeated (about 20% on average) contributes about 95% of all the repeated instructions.

The above make us conclude that the vast majority of the dynamic instructions executed is generated by a small percentage of static instructions.

Next, the dependences of the instructions generating repetitions are examined. In order to trace the various dependences, each instruction is tagged with a value, based on its data dependences. In order to take into account different instruction categories through different contexts, a global, a function level and a local analysis of the instructions are performed.

In global analysis, instructions are divided into three categories according to whether they depend on program input data, on global variables, or on data “hardwired” into the program (like an immediate). The results of the simulations indi-

cate that most of the static instructions as well as most of the repeated instructions depend on such “hardwired” data.

With function level analysis, the repeatability of function parameters is computed. Simulations show that all of the parameters of the majority (more than 50%) of function calls, are repeated. This usually (but not always) means that their outputs are also repeated.

Finally, function-wide statistics of instructions are collected with local analysis. The results show that in a function, most of the repeated instructions operate on data passed to functions as parameters. Data that come from global variables are also popular among repeated instructions. Once again, great repeatability is observed in instructions operating in data “hardwired” into the program, as well as instructions that read or write the stack pointer and instructions performing loads or stores of callee-saved registers.

In addition to that, in local analysis, instructions are categorized according to their position in the function. Usually, the instructions appearing in a function’s *prologue* and *epilogue* are most repeatable (because of the callee-saved registers).

Instruction repetition information could be very valuable when trying to speedup execution. For example, a small buffer could be used in order to take advantage of the fact that most of the repetitions are contributed by only a small fraction of static instructions. Other benefits of dynamic inspection of program code and data could be: (i) Detection of repeatability that cannot be predicted statically. (ii) Dynamic loading of popular addresses of the memory. (iii) Exploitation of repeatability of function arguments. (iv) Avoidance of register pressure. (v) Memoization (to remember a function’s output), since there are many repetitions of a function call using the same arguments.

Instruction repetition information could also be used in *value prediction* and *dynamic instruction reuse*, described in Sections 1.3 and 1.2 respectively.

On the other hand, some benefits could also be gained by the static exploitation of instruction repetition. There are many difficulties that

prevent this from happening though (like the fact that most of instruction repetition is revealed during runtime). But, compiler optimizations (like loop unrolling) could help in order to overcome some of these obstacles.

1.2 Dynamic Instruction Reuse [7] (Kyriakos Karenos)

Motivation Programs are generally written to execute and produce output dynamically for different inputs. In addition, operations on data structures tend to be compact and comprehensive. These two basic empirical observations suggest that as a program executes dynamically, a number of instructions with the same input may reappear. *Dynamic Instruction Reuse* [7], attempts to take advantage of this recurrence by storing the results of previous instruction in a *Reuse Buffer (RB)* to later retrieve and reuse them if the *reuse test* for that instruction succeeds. Although this may sound more like a software (compiler) responsibility, a hardware-based approach allows for proactive enhancements within the pipeline itself.

Two categories of instruction reuse motivate this work. *Squash reuse* refers to cases of reusing the results of instructions that were calculated as part of speculative execution which would have otherwise been discarded upon misprediction. *General reuse* refers to the cases where dependence chains exist among instructions: in this case since previous results are known (thus consumers need not wait for the producers) the dependence in the chain are implicitly eliminated and thus instructions may execute in parallel.

Instruction Reuse Mechanism The RB, indexed by the PC, is accessed during ID stage. If the *reuse test* succeeds the instruction effectively *skips* the EX stage. After commit, entry *invalidation* actions are taken, mentioned below. To realize this technique, three schemes are proposed:

Scheme S_v simply checks whether the operand *values* for some instruction are the same as with the buffered instruction. Memory operations can be handled for loads (address calculation as well as load result) but the RB entry is invalidated

when a store occurs for a specific memory address.

Scheme S_n implements the reuse test by checking register *names* rather than values. Thus, in addition to memory invalidations, a result present in the RB for some instruction must also be invalidated if a register upon this instructions is dependent is modified.

Scheme S_{n+d} attempts to resolve the *dependences* that are likely to hinder the performance of S_n . A *Register Source Table* is utilized in order to hold, for each register, the index of the RB, which contains the instruction which last produced that register's value. Similar to register renaming, dependent instructions need not be invalidated upon register update (only on instruction eviction) since dependence status is known.

Experimental Results Experiments firstly showed that a considerable percentage of instructions can be reused (average of 50%). The basic dimension used was buffer size whereas basic measures where reuse percentage, speedups and dependence resolution latency. S_{n+d} and S_v for larger buffers generally perform better due to lower invalidation frequencies. The microarchitecture interactions affect the overall performance thus speedups where not as impressive as reuse. However, the number of cycles waiting for an operand to be ready was reduced due to early dependency resolution (chain collapse).

1.3 Exceeding the Dataflow Limit via Value Prediction [4] (Thomas Repantis)

Introduction True data dependences between instructions impose the dataflow limit on the parallelization of instructions of sequential programs. Value Prediction [4] [5] [3] is a speculative technique to overcome the dataflow limit. In a taxonomy of speculative techniques, *Value Prediction* would fall in the category of data (not control) speculation. It differs from other data speculative techniques, in that it does not speculate on the data location (the address the data will be stored), but on the actual value the data will have.

Value Locality Value Prediction tries to exploit a characteristic that has been observed in the execution of many programs, namely Value Locality. *Value Locality* refers to the probability for a previously-seen value to occur again/repeatedly in a storage location. In the current case the Value Locality of general-purpose or floating-point registers after instructions that write to them is examined. Because real-world programs are designed to be general, they suffer from performance penalties, which are partially the result of a significant portion of register values being repeated. The register value locality that different benchmarks exhibit was measured, by counting the number of repetitions in the register values written by the static instructions. Even with a history depth of one (i.e. when checking only for matches against the last written value), it was found that most of the programs have value locality near 50%. Specific instruction types, such as integer and floating-point double loads, are more predictable in the values they produce.

Value Prediction Mechanism By exploiting the fact that the results of many instructions can be predicted, dependent instructions can be scheduled speculatively and may be able to complete execution sooner. To achieve this, two mechanisms were implemented: one to accurately predict the results (the Value Prediction Unit) and one to verify these predictions when the actual results are available.

The *Value Prediction Unit (VPU)* is structured in two levels, to enhance the accuracy of the predictions. The first level generates the prediction values, while the second decides if the predictions are likely to be accurate. The VPU consists of two direct-mapped tables, which are indexed by the instruction address (the PC) of the instruction being predicted. The *Classification Table (CT)* has a *valid field* to indicate valid entries and a *prediction history*, a saturating counter of correct and incorrect predictions, to define instructions as predictable or not. The *Value Prediction Table (VPT)* has –apart from a *valid field*– a *value history field*, where the results of instructions are stored, using LRU.

The mechanism for verifying the predictions

and for recovering from mispredictions (Figure 7 of [4]) enables the parallel issue and execution of dependent instructions. Only after the predicted result is verified, does a dependent instruction complete. The misprediction penalty is at most one additional cycle, along with structural hazards that might have been introduced by the misprediction.

A number of parameters to fine-tune the performance (hit rate) of the VPU include the number of entries in the VPT, the number of entries in the CT and the number of bits of the prediction history counter (which add hysteresis in the classification of the instructions as predictable or not). By measurements, the best choices for those parameters were decided to be 4096, 1024, and 2 accordingly.

Experiments To measure the performance of the VPU, three simulation models were implemented, two based on the PowerPC 620 and one based on an idealized model without structural dependencies. The experiments, consisting of trace generation, VPU simulation, and microarchitectural simulation, showed an average speedup of 22.7%, 3 to 4 times more than what would be achieved by just doubling the L1 data cache.

2 Part II: Analysis

2.1 Understanding the Differences Between Value Prediction and Instruction Reuse [9]

Both Instruction Reuse (IR) and Value Prediction (VP) try to reduce the execution time by exploiting the redundancy in programs to collapse the data dependences between instructions. However, the approaches of the two techniques are very different. VP *speculates* on the results of instructions (which are inputs for other instructions). This speculation is based on the previous results and the predicted values are used by the following instructions. When the actual results from the execution are available, the speculation is verified. On the other hand, IR *does not speculate* on the results of instructions; it just

detects if the current computation has been performed before and then uses the previous result. Therefore, the execution of the current instruction is avoided and furthermore no verification of the result is necessary. In order to achieve that, the results of instructions are stored in a Reuse Buffer and their validity and applicability on the current situation is verified by a reuse test, that examines if the current operands are the same as those used to calculate the results. It is therefore evident that IR verifies the results before using them (*early validation*), while VP after using them speculatively (*late validation*).

The above differences in the approach result in differences in the amount of redundancy the two techniques can capture and in differences in the way they interact with other microarchitectural features.

Since IR validates results early, it is more conservative in the amount of redundancy it captures. To reuse an instruction, it needs the inputs of the current instruction to be available and to be the same as the inputs of the instruction stored in the Reuse Buffer. VP on the contrary does not need the inputs of the current instruction and therefore can provide a speculation in more cases than IR.

Value misprediction has a penalty in VP, since all the instructions depending on the mispredicted value must be re-executed. IR on the other hand does not suffer from mispredictions, since it is not speculative.

Both VP and IR help in resolving branches earlier and thereby in decreasing the misprediction penalty. IR may reduce this penalty further, by detecting the misprediction early and –more importantly– by storing the results of the instructions that get squashed, which might be useful later. VP on the other hand may increase the branch misprediction penalty, by causing more mispredictions (if the branch is resolved while its operands are still speculative), or by delaying the branch prediction (if the branch is resolved only after its operands are verified).

Since reused instructions do not execute, IR reduces the demand for resources like functional units, cache ports etc. VP on the contrary may increase the demand for resources, since instruc-

tions which execute with wrong inputs need to re-execute.

IR decreases the execution latency, since the reuse of an operation takes just one cycle (instead of the number of cycles needed for the execution), whereas the completion of instructions in VP is delayed by the execution and the verification latency.

2.2 Repeatability and Performance

Cross-referencing the reuse percentages reported in [8] and the speedups observed in [9] we find that in many cases high reusability does not infer corresponding speedups (e.g. “m88ksim” benchmark with 98.8% dynamic instruction reuse has lower average speedup than “go” with 85.2%). Two reasons for this are inducted: (i) The limitations in the ability of IR and VP to capture redundancy. An improved solution might be an intelligent buffering technique –possibly combining IR and VP– taking into account empirical observations such as high repeatability in instructions in local scope (functions) whose inputs are arguments and global data values. (ii) Effects of microarchitecture interactions. Only detailed cycle-by-cycle simulation of the entire microarchitecture can verify performance improvements.

3 Part III: Thorough

3.1 The Predictability of Data Values [6] (Yannis Drougas)

Value Prediction, presented in Section 1.3, tries to exploit instruction repetition in order to speedup program execution. In order to do that, a method to predict the result of an instruction is utilized. One could significantly increase the speedup by improving (increasing the predictability of) this method.

In [6], three different instruction result predictors are compared. They predict the instruction result based on the results of previous executions of the same instruction. Let v_i be the result of the i th execution of the instruction. The predictors

are: (i) **Last Value (LV)**: The predicted result value v_i is the value v_{i-1} of the result of the last execution of the instruction. (ii) **Stride**: The predicted value v_i is equal to $v_{i-1} + (v_{i-1} - v_{i-2})$. These predictors perform better when for example, we have a variable in a loop, increased by a steady number in each iteration. (iii) **Finite Context Method (FCM)**: Statistics about the frequency of appearance of each value are kept, given the results of previous executions of the instruction. The value returned is the most likely to occur, based on these statistics and the results of the previous executions.

Simulations performed using the integer SPEC'95 benchmarks show that, generally $\text{Predictability}(\text{FCM}) > \text{Predictability}(\text{Stride}) > \text{Predictability}(\text{LV})$ holds. Moreover, FCM's predictability increases with an increase of the number of previous results that are taken into account.

3.2 Compiler-Directed Dynamic Computation Reuse [2] (Kyriakos Karenos)

We have discussed that intelligent buffering techniques for exploiting reusability are required. The *Compiler-directed Computational Reuse (CCR)* scheme proposes that this intelligence is fed to the hardware level by means of software, i.e. the compiler. This technique can exploit higher granularity repeatability, termed *regional computation reusability* instead of just instruction level reusability seen in IR. Therefore, support must be provided at both the hardware and software levels. At the hardware level a *Computation Reuse Buffer (CRB)* is employed that stores a number of *computational instances (CI)* within each entry, that contain the set of inputs and result values. CIs can be reused if all register values involved are identical to a previous computation. At the software level, the compiler is enhanced to identify computation correlations via heuristics and instruct the hardware of *Reusable Computational Regions (RCRs)* within the code.

Experimental results have shown the potential benefits from this technique since it was observed that 40% of static computations account for al-

most 90% of total reuse. Speedups have been measured to reach up to 30%. Improvements are primarily attributed to the fact that CCR can escape the upper bound of IR within a single block to whole computational regions.

3.3 Highly Accurate Data Value Prediction using Hybrid Predictors [10] (Thomas Repantis)

One [10] of the proposals [1] to improve the accuracy of Value Prediction includes four techniques, two of them being hybrid, and thus performing well for more benchmarks.

The first technique, Stride-based Value Prediction, monitors the stride by which the results of consecutive instances of an instruction vary and –if that stride is constant– speculates on the results of future instances. This works well because of the use of loops and arrays in programs.

The second technique, Two-Level Value Prediction, stores (as a pattern) a maximum of 4 most recent unique results of an instruction (instead of just the last one), and decides which one –if any– of those to present as a predicted value, by taking into account the history of the previous outcomes (the number of times a pattern has been repeated). If the number of repetitions is less than a threshold, no prediction is made.

The first hybrid predictor combines the original (last outcome-based) Value Prediction and the Stride Prediction. It predicts the last outcome when there have not been enough repetitions to calculate the stride, and zero in the initial repetition.

The second hybrid predictor combines the Two-Level Value Prediction and the Stride Prediction. It uses the Stride Prediction when the Two-Level Prediction is not able to propose a result (when the number of times a value has been repeated is less than a threshold).

The hybrid predictors have a higher percentage of correct predictions, and the last of them a lower percentage of mispredictions as well.

As a critical note we should state, that just as in branch prediction, higher hit rates may not necessarily translate into fewer execution cycles [4].

References

- [1] Brad Calder, Glenn Reinman, and Dean M. Tullsen. Selective value prediction. In *ISCA*, pages 64–74, 1999.
- [2] Daniel A. Connors and Wen mei W. Hwu. Compiler-directed dynamic computation reuse: Rationale and initial results. In *International Symposium on Microarchitecture*, pages 158–169, 1999.
- [3] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, California, 3rd edition, 2003. pages 257–259.
- [4] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *International Symposium on Microarchitecture*, pages 226–237, 1996.
- [5] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.
- [6] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *International Symposium on Microarchitecture*, pages 248–258, 1997.
- [7] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *ISCA*, pages 194–205, 1997.
- [8] Avinash Sodani and Gurindar S. Sohi. An empirical analysis of instruction repetition. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 35–45. ACM Press, 1998.
- [9] Avinash Sodani and Gurindar S. Sohi. Understanding the differences between value prediction and instruction reuse. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 205–215. IEEE Computer Society Press, 1998.
- [10] Kai Wang and Manoj Franklin. Highly accurate data value prediction using hybrid predictors. In *International Symposium on Microarchitecture*, pages 281–290, 1997.