Published in the Proceedings of the 26th International Symposium on Computer Architecture, May 1999

Storageless Value Prediction Using Prior Register Values

Dean M. Tullsen John S. Seng

Dept of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{tullsen,jseng}@cs.ucsd.edu

Abstract

This paper presents a technique called register value prediction (RVP) which uses a type of locality called register-value reuse. By predicting that an instruction will produce the value that is already stored in the destination register, we eliminate the need for large value buffers to enable value prediction. Even without the large buffers, register-value prediction can be made as or more effective than last-value prediction, particularly with the aid of compiler management of values in the register file.

Both static and dynamic register value prediction techniques are demonstrated to exploit register-value reuse, the former requiring minimal instruction set architecture changes and the latter requiring a set of small confidence counters. We show an average gain of 12% with dynamic RVP and moderate compiler assistance on a next generation processor, and 15% on a 16-wide processor.

1. Introduction

This paper presents techniques for identifying and using an alternate type of data locality, *register-value reuse*. Register-value prediction (RVP) identifies instructions that typically produce values that are already in the register file. The results of those instructions can be predicted using the values in the register file. This eliminates the need for large storage buffers for values, and keeps the prediction data right where it is needed, in the register file and close to the functional units.

All previously proposed value prediction schemes rely on large buffers to store potential values. Often, the assumed buffer is larger than current on-chip data caches. For example, a value prediction scheme with a 2K-entry buffer on a 64-bit processor requires 16KB of storage for the value buffer and an additional 9-13 KB for the tags, depending on the size of physical addresses. For such a mechanism to be viable, it must provide more performance than, for example, significantly increasing the size of the data cache. Lipasti and Shen [7] consider just this tradeoff to justify value prediction.

The value prediction mechanism described in this paper works with *no* added storage for values, yet is flexible enough to encompass a wider range of reuse patterns than most previously-proposed buffer-based prediction schemes.

In this research, we only assume the presence of hardware to exploit same-register reuse. The instruction uses the old value in the destination (architectural) register as a prediction for the new value produced. We depend on compiler transformations to convert other forms of register or value reuse into same-register reuse. This requires no changes to the ISA to indicate the location of the value to be used for the prediction – the destination register defines both the destination of the result and the source of the prediction. Thus, if an instruction that writes R3 is identified as predictable, it will have predicted correctly if the value written to R3 is the same value that was there before the instruction executed. Many times this happens naturally, but the compiler can make it happen much more often.

Figure 1 shows the percentage of time the value produced by a load instruction was already in the register file for the SPEC95 benchmarks. When it is already in the same register, the compiler need do nothing to take advantage of it with the hardware support we propose. When it is in another dead register (a register whose value will not be read again before it is written), it is likely that we could exploit it with a different register allocation. When it is in a live register, a simple move instruction could put the value in place for register-based value prediction. Even these scenarios do not exhaust the possibilities, as the particular values that are visible depend very much on the current register allocation; for example, the last column includes last-value reuse, which indicates the value was recently in the same register. At least 75% of the time, the value loaded from memory is either already in the register file, or was recently there.

Register-based (storage-less) value prediction certainly has limitations. It cannot track a large number of values (more than the size of the architectural register file) at once. It relies in some cases on compiler assistance, and in particular can only exploit more complex reuse patterns if the compiler or profiler can recognize them. However, it also

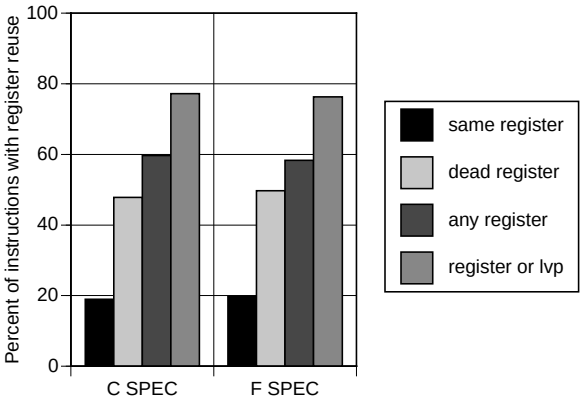

Figure 1. The degree of register-value reuse for loads in the SPEC suite. The graph is the percentage of time the loaded value is already in the same register, a register that is dead, any register, or either in a register or was the last value seen by the load.

has several inherent advantages over buffer-based value prediction:

1. No hardware cost for storage of values.

2. Fewer additional datapaths in the core of the processor. In particular, it requires no new datapaths to communicate predicted values to the processor core. Predicted values are read from the register file, like other operands.

3. It can be actively managed. Software has complete control over what values/variables are in what register.

4. No stale values. With buffer-based prediction, we must either insert speculative values in the buffer and possibly pollute it, or we must hold off inserting values until they become non-speculative, forcing new instructions to possibly use stale entries. With register-based prediction, register renaming and branch recovery keep the register map updated, so that we always read the most recent and correct version of the register (and if it has not been produced yet, we wait for it via normal register dependence mechanisms).

5. It can use the result of one instruction as a prediction for another. Standard value prediction only predicts based on the history of a single instruction — it cannot exploit correlated variables/instructions. Memory renaming [16, 11, 12] can identify correlated stores and loads, but register-based prediction is even more general. For our purposes we define correlated variables as ones that consistently hold the same value (as each other, it need not be the same actual value) over time.

The notion of register-value reuse was previously introduced and measured by Gabbay and Mendelson [4]. In contrast to that work, we show that register-value reuse can be greatly increased by the compilation process, and we present an architecture that can exploit register-value reuse with much greater effectiveness than the architecture proposed in [4], which is described in the next section.

This paper presents both static and dynamic techniques to exploit register-value reuse through RVP. The static techniques are applied only to load instructions, but we apply the dynamic techniques first to loads only, then to all instructions. With the static techniques we show the *potential* for as much as a 22% gain over the baseline hardware. For the dynamic techniques, we show the potential for as much as a 26% gain, and average 2% more performance than (a moderate implementation of) last-value prediction, despite dramatically lower hardware costs. Those results are for a next-generation 8-issue processor. For a more aggressive 16-issue processor, the gains are higher, both over no-prediction and over traditional last-value prediction.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes how register-based value prediction supports several models of value locality. Section 4 presents the architectural support we assume to enable register-based value prediction, and Section 5 describes the register-reuse profiling which supports our simulations. Section 6 gives our measurement methodology. The results for both static and dynamic RVP are shown in Section 7. We conclude the paper in Section 8.

2. Related Work

Lipasti and Shen [7] and Lipasti, et al. [8] introduce the concept of value prediction, and in particular last-value prediction (LVP) to exploit value locality. They use an untagged last-value prediction table and saturating confidence counters to predict values (as well as more idealized schemes).

Gabbay and Mendelson [4] also proposed value prediction to exceed inherent ILP limits, adding the concept of stride prediction and register-file prediction. Their register-file predictor is the closest predecessor to our dynamic register-value predictor, but they show it to have significantly less predictability than their other predictors, and do not pursue it further. However, our register-value predictor has key differences from theirs. The Gabbay register predictor is still a value file responsible for communicating values to the instructions, while ours never writes predicted values (from an external source) into the register file or functional units. But most importantly, Gabbay's confidence counters are associated with the register value file entries and are indexed by register number. In that scheme, register-value reuse is only available if it remains high for all definitions of the register. Our confidence counters are indexed by instruction PC, thus it only requires a single instruction to exhibit register-value reuse to be able to exploit it. Section 7 demonstrates a dramatic difference in prediction coverage and accuracy for the two approaches.

In addition, that study did not consider the ability of the compiler to create same-register locality or to turn other-register locality into same-register locality.

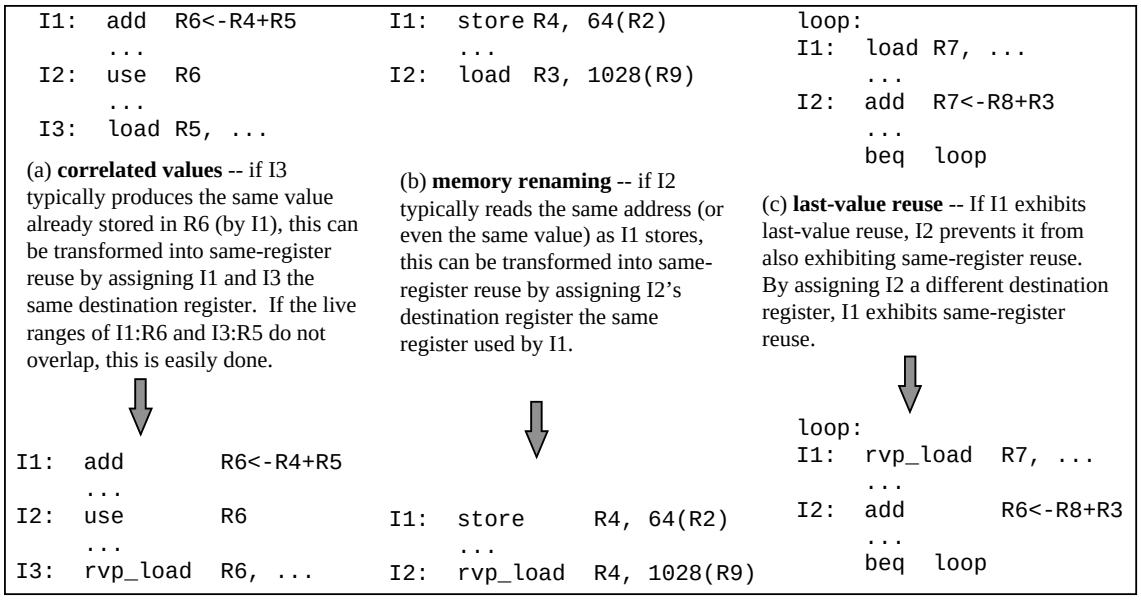

Figure 2. Three reuse patterns that can be turned into same-register reuse by register allocation.

Jourdan, et al. [6] also recognize the existence of register value reuse and propose that it can be used to implement physical register sharing among logical registers, as well as result reuse for possible prevention of execution of certain instructions. They do not exploit any speculative techniques, such as value prediction, although they mention the possibility. They depend on hardware to recognize other-register value-reuse, where we transform the program to turn it into same-register reuse. Their technique could be combined with ours to increase the effectiveness of RVP without compiler intervention.

Value speculation scheduling (Fu, et al. [3]) uses explicit instructions to manage value prediction hardware, allowing the compiler to select instructions (loads) for prediction, like our static RVP scheme. However, they use separate instructions for the load and the prediction, allowing software scheduling of the prediction. They also do misprediction recovery in software. They do not exploit any value reuse patterns besides same-instruction value reuse; however, because their scheme uses a buffer (like conventional value prediction), they can incorporate more sophisticated hardware mechanisms, like stride and context-based predictors.

Tyson and Austin [16] and Moshovos and Sohi [11] each present hardware schemes specifically aimed at predicting load values from recent stores. Register-value prediction can take advantage of memory communication through renaming simply by loading into the same register the value was stored from. Our scheme currently exploits that only when profiling finds the stored value still in a register, but more aggressive compilation could ensure that correlated stores and loads use the same register without intervening writes of the register.

Calder, et al. [1] and Gabbay and Mendelson [5] show that value locality can be profiled efficiently. They also show that static value locality is highly predictable across different inputs, which we also found.

Martin, et al. [10] and Lo, et al. [9] also recognize the utility of dead registers. Martin, et al. seek to identify dead registers in hardware to avoid writing useless information into them, while Lo, et al. make dead registers available for renaming. We are attempting to find ways to put the dead registers to work with useful data.

3. Exploiting Value Locality With Prior Register Values

Register-based value prediction (RVP) uses the previous value in the instruction's destination register as a prediction for the new result. Deciding which instructions to predict can be done statically or dynamically. This paper examines both approaches. In the static case, we assume that only loads are predicted. Thus, only a few extra opcodes need be found to support static register-based value prediction.

For example, instruction `load R3, 800(R5)` would be replaced by `rvp_load R3, 800(R5)`. Subsequent instructions that use `R3` will use the prior value of `R3` as a prediction and can issue immediately if the 'old' value is available. This technique can support several types of value locality:

Correlated Values — If the values of two variables are highly correlated, we can use register-based prediction to use one to predict the other. If they have non-overlapping lifetimes, they can be assigned the same register (See Figure 2a). If not, a single move instruction before the `rvp_load` can provide the correlated prediction.

Memory Renaming — Memory Renaming identifies stores and loads that typically go to the same address, and

passes the value from the store to the load through a structure similar to the value prediction buffer, but enhanced with even more tables. See Figure 2(b) for how we support this simply with RVP.

Last Value Locality — In cases where there has not been an intervening write to the destination register, any instruction that exhibits last-value predictability also exhibits register-value predictability. Figure 2(c) deals with the case when there are intervening writes.

Constant Locality — In some cases, constant prediction is more accurate than last-value prediction. For example, in reading a sparse matrix where most entries have value zero, predicting each value to be zero can have fewer mispredictions than last-value prediction. Constant prediction can be accomplished simply by moving a constant into the register.

Et Cetera — Stride prediction can be accomplished with the insertion of an add instruction. For path-based correlation (a variable is highly correlated with a different variable along two different paths), all three variables could be assigned the same register. Other re-use patterns could also be implemented.

This study does not take full advantage of most of these opportunities, only using profile-based knowledge of existing register-value reuse and last-value reuse. The potential for optimization beyond what this study assumes is great.

Dynamic register-based value prediction can also support all of these prediction models. It still takes advantage of compiler support to increase register reuse (through some of the above techniques), but performs well in many cases with no compiler support whatsoever.

4. Architectural Support

None of the register-value prediction schemes we simulate require storage for values. This section describes the hardware support that is needed.

Static register-based value prediction of loads requires new opcodes to identify predictable instructions. Dynamic RVP requires counters for confidence tracking and (possibly) extra register read ports when predicting instructions besides loads. Also, any form of value speculation will require misprediction recovery mechanisms; we model three different mechanisms, and introduce them here.

These discussions assume instruction-queue based dynamic instruction scheduling, where instructions read registers when issued or receive the values from forwarding.

4.1. Static Register Prediction

With static register-value prediction, candidates for prediction are identified with new opcodes (we assume only loads, but there is no reason rvp-versions of a few common or long-latency instructions could not also be added). When an instruction marked for prediction goes through register renaming, it is assigned a new physical destination register. That register name is not written into the register map (or if it is, it is written into an extra field), but a field is written indicating the mapping is speculative dependent on the `rvp_load`. Subsequent instructions that read the virtual register will read the old physical mapping, but know that they are speculative. They will execute and read the previous value of the register if it is available, otherwise they will wait for it.

The load instruction that was marked for prediction also takes the old mapping as a source operand, because it will read that value from the register file to be compared against the new value. This still only gives the load two source operands in most RISC processors. If the speculation is successful, the new register mapping can be abandoned in favor of the old, otherwise the old mapping must be replaced by the new.

4.2. Dynamic Register Prediction

The hardware mechanisms for dynamic register prediction are similar to static prediction, using the same register-mapping mechanism to provide the predicted values and to identify and correct mispredictions.

Since no ISA changes are required for dynamic RVP, we consider both restricting prediction to loads and the option of predicting all instructions. Non-load instructions that are predicted would require an extra register read port to read the predicted value (to identify mispredictions). This is not as heavy a burden as it may seem. One or two extra read ports would limit the number of predictions per cycle, but place no limit on the number of instructions that can use predicted values. In our simulations of dynamic RVP for all instructions, we average less than 0.2 predictions per cycle for the *drvp_all* results and about 0.5 predictions per cycle for the *drvp_all_dead_lv* results, so a single extra read port would likely suffice for the architecture we assume.

Dynamic prediction must be able to identify candidates for prediction. This is done with confidence counters and no value storage. Confidence counters are associated with *instructions* rather than *registers*. Therefore, if only one of multiple instructions that define a register has high register-value reuse, we will only predict that instruction. We assume a direct-mapped table (1K entries) of 3-bit counters indexed by the instruction PC. We assume that the confidence counters are not tagged with the PC; we have modeled results from both cases to confirm that untagged counters actually outperform tagged.

4.3. Misprediction Recovery

If an instruction is correctly predicted, we will immediately clear the field in the register map that indicates the mapping is speculative and then release the unused register mapping when the instruction is committed.

If an instruction is mispredicted, more significant measures must be taken. We examine three techniques, of in-

creasing complexity. The first two require only that we can identify the first instruction that uses a predicted value. The third requires that we track all dependences between predictions and subsequent instructions. In the latter two cases we need the ability to broadcast prediction results, and possibly new register mapping data, to dependent instructions waiting in the instruction queue (IQ).

These recovery schemes are not specific to RVP, and match reasonably well with others discussed in studies such as [7] and [16], so we will omit the details. The three recovery schemes are:

Refetch — a value mispredict is treated like a branch mispredict. Instructions beginning with the first-use of the predicted value are squashed, and the fetch unit is responsible for getting them back in the machine.

Reissue — all instructions after the first-use are kept in the IQ until they are no longer speculative, and may re-issue from there with minimal delay in case of a mispredict.

Selective Reissue — only instructions dependent on the predicted value (either directly or indirectly) are kept in the IQ until the prediction is resolved.

In the first case, the cost of a value misprediction is the same as a branch mispredict. In the other two cases, we assume a single cycle cost for checking the value — a dependent instruction will issue one cycle later after a mispredict than it would if the previous instruction were not predicted.

For the first scheme, it is sufficient to roll back the register map to the point of prediction, with the correct register mapping for the now non-speculative predicted instruction. For the other two, we must correct the register mapping for the predicted instruction on the fly, and signal all other speculative instructions that they must re-issue when their dependences are satisfied.

5. Register-reuse profiling

Although there will certainly already be some same-register value reuse in existing programs, the real power of this technique comes from the ability of the compiler to create register-value reuse. To understand the potential for this technique, we must make some assumptions about what the compiler will be able to do.

We profile each of the applications and create four lists of instructions that have (1) same-register value reuse, (2) high correlation with a value in a dead register, (3) high correlation with a value in a live register, and (4) high last-value predictability. In static register prediction, we assume the compiler has used this information both to mark predictable instructions and to alter register allocation to create register-value predictability. For dynamic RVP, predictable instructions are identified by hardware, but we assume for some results that one or more of the latter three lists have been used to alter register allocation.

We assume the compiler will take advantage of these types of reuse in the following ways. High correlation with a dead register can be exploited by changing the register allocation of the destination of the current instruction to match that of the dead register. The compiler can simply combine the live ranges for the purpose of register allocation as long as the live range of the current register definition does not overlap the live range of the previous definition. Otherwise, the compiler could break up live ranges or use the following technique for live registers.

When there is high correlation with a live register, the new value cannot be written into the same register because we know the live ranges overlap; however, the new register is always dead before it is written. Any value can be written into the register while it is dead to make the instruction predictable. Since the correlated value is already in a register, a simple register move accomplishes this. In our simulations we do not account for the latency of such a move, since many times it can be put in a place that will not impact the critical path of execution. Therefore, these results will represent a somewhat optimistic upper bound. For that reason, we show the results of the *live* optimization in only one graph (Figure 3).

When an instruction has high last-value locality, the compiler can automatically expose register-value predictability by ensuring that no other instructions define the same register between executions of the instruction. This is easy if the loop is small, but will place constraints on register allocation if the loop is large and there are many loads with last-value predictability.

These profiles do not expose all of the potential for compiler-created reuse. In addition to ignoring many of the possibilities listed in Section 3, variables that have high value correlation but are separated by a register write to the wrong register will not be visible to our current techniques.

For most of the results in Section 7, we make the optimistic assumption that the compiler will be able to expose all of the reuses (of a particular type) found by the profiler. For example, if an instruction is identified in our *dead* list as exhibiting value reuse with another register, we track reuse of the value in the other register for that instruction (to determine prediction success). For all unlisted instructions (in the dynamic case) we only track same-register reuse. In Section 7.3, we show that a realistic model of compiler-based register reallocation to exploit dead-register and last-value reuse does in fact achieve most of the potential performance (shown as the *dead_lv* results in Section 7), even in the more difficult case of all instructions being candidates for prediction.

6. Evaluation Methodology

All measurements are done on an execution-driven instruction-level simulator of a processor with a 9-stage pipeline and a 7-cycle misprediction penalty. Our simula-

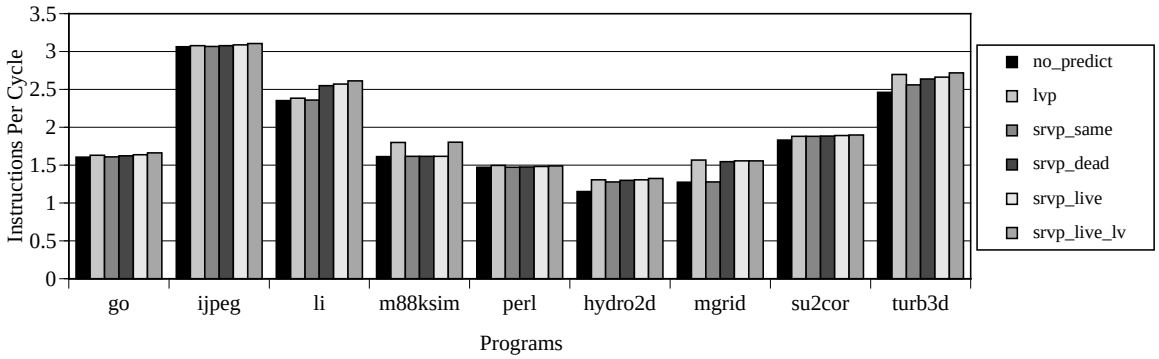

Figure 3. Static register-based value prediction on SPEC95 programs.

Inst queue size	32 int, 32 fp
Functional units	6 integer (4 can perform loads/stores); 3 fp
Pipeline	9 stages, 7-cycle branch mispredict
Branch prediction	256-entry BTB, 2K x 2-bit PHT, gshare
Fetch Bandwidth	Eight instructions
onchip L1 I cache	32KB, 4-way SA, 64-byte lines; 20-cycle miss pen.
onchip L1 D cache	32KB, 4-way SA, 64-byte lines; 20-cycle miss pen.
off-chip L2 cache	512KB, 2-way SA, 64-byte lines; 80-cycle miss pen.

Table 1. Processor parameters used in the simulator

tor is derived from the SMT simulator [14], but configured for single-thread execution. The simulator executes unmodified Alpha object code and models the execution pipelines, memory hierarchy, TLBs, and the branch prediction logic of the processor. We extended this capability to allow both buffer-based and register-based value prediction.

The processor we model (Table 1) is not particularly aggressive, because we realize the advantages of RVP are most attractive while the size of the value prediction table is still prohibitive; however, we show that RVP also has performance advantages that make it attractive beyond that time frame.

For last-value prediction, we assume a 1K-entry last-value prediction buffer with a 3-bit confidence counter assigned to each entry. We use resetting counters with a confidence threshold of 7. This means we only predict after we have seen seven consecutive hits. This is a conservative filter, but is consistent with our machine model, reducing the pressure prediction places on the instruction queues. It is also consistent with the thresholds we use for the profiler to select instructions for static prediction (80% predictability, except for figure 4, which uses 90%). The exact same counters are used for dynamic RVP.

Our workload consists of nine of the SPEC95 benchmarks. We compiled each program with the Digital Unix C (or FORTRAN) compiler under DEC OSF V4.0, with full optimization. Each program is simulated for 300 million committed instructions, except su2cor which runs for 3 billion due to a very long initialization period.

For static register-based prediction, we identify instructions for prediction through profiling, as described in the previous section. Identification of same-register reuse is relatively fast, but identifying correlations with other registers is slower; however, this paper is attempting to explore the limits of our prediction approach, so we made no attempt to find shortcuts to achieve more reasonable profile time. Many of the techniques from [1] could be applied.

We used profiles of the SPEC95 train data set to both mark instructions (for the static techniques) and to guide our assumptions about compiler-based re-allocation of registers; we then used that data for the measurement simulations on the ref data sets.

7. Results

This section details the results of our simulations for static register value prediction, dynamic RVP for loads, and dynamic RVP for all instructions. Other results shown are no-prediction, dynamic last-value prediction using the last-value prediction table, and in one case, Gabbay and Mendelson's register predictor [4].

7.1. Static RVP

Figure 3 shows the results for the selective re-execute recovery mechanism without value prediction (*nopredict*), with dynamic last-value prediction using a 1K-entry value prediction table (*lvp*), and with static RVP assuming various levels of compiler support for register allocation: no support (*srvp_same*), dead-register correlation optimization (*srvp_dead*), live-register correlation optimization (*srvp_live*), and *srvp_live* combined with last-value optimization (*srvp_live_lv*).

Because a key advantage of RVP prediction is the drastic reduction in required storage over even the simplest last-value predictors, we do not compare it with schemes that add additional storage and complexity to what is required for last-value prediction. Examples of such schemes are

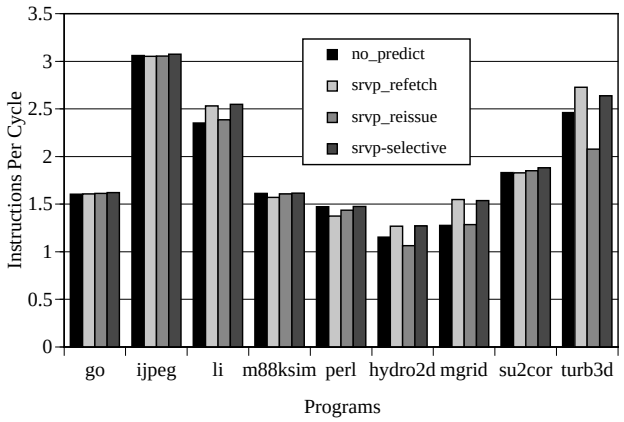

Figure 4. The effect of the recovery mechanism on the performance of value prediction. All RVP results are for the *dead* optimization.

stride predictors [4], context-based/two-level/hybrid predictors [17, 13], or memory renaming architectures [16, 11].

In three of the nine programs, enough register value locality already exists in unmodified code to achieve performance gains of 3% or more. In most of the programs, further gains are enabled by the compiler optimizations. *li* gains another 8% with the dead-register optimization, and *mgrid* gains 21%.

7.1.1 Performance of Recovery Mechanisms

The mispredict recovery mechanism had a significant impact on our results. These results are applicable to more than just register value prediction. The mechanisms, described in Section 4 are *refetch*, the scheme with the highest misprediction cost, *reissue*, and *selective* reissue. In each case, the *srvp_dead* result is shown.

Figure 4 shows that the relatively simple refetch scheme performs well on this architecture, often outperforming reissue by large margins and occasionally beating selective reissue. While refetch has the highest mispredict cost, it also imposes the least pressure on the architecture for a correct prediction. The other two schemes force instructions to stay in the queue as long as they remain speculative in the interest of reissuing them quickly. This prevents other instructions from getting into the machine, restricting parallelism and negating the advantage of the prediction. Because the selective mechanism holds fewer instructions in the queue, however, and has the low mispredict penalty, it still provides the best overall performance. We will use the selective scheme in the rest of the simulations, but recognize that refetch represents an attractive alternative due to its low complexity. A higher prediction threshold was used for this figure (90% instead of 80%) to mark instructions, even for the selective results, because *refetch* and *reissue* require more conservative prediction.

7.2. Dynamic RVP

Static RVP requires (small) ISA changes and depends heavily on the compiler to mark the right loads. Dynamic prediction, which uses the hardware to identify instructions with register-value reuse, requires no changes to the ISA. It also is more tolerant of compiler "mistakes" — if an instruction the compiler thought would have high predictability turns out not to, nothing is lost because it will not be predicted by the hardware. This will allow the compiler to actually be more aggressive in identifying and exposing possible register-value reuse through register allocation and other techniques. It also works in many cases with no compiler support at all.

Dynamic RVP, described in Section 4, requires a set of small confidence counters but no value storage. Although RVP could have many more confidence counters than the number of value file entries in an LVP implementation because the counters are so small, we assume the same number of confidence counters (1K, direct-mapped indexing).

In addition to the value storage buffers needed for dynamic LVP, we also assume dynamic LVP buffer entries are tagged with the PC, which improves performance. Tagging entries detects interference in the table to inhibit predictions in that case. We do not assume tags for the RVP confidence counters (which would eliminate much of the hardware savings). Unlike LVP, RVP actually performs (slightly) better without the tags. With RVP, positive interference can be exploited when there are no tags, as long as both instructions that map to the same confidence counter experience register-value reuse. With LVP, positive interference only occurs in the rare case when both instructions experience last-value reuse and the values are the same. This is an important result, because an LVP value file becomes virtually useless for a loop that is larger than the value prediction table due to interference, but that is not true for the RVP counters.

Figure 5 shows the performance of dynamic RVP applied only to load instructions, compared to last-value prediction for loads. RVP-dead only slightly under-performs the much more expensive last value prediction, while RVP-dead-lv outperforms LVP somewhat, achieving an 8% average gain over no prediction.

With dynamic prediction, we do not have to restrict ourselves to load instructions because of ISA restrictions. Any instruction that writes a register can exhibit register-value reuse. Figure 6 shows the result for LVP and dynamic RVP applied to all instructions. Dynamic RVP can be quite powerful in this scenario, with the dead-register plus last-value (*dead_lv*) reallocation results providing 12% more performance than no prediction. Even the dead-register optimization alone provides more performance than the buffer-based last-value prediction mechanism. Also included in these results is register-value prediction using the Gabbay and

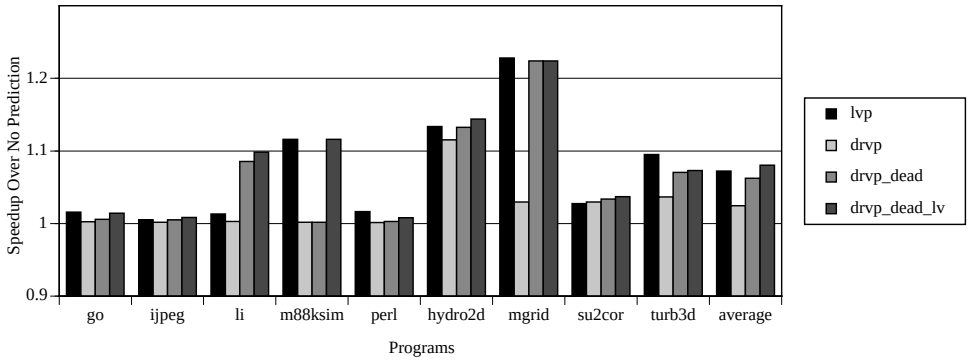

Figure 5. The performance of dynamic register-based value prediction for load instructions on the SPEC programs.

	% Insts Predicted/Pred. Rate			
	drvp			
	dead	dead_lv	lvp	G&M RP
go	4/93.7	15/95.7	14/94.8	.3/95.9
hydro	22/99.3	38/99.7	28/99.2	7/98.3
ijpeg	15/98.8	10/98.9	12/98.4	2/97.8
li	9/98.6	26/99.1	24/98.2	.4/91.1
m88k	29/99.8	64/100	57/99.9	3/98.4
mgrid	7/99.9	9/99.7	7/99.4	4/97.9
perl	18/99.1	4/95.2	16/98.8	.4/87.5
su2	9/96.9	22/99.2	21/98.2	1/94.1
tu3d	28/98.2	38/99.0	32/98.4	8/94.4

Table 2. The percentage of instructions that were predicted and the prediction accuracy for RVP, last-value prediction, and the Gabbay register predictor.

Mendelson scheme (*Grp_all*), but without the stride predictor they include, to equalize comparisons. That register predictor suffers from high interference on the predictors, as every instruction that writes a register shares the same counter.

Table 2 shows the prediction coverage for the *rvp_dead* algorithm, the last-value predictor, and the Gabbay and Mendelson scheme. In comparing the RVP and LVP results, we see that both get extremely high accuracy from the conservative resetting counters with the threshold set at seven. Comparing the table with our performance results (Figure 6), we see that there is more correlation between coverage and performance than accuracy and performance. However, neither is a particularly good predictor of performance. In [15] it is shown that overall predictability of instructions is not necessarily highly correlated with the predictability of instructions on the critical path, the ones that impact performance.

For both dynamic RVP techniques, we have shown that the low hardware cost of register-based value prediction is in no way a barrier to high prediction accuracy and significant performance gains, being more than competitive with much more expensive techniques.

7.3. Register Re-allocation to Support Register Reuse

The results shown do not represent an upper bound on the performance available with register reuse. In fact, we take only limited advantage of opportunities for value reuse. However, the results shown do represent upper bounds for the performance available given the register reallocations suggested by the particular profiles we use.

In this section, we will examine the compiler's ability to exploit the dead-register and last-value optimizations through register reallocation. Clearly not all register-reuse opportunities can be accommodated with a given legal register allocation. For example, two neighboring instructions may both exhibit reuse with the same register. If the first instruction's result is still live when the second executes, they cannot both be allocated to the same register as the dead value. If too many instructions within a loop exhibit last-value reuse, not all can be assigned an exclusive register within the loop.

For these results we filter out register-reuse patterns that cannot be supported with a legal register allocation using traditional register coloring techniques [2]. We start with the dead-register and last-value profile data and attempt to support as many of the reuses as possible through register reallocation. No instructions are added to move data between live registers — those reuses are considered unusable.

For each procedure we create a register interference graph using live range analysis, assuming that all non-volatile registers are live at entrance and exit, and that each procedure call uses all argument registers. Then the interference graph is supplemented by each register redefinition suggested by our profiles. For dead-register reuse, the live range of the register defined by the instruction is combined with the live range of the instruction that was the primary producer of the value in the reused register, thus assigning them the same color for the register allocation. We need a new profile to identify the primary producer of a value, since only the identity of the register was needed for our other results. At this point, many reuses are found to be

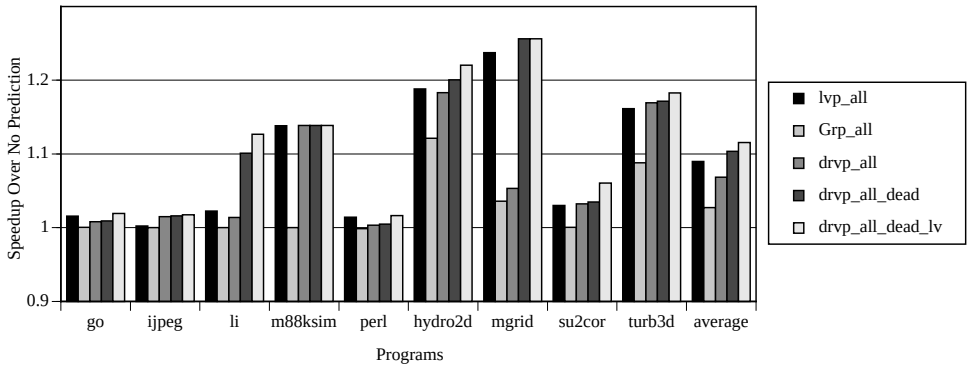

Figure 6. The performance of dynamic register-based value prediction for all instructions on the SPEC programs.

illegal because the live ranges already conflict in the interference graph. Although we know the producing instruction's value is dead when the re-using instruction executes, it may be that the re-using instruction's live range wraps around and overlaps the producing instruction. Also, previous register-reuse or last-value reuse graph changes may have created the conflict. Regardless of the reason, the instruction reuse is abandoned if the live ranges cannot be combined. We also do not allow reuse of registers defined in other procedures, although we manually made a handful of exceptions when the dead value was in a volatile register and the mapping was otherwise legal.

If an instruction exhibits last-value reuse (LVR), we create an interference edge with every instruction in the innermost loop containing the instruction. Any instruction that is not in a loop within the procedure is abandoned. Occasionally the LVR instruction already shares a color with another instruction in the loop — this also makes the LVR unusable.

This process can add significantly to the edges in the interference graph. If it is possible to find a legal coloring for the register graph, we allow the remaining register reuses that have not already been found illegal to be used. If the graph cannot be colored with 31 registers, register reuses are removed until the coloring succeeds. We use a combination of heuristics to choose reuses to abandon. Starting with the highest priority heuristic, we:

1. Remove LVR reuse before register reuse. LVR typically adds *much* more complexity to the graph.

2. Start at outer loops (particularly with LVR) and move inward. LVR instructions in outer loops are executed with less frequency and add more complexity than inner-loop LVR.

3. Use critical-path profiles [15] to gauge the importance of each instruction. This quantifies each instruction's contribution to the critical data dependence path through the entire program. Many instructions do not contribute at all to the critical path and are good candidates to be removed.

By the time the register reallocation is done, we typically have thrown out over half of the register reuses; however,

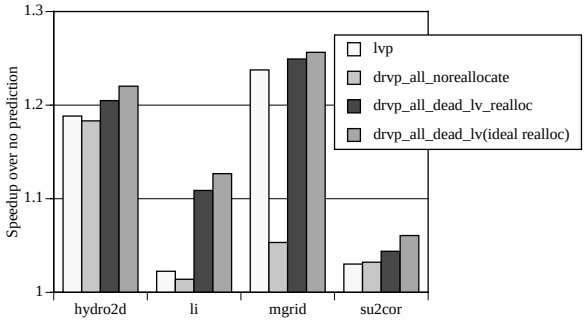

Figure 7. Speedup over no prediction with a more realistic model of register reallocation, compared to LVP, no reallocation, and ideal reallocation.

the automatically-eliminated ones often are not the important ones (e.g., LVR that is not in a loop), and our heuristics hopefully guard the important instructions during the pruning to enable coloring. Figure 7 shows the results for the four applications where there was a significant difference between DRVP with and without ideal reallocation. Among those left off this graph are *ijpeg* and *m88ksim* that get all of their performance gain without any need for compiler assistance.

Compiler-based register reallocation appears able to generate most of the performance potential uncovered by our profiles. In each case where traditional last-value prediction outperformed the base DRVP result, the register reallocation was sufficient to exceed it.

7.4. More Aggressive Architectures

The results presented up to this point have focused on a near-future architecture, assuming the cost of a buffer-based value prediction scheme will become less of an issue over time. In this section we show that register value prediction will continue to be competitive with last-value prediction even when the hardware differences are minimized, due purely to performance advantages.

Figure 8 examines the performance of RVP on a more aggressive processor architecture. This architecture has

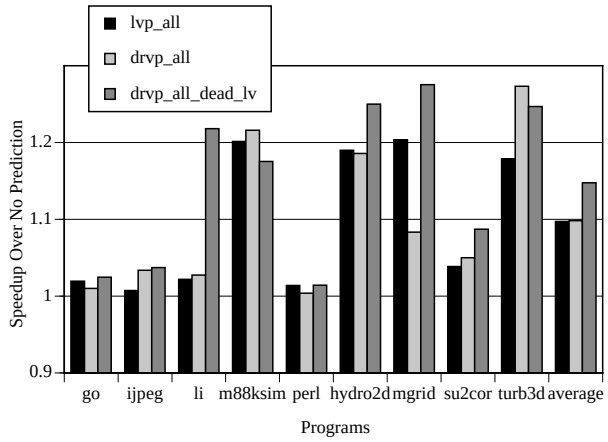

Figure 8. The performance of value prediction mechanisms on a more 16-wide processor architecture.

double the instruction queue entries, functional units, renaming registers, and fetch bandwidth of the processor modeled for all prior results. It also has the ability to fetch up to three basic blocks per cycle to take advantage of the increased fetch bandwidth.

In removing many of the limitations to instruction-level parallelism existent in the previous processor, the performance of RVP increases, both over no-prediction (15% performance gain) and over traditional last-value prediction (5% higher performance). In fact, RVP with no compiler support (*rvp_all*) provides equal performance to the last-value architecture.

8. Conclusions

The programs we have studied exhibit register-value reuse. We have shown techniques for register-based value prediction that have the potential to achieve significant speedup with only a small fraction of the hardware cost of previously proposed value prediction schemes. These speedups are not only over no-prediction, but even over a non-aggressive (but still very expensive) buffer-based last-value prediction scheme.

While recently proposed hardware-based value prediction mechanisms have become more and more sophisticated (and expensive), they all depend solely on the history of a single instruction. The simple mechanism we propose enables an unlimited number of other inputs to prediction, including the values of other variables and the path taken through the program. We have modeled the bounds of some simple techniques to expose register-value reuse through register re-allocation to combine correlated values.

We demonstrate both static and dynamic techniques for a processor to exploit register-value reuse. On a next-generation processor with realistic limits on fetch bandwidth, instruction queue size, and issue rates, RVP can achieve speedups up to 12% on average over no prediction. With a more aggressive processor model, the potential speedup is 15% over no prediction.

Acknowledgments

We would like to thank the anonymous reviewers for their useful comments. This work was funded in part by NSF CAREER grant No. MIP-9701708, NSF grant No. CCR-980869, and a grant from Compaq Computer Corporation.

References

[1] B. Calder, P. Feller, and A. Eustace. Value profiling. In *30th International Symposium on Microarchitecture*, December 1997.

[2] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Programming Languages*, 6(1):47–57, 1981.

[3] C. Fu, M. Jennings, S. Larin, and T. Conte. Value speculation scheduling for high performance processors. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[4] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. EE Department TR 1080, Technion - Israel Institue of Technology, Nov. 1996.

[5] F. Gabbay and A. Mendelson. Can program profiling support value prediction? In *30th International Symposium on Microarchitecture*, Dec. 1997.

[6] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *31st International Symposium on Microarchitecture*, Nov. 1998.

[7] M. Lipasti and J. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, Dec. 1996.

[8] M. Lipasti, C. Wilkerson, and J. Shen. Value locality and load value prediction. In *17th International Conference on Architectural Support for Programming Languages and operating Systems*, pages 138–147, Oct. 1996.

[9] J. Lo, S. Parekh, S. Eggers, H. Levy, and D. Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, to appear.

[10] M. Martin, A. Roth, and C. Fisher. Exploiting dead value information. In *30th International Symposium on Microarchitecture*, pages 125–135, Dec. 1997.

[11] A. Moshovos and G. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *30th International Symposium on Microarchitecture*, Dec. 1997.

[12] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin. Profile guided load marking for memory renaming. Technical Report Technical Report UCSD-CS98-593, University of California, San Diego, July 1998.

[13] Y. Sazeides and J. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, 1997.

[14] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[15] D. Tullsen and B. Calder. Computing along the critical path. Technical report, University of California, San Diego, Oct. 1998.

[16] G. Tyson and T. Austin. Improving the accuracy and performance of memory communication through renaming. In *30th Annual International Symposium on Microarchitecture*, pages 218–227, Dec. 1997.

[17] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, pages 281–290, Dec. 1997.