.

Exploiting Value Locality to Exceed the Dataflow Limit

Corresponding author:
Mikko H. Lipasti
IBM Server Group
Department DDCA/040-3
3605 Highway 52 North
Rochester, MN 55901
E-mail: mikko@us.ibm.com
Phone: (507) 253-0813 Fax: (507) 253-4901

John Paul Shen
Department of Electrical and Computer Engineering
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
shen@ece.cmu.edu

Abstract

The serialization constraints imposed by true data dependences have always been regarded as an absolute dataflow limit on the parallel execution of serial programs. This paper describes value prediction, a new technique that allows data dependent instructions to issue and execute in parallel without violating program semantics. This technique exploits value locality, or the likelihood of the recurrence of a previously-seen value within a storage location inside a computer system. Value prediction consists of predicting entire 32- and 64-bit register values based on previously-seen values. We find that values loaded from memory or generated by ALU instructions are frequently predictable. Furthermore, we show that simple microarchitectural enhancements to a modern microprocessor implementation based on the PowerPC 620 that enable value prediction can effectively exploit value locality to collapse true dependences, reduce average memory and result latencies, and provide average performance gains of 3%-23% by exceeding the dataflow limit.

Keywords: speculative execution, value prediction, superscalar processors, dataflow limit

Exploiting Value Locality to Exceed the Dataflow Limit

1. Motivation and Related Work

There are two fundamental restrictions that limit the amount of *instruction level parallelism* (*ILP*) that can be extracted from sequential programs: *control flow* and *data flow. Control flow* limits *ILP* by imposing serialization constraints at forks and joins in a program's control flow graph [1]. *Data flow* limits *ILP* by imposing serialization constraints on pairs of instructions that are data dependent (i.e. one needs the result of another to compute its own result, and hence must wait for the other to complete before beginning to execute). Examining the extent and effect of these limits has been a popular and important area of research, particularly in the case of control flow [2,3,4]. Continuing advances in the development of accurate branch predictors (e.g. [5]) have led to increasingly-aggressive control-speculative microarchitectures (e.g. the Intel Pentium Pro [6]), which undertake aggressive measures to overcome control-flow restrictions by using branch prediction and speculative execution to bypass control dependences and expose additional instruction-level parallelism to the microarchitecture. Meanwhile, numerous mechanisms have been proposed and implemented to eliminate false data dependences and tolerate the latencies induced by true data dependences by allowing instructions to execute out of program order (see [7] for an overview).

Surprisingly, in light of the extensive energies focused on eliminating control-flow restrictions on parallel instruction issue, less attention has been paid to eliminating data-flow restrictions on parallel issue. Recent work has focused primarily on reducing the latency of specific types of instructions (usually loads from memory) by rearranging pipeline stages [8], initiating memory accesses earlier [9], or speculating that dependences to earlier stores do not exist [10, 11, 12, 13].

The most relevant prior work in the area of eliminating data-flow dependences consists of the *Tree Machine* [14,15], which uses a *value cache* to store and look up the results of recurring arithmetic expressions to eliminate redundant computation (the *value cache*, in effect, performs *common subexpression elimination* [1] in hardware). Richardson follows up on this concept in [16] by introducing the concepts of *trivial computation*, which is defined as the trivialization of potentially complex operations by the occurrence of simple operands; and *redundant computation*, where an operation repeatedly performs the same computation because it sees the same operands. He proposes a hardware mechanism (the *result cache*) which reduces the latency of such trivial or redundant complex arithmetic operations by storing and looking up their results in the *result cache*.

In [17], we first described *value locality*, a concept related to *redundant computation*, and demonstrated a technique--*Load Value Prediction*, or *LVP*--for predicting the results of load instructions at dispatch by exploiting the affinity between load instruction addresses and the values the loads produce. *LVP* differs from Harbison's *value cache* and Richardson's *result cache* in two important ways: first, the LVP table is indexed by instruction address, and hence value lookups can occur very early in the pipeline; second, it is speculative in nature, and relies on a verification mechanism to guarantee correctness. In contrast, both Harbison and Richardson use table indices that are only available later in the pipeline (Harbison uses data addresses, while Richardson uses actual operand values); and require their predictions to be correct, hence requiring mechanisms for keeping their tables coherent with all other computation.

In this paper, we summarize our investigation into LVP and extend it from predicting the results of load instructions to predicting all instructions that write an integer or floating point register [18]. We show that a significant proportion of such writes are trivially predictable; describe a *value prediction* hardware mechanism that allows dependent instructions to execute in parallel; and present results that demonstrate significant performance increases over our baseline machine models.

2. Taxonomy of Speculative Execution

In order to place our work on value prediction into a meaningful historical context, we introduce a taxonomy of speculative execution. This taxonomy, summarized in Figure 1, categorizes ours as well as previously-introduced techniques based on which types of dependences are being bypassed (control vs. data), whether the speculation relates to storage location or value, and what type of decision must be made to enable the speculation (binary vs. multi-valued).

2.1. Control Speculation

There are essentially two types of control speculation: speculating on the direction of a branch, which requires a binary decision (taken vs. not-taken); and speculating on the target of a branch, which requires a multi-valued decision (the target can potentially be anywhere in the program's address space). Examples of the former are any of the many branch prediction schemes explored in the literature (e.g. [19,5]), while examples of the latter are the Branch Target Buffer (*BTB*) or Branch Target Address Cache (*BTAC*) units included on most modern high-end microprocessors (e.g. the PowerPC 604/620 [13] or the Intel Pentium Pro [6]).

2.2. Data Speculation

Data speculation techniques break down logically into two categories: those that speculate on the storage location of the data, and those that speculate on the actual value of the data. Furthermore, techniques that speculate on the location come in two fundamentally different flavors: those that speculate on a specific attribute of the storage location (e.g. whether or not it is aliased with an earlier definition), and those that speculate on the address of the storage location. An example of the former is *speculative disambiguation*, which optimistically assumes that an earlier definition does not alias with a current use, and provides a mechanism for checking the accuracy of that assumption. Speculative disambiguation has been implemented both in software [11] as well as in hardware [10, 12, 13]. Another example of this type of speculation occurs implicitly in most control-speculative processors, whenever execution proceeds speculatively past a join in the control-flow graph where multiple reaching definitions for a storage location are live [1]. By speculating past that join, the processor hardware is implicitly speculating that the definition on the predicted path to the join in question is in fact the correct one (as opposed to the definition on an alternate path).

There are a large number of techniques that speculate on data address. Most prefetching techniques, for example, are speculative in nature and rely on some heuristic for generating addresses of future memory references (e.g. [20][21][22]). Of course, since prefetching has no architected side effects, no mechanism is needed for verifying the accuracy of the prediction or for recovering from mispredictions. Another example of a technique that speculates on data address is *fast address calculation* [9], which enables early initiation of memory loads by speculatively generating addresses early in the pipeline.

The final category in our taxonomy, techniques that speculate on data value, has received less attention in the literature. The first work we are aware of is value prediction, initially described in [17]. This paper also falls squarely into the data-value-speculative category, since it is an extension of the LVP approach. Note that neither the *Tree Machine* [14][15] or Richardson's work [16] qualify since they are not speculative.

3. Value Locality

In this paper, we revisit the concept of *value locality*, which we first introduced in [17] as the likelihood of a previously-seen value recurring repeatedly within a storage location. Although the concept is general and can be applied to any storage location within a computer system, we have limited our current study to examine only the value locality of general-purpose or floating point registers immediately following instructions that write to those reg-

isters. A plethora of previous work on static and dynamic branch prediction (e.g. [19][5]) has focused on an even more restricted application of value locality, namely the prediction of a single condition bit based on its past behavior. In [17], we first examined the value locality of registers being targeted by loads from memory. This paper, an expansion of [18], summarizes and continues that work, extending the prediction of load values to the prediction of all integer and floating point register values.

Intuitively, it seems that it would be a very difficult task to discover any useful amount of value locality in a general purpose register. After all, a 32-bit register can contain any one of over four billion values--how could one possibly predict which of those is even somewhat likely to occur next? As it turns out, if we narrow the scope of our prediction mechanism by considering each static instruction individually, the task becomes much easier and we are able to accurately predict a significant fraction of register values being written by machine instructions.

What is it that makes these values predictable? After examining a number of real-world programs, we assert that value locality exists primarily for the same reason that *partial evaluation* [23] is such an effective compile-time optimization; namely, that real-world programs, run-time environments, and operating systems incur severe performance penalties because they are *general by design*. That is, they are implemented to handle not only contingencies, exceptional conditions, and erroneous inputs, all of which occur relatively rarely in real life, but they are also often designed with future expansion and code reuse in mind. Even code that is aggressively optimized by modern, state-of-the-art compilers exhibits these tendencies. We have made the following empirical observations about the programs we examined for this study, and feel that they are helpful in understanding why value locality exists:

- **Data redundancy**: Frequently, the input sets for real-world programs contain data that has little variation. Examples of this are sparse matrices, text files with white space, and empty cells in spreadsheets.
- **Bit masking**: Many instructions significantly reduce the information content of the values they process. For example, shift- and bit-masking operations often extract only a few bits of information from one or more 32-bit inputs.
- **Arithmetic reduction**: Similarly, many arithmetic operations significantly reduce the information content of the values they process. For example, integer divide and modulo operations often extract only a few bits of information from 32-bit inputs.
- **Trivial operands**: As pointed out by Richardson [16], several instructions are susceptible to trivialization by certain input operand values. One example is integer multiplication by zero--shown by Richardson to be surprisingly common--which always results in the same result value.
- **Error-checking**: Checks for infrequently-occurring conditions often compile into instructions that generate or consume values that are effectively run-time constants.
- **Program constants**: It is often more efficient to generate code to load program constants from memory than code to construct them with immediate operands.
- **Computed branches**: To compute a branch destination, for e.g. a switch statement, the compiler must generate code to load a register with the base address for the branch, which is a run-time constant.
- **Virtual function calls**: To call a virtual function, the compiler must generate code to load a function pointer, which is a run-time constant.
- **Glue code**: Due to addressability concerns and linkage conventions, the compiler must often generate glue code for calling from one compilation unit to another**.** This code frequently contains loads of instruction and data addresses that remain constant throughout the execution of a program.
- **Addressability**: To gain addressability to non-automatic storage, the compiler must load pointers from a table that is not initialized until the program is loaded, and thereafter remains constant.

- **Call-subgraph identities**: Functions or procedures tend to be called by a fixed, often small, set of functions, and likewise tend to call a fixed, often small, set of functions. As a result, loads that restore the link register as well as other callee-saved registers can have high value locality.
- **Memory alias resolution**: The compiler must be conservative about stores aliasing loads, and will frequently generate what appear to be redundant loads to resolve those aliases.
- **Register spill code**: When a compiler runs out of registers, variables that may remain constant are spilled to memory and reloaded repeatedly.

Naturally, many of the above are subject to the particulars of the instruction set, compiler, and run-time environment being employed, and it could be argued that some of them could be eliminated with changes in the ISA, compiler, or run-time environment, or by applying link-time or run-time code optimizations (e.g. [24][25]). However, such changes and improvements have been slow to appear; the aggregate effect of above factors on value locality is measurable and significant today on the modern RISC ISA that we examined, which provides a state-of-the-art compiler and run-time system. It is worth pointing out, however, that the value locality of particular static instructions in a program can be significantly affected by compiler optimizations such as loop unrolling, loop peeling, 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. A similar effect on load latencies (i.e. per-static-load cache miss rates) has been reported by Abraham et al. in [26].

Our results--which agree with Richardson's persuasive arguments and results in [16]--show that even code that is aggressively optimized by modern, state-of-the-art compilers exhibits these tendencies and contains significant amounts of value locality.

The benchmark set we use to explore value locality and quantify its performance impact is summarized in Table I. We chose thirteen integer benchmarks, five from SPEC '92, one from SPEC '95, along with two image-processing applications (cjpeg and mpeg), two commonly-used Unix utilities (gawk and grep), GNU's perfect hash function generator (gperf), a more recent version of GCC (cc1-271), and a recursive quicksort. In addition, we chose four of the SPEC '92 floating-point benchmarks. All benchmarks are compiled at full optimization with the IBM CSET reference compilers for the PowerPC ISA, and are run to completion with the input sets described, but do not include supervisor-state instructions, which our tracing tool is unable to capture.

3.1. Load Value Locality

Figure 2 shows the value locality for load instructions in each of the benchmarks. The value locality for each benchmark is measured by counting the number of times each static load instruction retrieves a value from memory that matches a previously-seen value for that static load and dividing by the total number of dynamic loads in the benchmark. Two sets of numbers are shown, one (light bars) for a history depth of one (i.e. we check for matches against only the most-recently-retrieved value), while the second set (dark bars) has a history depth of sixteen (i.e. we check against the last sixteen unique values)[1]. We see that even with a history depth of one, most of the integer programs exhibit load value locality in the 50% range, while extending the history depth to sixteen (along with a hypothetical perfect mechanism for choosing the right one of the sixteen values) can improve that to better than 80%. What this means is that the vast majority of static loads exhibit very little variation in the values that they load during

1. The history values are stored in a direct-mapped table with 1K entries indexed but not tagged by instruction address, and the values (one or sixteen) stored at each entry are replaced with an LRU policy. Hence, both constructive and destructive interference can occur between instructions that map to the same entry.

the course of a program's execution. Unfortunately, three of our benchmarks (cjpeg, swm256, and tomcatv) demonstrate poor load value locality.

The cjpeg benchmark nearly doubles its value locality as the history depth increases to sixteen, indicating that there is some temporal variation in the image input data being loaded, but that there is a relatively small set of unique values. We attribute the poor value locality of swm256 and tomcatv to the synthetic pseudo-random input data used in these benchmarks.

To further explore the notion of value locality, we collected data that classifies loads based on the type of data being retrieved: floating-point data, non-floating-point data, instruction addresses, and data addresses (pointers). These results are summarized in Figure 3. Once again, two sets of numbers are shown for each benchmark, one for a history depth of one (light bars), and the other for a depth of sixteen (dark bars). In general, we see that address loads tend to have better locality than data loads, with instruction addresses holding a slight edge over data addresses, and integer data loads holding an edge over floating-point loads.

3.2. Register Value Locality

Figure 4 shows the *register value locality* for all instructions that write an integer or floating point register in each of the benchmarks. The register value locality for each benchmark is measured by counting the number of times each static instruction writes a register value that matches a previously-seen value for that static instruction and dividing by the total number of dynamic register writes in the benchmark. Two sets of numbers are shown, one (light bars) for a history depth of one (i.e. we check for matches against only the most-recently-written value), while the second set (dark bars) has a history depth of four (i.e. we check against the last four unique values).[1] We see that even with a history depth of one, most of the programs exhibit value locality in the 40-50% range (average 49%), while extending the history depth to four (along with a perfect mechanism for choosing the right one of the four values) can improve that to the 60-70% range (average 61%). What that means is that a majority of static instructions exhibit very little variation in the values that they write during the course of a program's execution.

To further explore the notion of value locality, we collected value predictability data that classifies register writes based on instruction type (the types are summarized in Table II). These results are summarized in Figure 5. Once again, two sets of numbers are shown; one for a history depth of one, and another for a history depth of four. Integer and floating-point double loads (I_LD and FPD_LD) are the most predictable frequently-occurring instructions. FP_OTH, FP_MV, MC_MV are also very predictable but make up an insignificant portion of the dynamic instruction mix. For the single-cycle instructions, fewer input operands (one vs. two) correlate with higher value locality. For the multi-cycle instructions, however, the opposite is true.

The worst value locality is exhibited by the load-floating-point-single instructions. We attribute this to the fact that the floating-point benchmarks we used initialize input arrays with pseudo-random numbers, resulting in poor value locality for loads from these arrays.

The store-with-update (ST_U) instruction type also has poor value locality. This makes sense, since the ST_U instruction is used to step through an array at a fixed stride (hence the base address register is updated with a different value every time the instruction executes, and history-based value prediction will fail). On the other hand, ST_U is also used in function prologues to update the stack frame pointer, where, given the same call-depth, the value is predictable from one call to the next. Hence, some of our call-intensive benchmarks report higher value locality for ST_U. However, the former effect dominates and lowers the overall value locality for ST_U.

---

1. As with the data collected for load value locality, the table used was finite, and hence, the potential exists for both constructive and destructive interference between instructions that map to the same entry.

4. Exploiting Value Locality

The fact that loads from memory as well as all register writes--across a variety of programs--demonstrate a significant degree of value locality opens up exciting new possibilities for the microarchitect. Since the results of many instructions can be accurately predicted before they are issued or executed, dependent instructions are no longer bound by the serialization constraints imposed by operand data flow. Instructions can now be scheduled speculatively with additional degrees of freedom to better utilize existing functional units and hardware buffers, and are frequently able to complete execution sooner since the critical paths through dependence graphs have been collapsed. However, in order to exploit value locality and bring about all of these benefits, two mechanisms must be implemented: one for accurately predicting values--the VP (value prediction) unit--and one for verifying these predictions.

4.1. The Value Prediction Unit

Value prediction is useful only if it can be done accurately, since incorrect predictions can lead to increased structural hazards and longer latency (the misprediction penalty is described in greater detail in Section 5.3). Hence, we propose a two-level prediction structure for the VP Unit: the first level is used to generate the prediction values, and the second level is used to decide whether or not the predictions are likely to be accurate.

The internal structure of the VP Unit is summarized in Figure 6. The VP Unit consists of two tables: the *Classification Table* (CT) and the *Value Prediction Table* (VPT), both of which are direct-mapped and indexed by the instruction address (PC) of the instruction being predicted. Entries in the CT contain two fields: the *valid* field, which consists of either a single bit that indicates a valid entry or a partial or complete tag field that is matched against the upper bits of the PC to indicate a valid field; and the *prediction history*, which is a saturating counter of 1 or more bits. The prediction history is incremented or decremented whenever a prediction is correct or incorrect, respectively, and is used to classify instructions as either predictable or unpredictable. This classification is used to decide whether or not the result of a particular instruction should be predicted. Increasing the number of bits in the saturating counter adds hysteresis to the classification process and can help avoid erroneous classifications by ignoring anomalous values and/or destructive interference.

The VPT entries also consist of two fields: a *valid* field, which, again, can consist of a single valid bit or a full or partial tag; and a *value history* field, which contains one or more 32- or 64-bit values that are maintained with an LRU policy. The *value history* fields are replaced when an instruction is first encountered (by its result) or whenever a prediction is incorrect (by the actual result). The VPT replacement policy is also governed by the CT prediction history to introduce hysteresis and avoid replacing useful values with less useful ones.

As a preliminary exploration of the VP Unit design space, we analyzed sensitivity to a few key parameters, and then selected several design points to use with our microarchitectural studies (see Section 7). However, the intent of this paper is not to explore the details of such a design; rather, our intent is to explore the larger issue of the impact of value prediction on microarchitecture and instruction-level parallelism, and to leave such details to future work.

In Figure 7, we show the sensitivity, when predicting all register writes, of the VPT hit rate to size for each of our benchmarks. We see that for most benchmarks, the hit rate levels off at or around 4096 entries, though in several cases significant improvements are possible beyond that size. Nevertheless, we chose 4096 as one of our design points, since going beyond that size (i.e. 4096 entries x 8 bytes/entry = 32KB) seemed unreasonable without severely impacting processor cycle time.

For our realistic Load Value Prediction experiments, we chose a smaller VPT size of 1024 entries, since the load footprint is significantly smaller than--roughly 25% of--the footprint of all register-writing instructions. For our *Limit* configuration (see Table IV) we also simulated a table with 4096 entries to get an indication of performance potential.

The purpose of the CT is to partition instructions into two classes: those that are predictable by the VPT, and those that are not. To measure its effectiveness at accomplishing this purpose, we simulated a number of different CT configurations.

To evaluate CT efficacy for register value prediction, we simulated the CT sizes summarized in Table III. The state descriptions specify the effect of each state on both value prediction as well as the replacement of values in the VPT when new values are encountered. The results for the VP configurations are summarized in Figure 8. From the results, we conclude that the best choice for maximizing both the predictable and unpredictable hit rates is the *1024/3-bit* configuration (this is not surprising, since it has the highest hardware cost). However, since the *1024/2-bit* configuration is only slightly worse at identifying predictable instructions and is actually better at identifying unpredictable ones (hence minimizing misprediction penalty), and is significantly cheaper to implement (it uses 1/3 fewer bits), we decided to use the latter in our microarchitectural simulation studies. We note that the unpredictable hit rates of the 3-bit configurations are worse (relative to the 1- bit and 2-bit configurations) than their predictable hit rates, and conclude that this must be because the 3-bit state assignments heavily favor prediction (see Table III). Changing the state assignments would improve the unpredictable hit rate at the expense of reducing the predictable hit rate.

For our experiments with Load Value Prediction, we simulated the first four configurations shown in Table IV (*SimpleLVP, ConstantLVP, LimitLVP,* and *PerfectLVP*). Table IV shows the VPT size (column 2), history depth (column 3), number of CT entries (column 4), and the size of each saturating counter (column 5). The saturating counters assign the available states as described in Table III. In Table V, we show the percentage of all unpredictable loads the LCT is able to classify as unpredictable (columns 2 and 4) and the percentage of predictable loads the LCT is able to correctly classify as predictable (columns 3 and 5) for the *SimpleLVP* and *LimitLVP* configurations. Overall, the CT configurations shown are capable of identifying between 61% and 99% of all predictable loads, and between 44% and 100% of unpredictable loads. Clearly, there is significant room for improvement in CT design.

4.2. Verifying Predictions

Since value prediction is by nature speculative, we need a mechanism for verifying the correctness of the predictions and efficiently recovering from mispredictions. This mechanism is summarized in the example of Figure 9, which shows the parallel execution of two data-dependent instructions. The producer instruction, shown on the left, has its value predicted and written to its rename buffer during the *fetch* and *dispatch* cycles. The consumer instruction, shown on the right, reads the predicted value from the rename buffer at the beginning of the *execute* cycle, and is able to issue and execute normally, but is forced to retain its reservation station. Meanwhile, the predicted instruction also executes, and its computed result is compared with the predicted result during its *completion* stage. If the values match, the consumer instruction releases its reservation station. If not, completion of the first instance of the consumer instruction is invalidated, and a second instance reissues with the correct value.

4.3. Verifying Constant Loads

In our experiments with Load Value Prediction, we discovered that certain loads exhibit constant behavior; that is, they load the same constant value repeatedly. To exploit this behavior and avoid accessing the conventional memory hierarchy for these loads, we proposed the constant verification unit (CVU) [17]. To verify *predictable* loads, we simply retrieve the value from the conventional memory hierarchy and compare the predicted value to the actual value, just as we do in the more generalized value prediction scheme (see Figure 9). However, for highly-predictable or *constant* loads, we use the CVU, which allows us to avoid accessing the conventional memory system completely by forcing the VPT entries that correspond to constant loads to remain coherent with main memory (as shown in Table III, loads are classified as constant if the saturating counter at their CT entry has reached its maximum value).

For the VPT entries that are classified as constants by the CT, the data address and the index of the VPT entry are placed in a separate, fully-associative table inside the CVU (refer to Figure 10). This table is kept coherent with main memory by invalidating any entries where the data address matches a subsequent store instruction. Meanwhile, when the constant load executes, its data address is concatenated with the VPT index (the lower bits of the instruction address) and the CVU's content-addressable-memory (CAM) is searched for a matching entry. If a matching entry exists, we are guaranteed that the value at that VPT entry is coherent with main memory, since any updates (stores) since the last retrieval would have invalidated the CVU entry. If one does not exist, the constant load is demoted from *constant* to just *predictable* status, and the predicted value is now verified by retrieving the actual value from the conventional memory hierarchy. Table VI shows the percentage of all dynamic loads that are successfully identified and treated as constants. This can also be thought of as the percentage decrease in required bandwidth to the L1 data cache. We find that an average of 6% and 11% for the *Simple* and *Limit* configurations (up to 33% and 46% for some benchmarks) of loads from memory can be verified with the CVU, resulting in a proportional reduction of L1 cache bandwidth requirement. Although we were disappointed that we are unable to obtain a more significant reduction, we are pleased to note that load value prediction, unlike other speculative techniques like prefetching and branch prediction, reduces, rather than increases, memory bandwidth requirements.

5. Microarchitectural Models

In order to validate and quantify the performance impact of the Value Prediction Unit, we implemented three cycle-accurate simulation models, two of them based on the PowerPC 620 [27][13]--one which matches the current 620 closely, and one, termed the 620+, which alleviates some of its known bottlenecks--and an additional idealized model which removes all structural dependences[1]. The number of functional units and issue and result latencies for common instruction types on the three machines are summarized in Table VII. Our idealized *infinite* model also implements the following assumptions:

- Perfect caches
- Perfect alias detection and store-to-load forwarding
- Perfect instruction fetching (limited to one taken branch per cycle).
- Unit latency for mispredicted branches with no fetch bubble (i.e. instructions following a mispredicted branch are able to execute in the cycle following resolution of the mispredicted branch).

We intend the *infinite* model to match the *SP* machine model presented in [4], except for the branch prediction mechanism, which is a 2048-entry BHT design with a 2-bit saturating counter per entry, copied exactly from our 620 model. Table VIII summarizes the performance of each of our benchmarks on each of the three baseline machine models without value prediction.

5.1. PowerPC 620 Microarchitecture

The microarchitecture of the PowerPC 620 is summarized in Figure 11. Our model is based on published reports on the PowerPC 620 [27][13], and accurately models all key aspects of the microarchitecture, including branch prediction, fetching, dispatching, register renaming, out-of-order issue and execution, result forwarding, the non-blocking cache hierarchy, store-to-load alias detection, and in-order completion. To alleviate some of the bottlenecks we found in the 620 design, we also modeled an aggressive "next-generation" version of the 620, which we termed the 620+. The 620+ differs from the 620 by doubling the number of reservation stations, FPR and GPR rename buff-

1. For reasons of efficiency, the instruction window of our simulator is limited to 4096 active instructions. Hence, we did not truly model an infinite number of resources, only one that approaches that number.

ers, and completion buffer entries; adding an additional load/store unit (LSU) without an additional cache port (the base 620 already has a dual-banked data cache); and relaxing dispatching requirements to allow up to two loads or stores to dispatch and issue per cycle. In addition, we added a VP Unit that predicts register writes by keeping a value history indexed by instruction addresses, augmented with a CVU for our LVP experiments.

5.2. VP Unit Operation

The VP Unit predicts the values during fetch and dispatch, then forwards them speculatively to subsequent dependent instructions via the 620's rename buffers. Up to four predictions can be made per cycle on our 620/620+ models, while the infinite model can make up to 4096 predictions per cycle. Dependent instructions are able to issue and execute immediately, but are prevented from completing architecturally and are forced to retain possession of their reservation stations until their inputs are no longer speculative. Speculatively forwarded values are tagged with the uncommitted register writes they depend on, and these tags are propagated to the results of any subsequent dependent instructions. Meanwhile, uncommitted instructions execute in their respective functional units, and the predicted values are verified by the CVU (refer to Section 4.3) or by a comparison against the actual values computed by the instructions. Once a prediction is verified, its tag gets broadcast to all active instructions, and all the dependent instructions can either release their reservation stations and proceed into the completion unit (in the case of a correct prediction), or restart execution with the correct register values (if the prediction was incorrect). Since a large number of instructions can be in flight at the same time (16 on the base 620, 32 on the 620+, and up to 4096 in our *infinite* model), verifying a predicted value can take dozens of cycles or more, allowing the processor to speculate multiple levels down the dependence chain beyond the write, executing instructions and resolving branches that would otherwise be blocked by data-flow dependences.

5.3. Misprediction Penalty

The worst-case penalty for an incorrect value prediction in this scheme, as compared to not predicting the value in question, is one additional cycle of latency along with structural hazards that might not have occurred otherwise. The penalty occurs only when a dependent instruction has already executed speculatively, but is waiting in its reservation station for one of its predicted inputs to be verified. Since the value comparison takes an extra cycle beyond the pipeline result latency, the dependent instruction will reissue and execute with the correct value one cycle later than it would have had there been no prediction. In addition, the earlier incorrect speculative issue may have caused a structural hazard that prevented other useful instructions from dispatching or executing. In those cases where the dependent instruction has not yet executed (due to structural or other unresolved data dependences), there is no penalty, since the dependent instruction can issue as soon as the actual computed value is available, in parallel with the value comparison that verifies the prediction. In any case, due to the CT which accurately prevents incorrect predictions (see Figure 8), the misprediction penalty does not significantly affect performance.

There can also be a structural hazard penalty even in the case of a correct prediction. Since speculative values are not verified until one cycle after the actual values become available, speculatively issued dependent instructions end up occupying their reservation stations for one cycle longer than they would have had there been no prediction.

6. Experimental Framework

Our experimental framework consists of three main phases: trace generation, VP Unit simulation, and microarchitectural simulation. Traces are collected and generated with the TRIP6000 instruction tracing tool, which is an early version of a software tool developed for the IBM RS/6000 that captures all instruction, value and address

references made by the CPU while in user state. Supervisor state references between the initiating system call and the corresponding return to user state are lost. The instruction, address, and value traces are fed to a model of the VP Unit described earlier, which annotates each instruction in the trace with one of three value prediction states: no prediction, incorrect prediction, or correct prediction. The annotated trace is then fed to a cycle-accurate microarchitectural simulator that correctly accounts for the behavior of each type of instruction. All of our microarchitectural models are implemented using the VMW framework [28], which enables significant productivity gains by allowing us to reuse and retarget existing models. The VP Unit model is separated from the microarchitectural models for two reasons: to shift complexity out of the microarchitectural models and thus better distribute our simulations across multiple CPUs; and to conserve trace bandwidth by passing only two bits of state per instruction to the microarchitectural simulator, rather than the full 32/64 bit values being written.

7. Performance Results

We collected performance results for each of the three machine models described in Section 5 (base *620*, enhanced *620+*, and *infinite*) in conjunction with the nine different VP Unit configurations summarized in Table IV. Attributes that are marked *perfect* in Table IV indicate behavior that is analogous to *perfect caches*; that is, a mechanism that always produces the right result is assumed. More specifically, in the 1PerfCTVP, 4PerfCTVP and 8PerfCTVP configurations, we assume an *oracle CT* that is able to correctly identify all predictable and unpredictable register writes. Furthermore, in the LimitLVP, 4PerfCTVP and 8PerfCTVP configurations, we assume a perfect mechanism for choosing which of the 4, 8 or 16 values stored in the value history is the correct one. Moreover, we assume that the Perfect configuration can always correctly predict a value for every register write. We point out that the only VP Unit configurations that we know how to build today are *SimpleLVP*, *ConstantLVP*, and *SimpleVP*, while the other six are merely included to measure the potential contribution of improvements to both VPT and CT prediction accuracy.

7.1. PowerPC 620 with Load Value Prediction

In Figure 12, we show speedup numbers relative to the baseline 620 for two Load Value Prediction configurations that we consider realistic (i.e. buildable within one or two processor generations) as well as two idealized configurations. The two realistic configurations, *SimpleLVP* and *ConstantLVP*, are described inTable IV. To explore the limits of load value prediction, we also include results for the *LimitLVP* and *PerfectLVP* configurations (also described in Table IV). The former is similar to the *SimpleLVP* configuration, only much larger, but it is not realistic, since it assumes a hypothetical perfect mechanism for selecting which of the sixteen values associated with each load instruction address is the correct one to predict. The latter configuration, *PerfectLVP*, is able to correctly predict all load values, but does not classify any of them as constants. Neither of these configurations is buildable, but the configurations are nevertheless interesting, since they give us a sense of how much additional performance we can expect from more aggressive and accurate LVP implementations.

Overall, the speedups shown are not dramatic. We attribute this to two primary factors. First of all, from the perspective of machine parallelism, the 620 is already a relatively well-balanced machine. Upsetting this balance by eliminating data dependences merely shifts the performance bottleneck elsewhere. We attempt to offset this effect by removing some structural bottlenecks in our contrived 620+ model. Second, we observe that the limited fetch bandwidth of the 620 dramatically limits the usefulness of value prediction by delaying dependent instructions from entering the window of active instructions. In many cases this delay masks the result latency of the loads being predicted, hence obviating the usefulness of the prediction. This effect has since been examined at length by Gabbay and Mendelson [29].

However, two benchmarks (*grep* and *gawk*) stand out for the dramatic performance increases they achieve on the PowerpC 620. This gain results from the fact that both benchmarks are data-dependence bound, i.e. they have important but relatively short dependency chains in which load latencies make up a significant share of the critical path. Thus, according to Amdahl's Law, collapsing the load latencies results in significant speedups. Conversely, benchmarks which we would expect to perform better based on their high load value locality (e.g. *hydro2d* and *mpeg* on both models), fail to do so because load latencies make up a lesser share of the critical dependency paths.

Additional results for the enhanced 620+ machine model with the same LVP configurations are published elsewhere [17][30]. These results support our assertion that the increased machine parallelism of the 620+ more closely matches the parallelism exposed by load value prediction, since the relative gains for the realistic LVP configurations are nearly 50% higher than they are for the 620 baseline results shown in Figure 12.

7.2. PowerPC 620 with Register Value Prediction

In Figure 13 we show the speedups that the latter 5 VP Unit configurations of Table IV obtain over the base PowerPC 620 machine model. The *SimpleVP* configuration achieves an average speedup of 4.5% (geometric mean), the *1PerfCT*VP configuration improves that to 5.6%, *4PerfCTVP* to 6.7%, *8PerfCTVP* to 7.1%, and *PerfectVP* all the way to 11.6%. Two benchmarks, *gawk* and *grep*, demonstrate outstanding performance gains, even with the imperfect configurations, while the gains for *cjpeg* and *compress* are nonexistent, even with perfect CTs. We attribute the poor showing of *cjpeg* and *compress* to their lack of register value locality (see Figure 4).

Detailed profiling of *grep* and *gawk* revealed that both spend a significant portion of their time in the *bmexec()* and *dfaexec()* routines, which implement string search routines in loops with long dependence chains. For both benchmarks, value prediction is frequently able to break these dependence chains, resulting in significant additional parallelism.

The speedups for several benchmarks (*cc1-271*, *grep*, *perl*, *doduc*, and *hydro2d*) are quite sensitive to CT accuracy (i.e. a perfect CT produces significantly more speedup), indicating a need for a more accurate classification mechanism. In general, however, we are pleased with our results, which show that value prediction is able to produce measurable speedups on a current-generation microprocessor design.

7.3. PowerPC 620+ with Register Value Prediction

In Figure 14 we show the value prediction speedups over the baseline 620+ machine model. The *SimpleVP* configuration achieves an average speedup of 6.8% (geometric mean), the *1PerfCTVP* configuration improves that to 8.4%, *4PerfCTVP* to 9.7%, *8PerfCTVP* to 10.2%, and *PerfectVP* all the way to 15.1%. While the trends are similar to the speedups for the base 620 model, the speedups are higher across the board. We attribute this to the fact that the increased machine parallelism and additional hardware resources provided by this model better match the additional instruction-level parallelism exposed by value prediction. Furthermore, the wider execution hardware is better able to tolerate the increase in structural hazards caused by value prediction

Perhaps the most interesting observation about Figure 14 (which applies to Figure 13 as well) is the lack of any obvious correlation to Figure 4, which shows the value locality for each benchmark. This underscores the observation that a high hit rate (i.e. high value locality) does not necessarily translate into a proportional reduction in execution cycles. This follows from the fact that benchmarks with high value locality may not necessarily be sensitive to result latency (i.e. they are not data-flow-limited), whereas benchmarks with lower value locality may be very sensitive, and hence may derive significant performance benefits even if only a small fraction of register writes are predictable. For example, *eqntott* has significantly better value locality than *grep*, yet *grep* obtains significantly more speedup from value prediction.

7.4. Infinite Machine Model Speedups

In Figure 15 we show the value prediction speedups over the *Infinite* machine model. The *SimpleVP* configuration achieves an average speedup of 22.7% (geometric mean), the *1PerfCTVP* configuration improves that to 34.0%, *4PerfCTVP* to 36.9%, *8PerfCTVP* to 38.0%, and *PerfectVP* all the way to 69.8%. These numbers are very encouraging to us, since they demonstrate that the ultimate performance potential of value prediction remains largely untapped by current and even reasonably-extrapolated next generation processors, and that much work remains to be done to find more effective ways to apply it to realistic microarchitectures.

Several benchmarks that displayed measurable speedups with the finite models show negligible speedup with the infinite model (e.g. *mpeg*, *perl*, *sc*, *xlisp*), which leads us to conclude that they are not dataflow-limited by nature. However, the fact that they do show speedups with the finite models highlights the fact that value prediction, by removing serialization constraints, allows a processor to more efficiently utilize a limited number of execution resources.

We included the infinite model results to support our assertion that value prediction can be used to exceed the dataflow limit. Our infinite machine model measures the dataflow limit, since, for all practical purposes (ignoring our limit of 4096 active instructions), parallel issue in the *infinite* model is restricted only by the following three factors:

- Branch prediction accuracy
- Fetch bandwidth (single taken branch per cycle)
- Data-flow dependences

Value prediction directly impacts only the last of these, and yet we are able to demonstrate average and peak speedups of 22.7% and 198% (2.98x speedup for *gawk*) using our *SimpleVP* configuration. Hence, we lay claim to exceeding the dataflow limit.

8. VP Unit Implementation

An exhaustive design study of VP Unit design parameters and implementation details is beyond the scope of this paper. As stated earlier, some preliminary exploration of the design space was conducted by analyzing sensitivity to a few key parameters. We realize that the design selected is by no means optimal, minimal, or even reasonably efficient, and could be improved significantly with some effort. For example, we reserve a full 64 bits per value entry in the VPT, while most instructions generate only 32 or fewer bits, and space in the table could certainly be shared between such entries with some clever engineering.

However, to evaluate the practical desirability of implementing a VP Unit in a real-world processor, we compare it against one alternative approach that consumes roughly the same amount of chip space: doubling the first-level data cache to 64K by increasing the line size from 64 bytes to 128 bytes. The results of this comparison, which are shown in Figure 16, make clear that, at least for this benchmark set, value prediction delivers three to four times more speedup than doubling the data cache for both the 620 and 620+ machine models.

Furthermore, the VP Unit has several characteristics that make it attractive to a CPU designer. First of all, since the VPT and CT lookup indices are available very early, at the beginning of the instruction fetch stage, access to these tables can be superpipelined over two or more stages. Hence, given the necessary chip space, even relatively large tables could be built without impacting cycle time. Second, the design adds little or no complexity to critical delay paths in the microarchitecture. Rather, table lookups and verifications are done in parallel with existing activities or are serialized with a separate pipeline stage (value comparison). Hence, it is unlikely that VP would have an adverse effect on processor cycle time, whereas doubling the data cache would quite likely do just that.

9. Conclusions and Future Work

We make four major contributions in this paper. First, we present a taxonomy of speculative execution techniques. Second, we demonstrate that both loads from memory as well as all ALU instructions that write general purpose or floating point registers, when examined on a per-instruction-address basis, exhibit significant amounts of value locality. Third, we describe value prediction, a data-speculative microarchitectural technique for capturing and exploiting value locality to reduce data-flow restrictions on parallel instruction issue. Fourth, we demonstrate that value prediction can be used to exceed the dataflow limit by 23% (geometric mean), as measured on a processor model with no structural hazards. We are very encouraged by our results. We have shown that measurable (5% on average for the 620, 7% on average for the 620+) and in some cases dramatic (up to 33% on the 620 and 54% on the 620+) performance gains are achievable with simple microarchitectural extensions to current-generation and reasonably-extrapolated next-generation microprocessor implementations.

Acknowledgments

This work was supported by ONR grants N00014-96-1-0928 and N00014-97-1-0701. We also gratefully acknowledge the generosity of the Intel Corporation for donating numerous fast Pentium Pro-based workstations for our simulation runs. We also wish to thank the IBM Corporation for letting us use the TRIP6000 instruction tracing tool in our work.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.
2. E. M. Riseman and C. C. Foster. “The inhibition of potential parallelism by conditional jumps.” *IEEE Transactions on Computers*, pages 1405–1411, December 1972.
3. D. W. Wall. “Limits of instruction-level parallelism.” In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–189, Santa Clara, California, 1991.
4. M. Lam and R. Wilson. “Limits of control flow on parallelism.” In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 46–57, 1992.
5. T. Y. Yeh and Y. N. Patt. “Two-level adaptive training branch prediction.” In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61, November 1991.
6. R. P. Colwell and R. Steck. “A 0.6um BiCMOS processor with Dynamic Execution.” In *Proceedings of ISSCC*, 1995.
7. M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.
8. N. P. Jouppi. “Architectural and organizational tradeoffs in the design of the MultiTitan CPU.” Technical Report TN-8, DEC-wrl, December 1988.
9. T. M. Austin and G. S. Sohi. “Zero-cycle loads: Microarchitecture support for reducing load latency.” In *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 82–92, December 1995.
10. M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison, 1993.
11. A. S. Huang, G. Slavenburg, and J. P. Shen. “Speculative disambiguation: A compilation technique for dynamic memory disambiguation.” In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 200–210, Chicago, IL, April 1994.
12. D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. -M. Hwu. “Dynamic memory disambiguation using the memory conflict buffer.” In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, San Jose, California, October 4–7, 1994.
13. D. Levitan, T. Thomas, and P. Tu. “The PowerPC 620 microprocessor: A high performance superscalar RISC processor.” *COMPCON 95*, 1995.
14. S. P. Harbison. *A Computer Architecture for the Dynamic Optimization of High-Level Language Programs*. PhD thesis, Carnegie Mellon University, September 1980.
15. S. P. Harbison. “An architectural alternative to optimizing compilers.” In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 57–65,

March 1982.

16. S. E. Richardson. “Caching function results: Faster arithmetic by avoiding unnecessary computation.” Technical report, Sun Microsystems Laboratories, 1992.
17. M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. “Value locality and load value prediction.” In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, October 1996.
18. M. H. Lipasti and J. P. Shen. “Exceeding the dataflow limit via value prediction.” In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1996.
19. J. E. Smith. “A study of branch prediction techniques.” In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 135–147, June 1981.
20. W. Y. Chen, S. A. Mahlke, P. P. Chang, and W.-M. Hwu. “Data access microarchitecture for superscalar processors with compiler-assisted data prefetching.” In *Proceedings of the 24th International Symposium on Microarchitecture*, 1991.
21. T.-F. Chen and J.-L. Baer. “A performance study of software and hardware data prefetching schemes.” In *21st Annual International Symposium on Computer Architecture*, pages 223–232, 1994.
22. M. H. Lipasti, W. J. Schmidt, R. R. Roediger, and S. R. Kunkel. “SPAID: Software prefetching in pointer- and call-intensive environments.” In *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, 1995.
23. SIGPLAN. *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26, Cambridge, MA, September 1991. SIGPLAN Notices.
24. A. Srivastava and D. W. Wall. “Link-time optimization of address calculation on a 64-bit architecture.” *SIGPLAN Notices*, 29(6):49–60, June 1994. Proceedings of the ACM SIGPLAN ’94 Conference on Programming Language Design and Implementation.
25. D. Keppel, S. J. Eggers, and R. R. Henry. “Evaluating runtime-compiled, value-specific optimizations.” Technical report, University of Washington, 1993.
26. S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. “Predictability of load/store instruction latencies.” In *Proceedings of the 26th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1993.
27. T. A. Diep, C. Nelson, and J. P. Shen. “Performance evaluation of the PowerPC 620 microarchitecture.” In *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.
28. T. A. Diep and J. P. Shen. “VMW: A visualization-based microarchitecture workbench.” *IEEE Computer*, 28(12):57–64, 1995.
29. F. Gabbay and A. Mendelson. “The effect of instruction fetch bandwidth on value prediction.” In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, Barcelona, Spain, June 1998.
30. M. H. Lipasti. *Value Locality and Speculative Execution*. PhD thesis, Carnegie Mellon University, May 1997.

Table I: Benchmark Descriptions

Bench-mark	Description	Input Set	Instr. Count
cc1-271	GCC 2.7.1	SPEC95 genoutput.i	102M
cc1	SPEC92 GCC 1.35	SPEC92 insn-recog.i	146M
cjpeg	JPEG encoder	128x128 BW image	2.8M
com-press	SPEC92 compression	1 iter. w/ 1/2 input	38.8M
eqntott	SPEC92 eqn to tr tbl	SPEC92 mod. input	25.5M
gawk	GNU awk	Parse 1.7M output	25.0M
gperf	GNU hash fn gen	-a -k 1-13 -D -o dict	7.8M
grep	GNU grep -c “st*mo”	Same as compress	2.3M
mpeg	MPEG decoder	4 frames	8.8M
perl	SPEC95 anagram srch	“admits” in 1/8 input	105M
quick	Recursive quick sort	5,000 elements	688K
sc	SPEC92 spreadsheet	SPEC92 short input	78.5M
xlisp	SPEC92 LISP	6 queens	52.1M
doduc	SPEC92 Nucl sim	SPEC92 tiny input	35.8M
hydro2d	SPEC92 galactic jets	SPEC92 short input	4.3M
swm256	SPEC92 water model	5 iterations	43.7M
tom-catv	SPEC92 mesh gen	4 iterations (vs. 100)	30.0M
Total			720M

.

Table II: Instruction Types

Instr Type	Description	Freq (%)
SC_A	Single-cycle arithmetic, 2 reg. operands	5.45
SC_A_I	Single-cycle arithmetic, 1 reg. operand	23.55
SC_L	Single-cycle logical, 2 reg. operands	1.86
SC_L_I	Single-cycle logical, 1 reg. operand	9.89
MC_A	Multi-cycle arithmetic, 2 reg. operands	0.14
MC_A_I	Multi-cycle arithmetic, 1 reg. operand	0.06
MC_MV	Multi-cycle register move	1.86
I_LD	Integer load instructions	33.00
ST_U	Store with base reg. update	5.14
FP_LD	FP load single	3.16
FPD_LD	FP load double	4.76
FP_A	FP instructions other than multiply	3.52
FP_M	FP multiply instructions	2.11
FP_MA	FP multiply-add instructions	3.65
FP_OTH	FP div,abs,neg,round to single precision	1.61
FP_MV	FP register move instructions	0.26

Table III: Classification Table State Descriptions

Size (entries/ bits)	State Descriptions
256/1-bit	{0=no pred,1=pred & no repl & const}
256/2-bit	{0,1=no pred,2-3=pred, 3=no repl & const}
256/3-bit	{0,1=no pred,2-7=pred,5-7=no repl,7=const}
1024/1-bit	{0=no pred,1=pred & no repl & const}
1024/2-bit	{0,1=no pred, 2-3=pred, 3=no repl & const}
1024/3-bit	{0,1=no pred,2-7=pred,5-7= no repl,7=const}

.

Table IV: VP Unit Configurations.
For history depth greater than one, a hypothetical perfect selection mechanism is assumed.

VP Unit Configuration	VP Table		CT		CVU
	Entries	History Depth	Entries	Bits/ Entry	Bits/ Entry
SimpleLVP	1024	1	256	2	32
ConstantLVP	1024	1	256	1	128
LimitLVP	4096	16/Perf	1024	2	128
PerfectLVP	∞	∞/Perf	∞	Perfect	0
SimpleVP	4096	1	1024	2	N/A
1PerfCTVP	4096	1	∞	Perfect	N/A
4PerfCTVP	4096	4/Perfect	∞	Perfect	N/A
8PerfCTVP	4096	8/Perfect	∞	Perfect	N/A
PerfectVP	∞	Perfect	∞	Perfect	N/A

Table V: CT Hit Rates for LVP.

Percentages shown are fractions of unpredictable and predictable loads identified as such by the CT.

Benchmark	Load Predictability			
	SimpleLVP		LimitLVP	
	Unpr	Pred	Unpr	Pred
cc1-271	86%	64%	58%	90%
cjpeg	97%	61%	92%	61%
compress	99%	94%	97%	90%
doduc	83%	75%	82%	92%
eqntott	91%	85%	88%	99%
gawk	85%	92%	44%	95%
gperf	93%	75%	76%	97%
grep	93%	88%	67%	81%
hydro2d	82%	85%	63%	91%
mpeg	86%	90%	78%	93%
perl	84%	71%	65%	93%
quick	98%	84%	93%	89%
sc	77%	90%	59%	97%
swm256	99%	89%	99%	93%
tomcatv	100%	89%	100%	98%
xlisp	88%	83%	77%	93%
GM	90%	81%	75%	90%

.

Table VI: Constant Verification Rates.
Shown are ratios of constant loads to all dynamic loads.

Benchmark	Constant Loads	
	SimpleLVP	LimitLVP
cc1-271	13%	23%
cjpeg	4%	7%
compress	33%	34%
doduc	5%	20%
eqntott	19%	44%
gawk	10%	28%
gperf	21%	39%
grep	16%	24%
hydro2d	2%	8%
mpeg	12%	25%
perl	8%	19%
quick	0%	0%
sc	32%	46%
swm256	8%	17%
tomcatv	0%	0%
xlisp	14%	45%
GM	6%	11%

Table VII: Machine Model Specifications

Instruction Class	PowerPC 620/620+				Infinite
	# FU/RS		Issue Lat	Result Lat	I&R Lat
	620	620+			
Simple Int	2/4	2/8	1	1	1,1
Complex Int	1/2	1/4	1-35	1-35	1,1
Load/Store	1/3	2/6	1	2	1,1
Simple FP	1/2	1/4	1	3	1,1
Complex FP	shared	shared	18	18	11,
Br (pr/mispr)	1/4	1/8	1	0/1+	1,0/1+

.

Table VIII: Baseline Performance (IPC)

Bench mark	620	620+	Infinite
cc1-271	1.05540	1.07260	6.40244
cc1	1.20880	1.30892	6.81969
cjpeg	0.99308	1.10503	10.11820
compress	1.15508	1.22739	5.66520
eqntott	1.35984	1.41655	5.58084
gawk	1.22254	1.23106	4.05087
gperf	1.61187	1.82027	7.00588
grep	1.07909	1.06635	2.02673
mpeg	1.62410	1.86998	7.99286
perl	1.00018	1.05241	8.03310
quick	0.97000	0.99904	4.91123
sc	1.24365	1.31691	6.75335
xlisp	1.15722	1.21509	8.30155
doduc	0.81249	0.83851	5.80629
hydro2d	0.80267	0.82059	5.53410
swm256	0.85172	0.88852	4.15299
tomcatv	0.91337	0.93081	5.77235
GM	1.09757	1.15473	5.84794

Figure Captions

Figure 1 **Taxonomy of Speculative Execution Techniques.**

Figure 2 **Load Value Locality.**The light bars show load value locality for a history depth of one, while the dark bars show it for a history depth of sixteen.

Figure 3 **Load Value Locality by Data Type.**The light bars show value locality for a history depth of one, while the dark bars show it for a history depth of sixteen.

Figure 4 **Register Value Locality.** The light bars show value locality for a history depth of one, and dark bars show it for a history depth of four.

Figure 5 **Register Value Locality by Instruction Type.**

Figure 6 **Value Prediction Unit.** The PC of the instruction being predicted is used to index into the VPT to find a value to predict. At the same time, the CT is also indexed with the PC to determine whether or not a prediction should be made. When the instruction completes, both the prediction history and value history are updated.

Figure 7 **VPT Hit Rate Sensitivity to Size**

Figure 8 **CT Hit Rates.**The Predictable Hit Rate is the number of correct value predictions that were identified as such by the CT divided by the total number of correct predictions, while the Unpredictable Hit Rate is the number of incorrect predictions that were identified as such by the CT divided by the number of incorrect predictions.

Figure 9 **Example use of Value Prediction Mechanism.** The dependent instruction shown on the right uses the predicted result of the instruction on the left, and is able to issue and execute in the same cycle.

Figure 10 **Block Diagram of the LVP Mechanism.** The Load PC is used to index into the VPT and CT to find a value to predict and to determine whether or not a prediction should be made. Constant loads that find a match in the CVU needn't access the cache, while stores cancel all matching CVU entries. When the load completes, the predicted and actual values are compared, the VPT and CT are updated, and dependent instructions are reissued if necessary.

Figure 11 **PPC 620 and 620+ Block Diagram.** Buffer sizes are shown as (620/620+).

Figure 12 **PowerPC 620 Speedup with Load Value Prediction.**

Figure 13 **PowerPC 620 Speedup with Register Value Prediction**

Figure 14 **PowerPC 620+ Speedup with Register Value Prediction**

Figure 15 **Infinite Machine Model Speedup with Register Value Prediction**

Figure 16 **Doubling PowerPC 620/620+ Data Cache vs. Value Prediction**

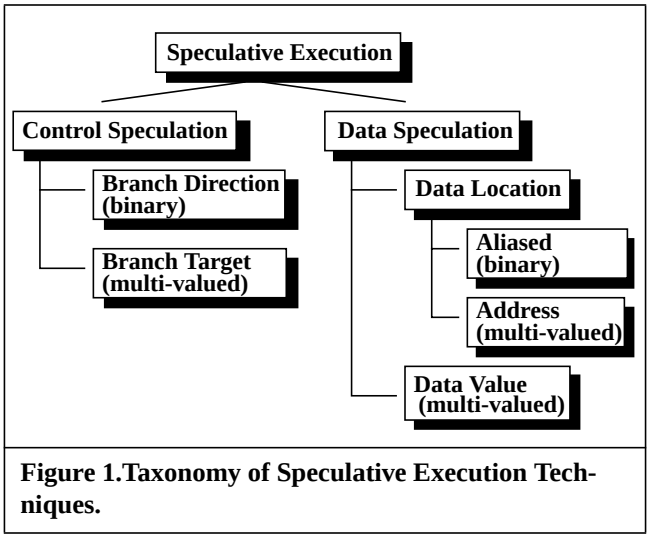

Figure 1.Taxonomy of Speculative Execution Techniques.

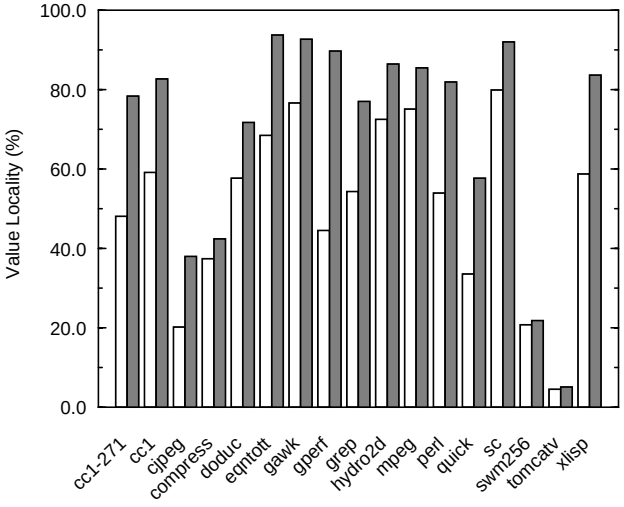

Figure 2.Load Value Locality. The light bars show load value locality for a history depth of one, while the dark bars show it for a history depth of sixteen.

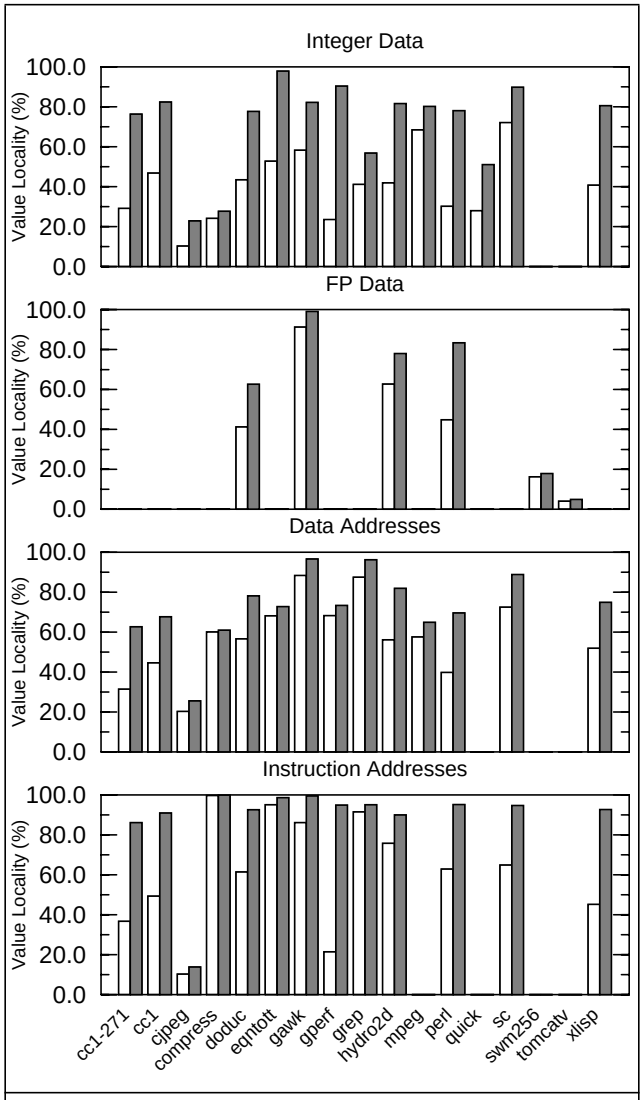

Figure 3.Load Value Locality by Data Type. The light bars show value locality for a history depth of one, while the dark bars show it for a history depth of sixteen.

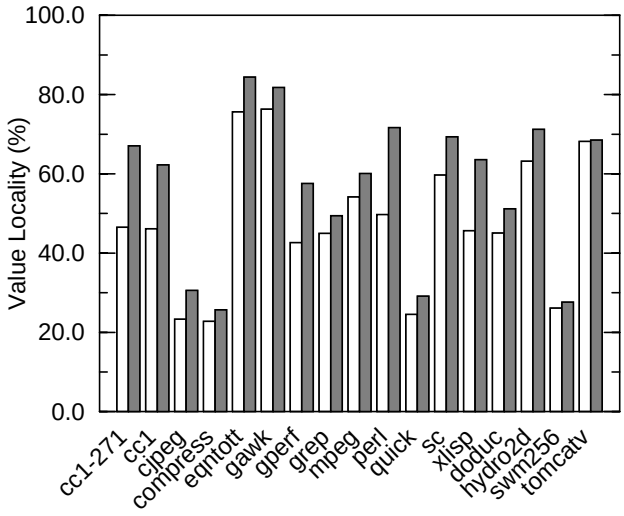

Figure 4.Register Value Locality. The light bars show value locality for a history depth of one, and dark bars show it for a history depth of four.

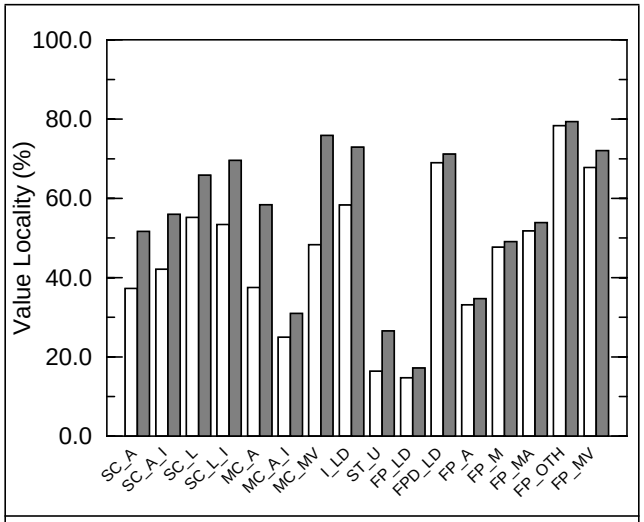

Figure 5.Register Value Locality by Instruction Type.

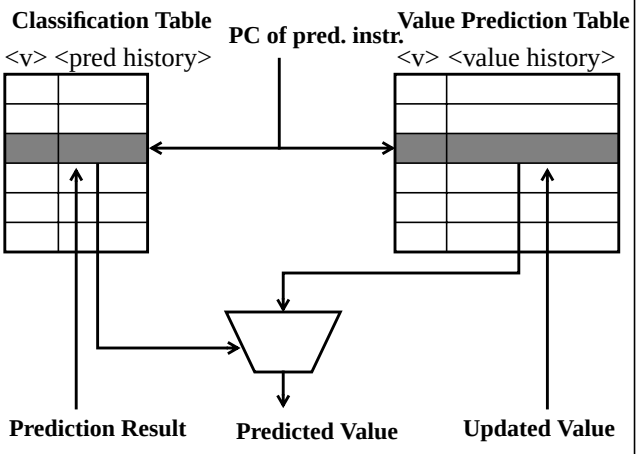

Figure 6.Value Prediction Unit. The PC of the instruction being predicted is used to index into the VPT to find a value to predict. At the same time, the CT is also indexed with the PC to determine whether or not a prediction should be made. When the instruction completes, both the prediction history and value history are updated.

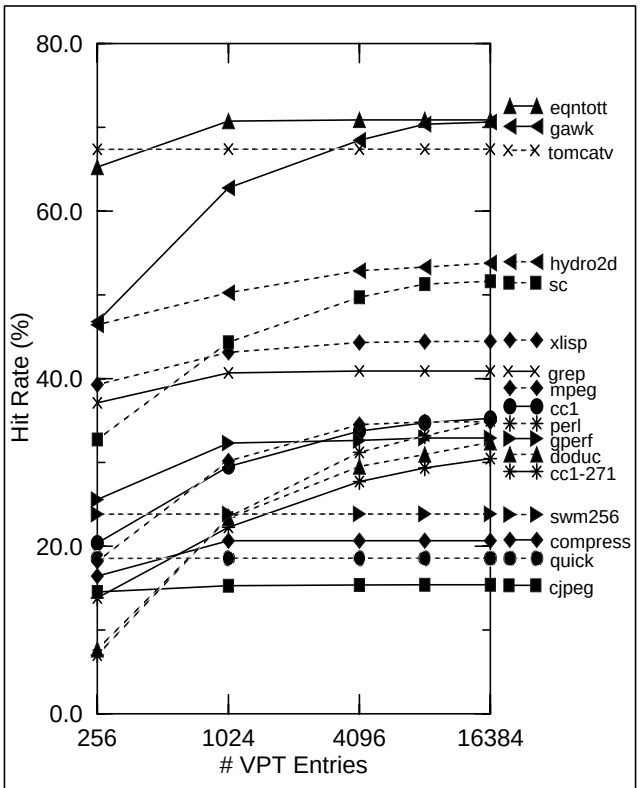

Figure 7.VPT Hit Rate Sensitivity to Size

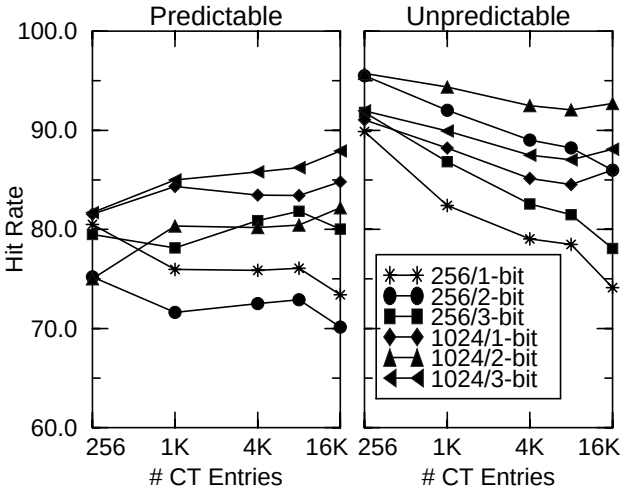

Figure 8.CT Hit Rates. The Predictable Hit Rate is the number of correct value predictions that were identified as such by the CT divided by the total number of correct predictions, while the Unpredictable Hit Rate is the number of incorrect predictions that were identified as such by the CT divided by the number of incorrect predictions.

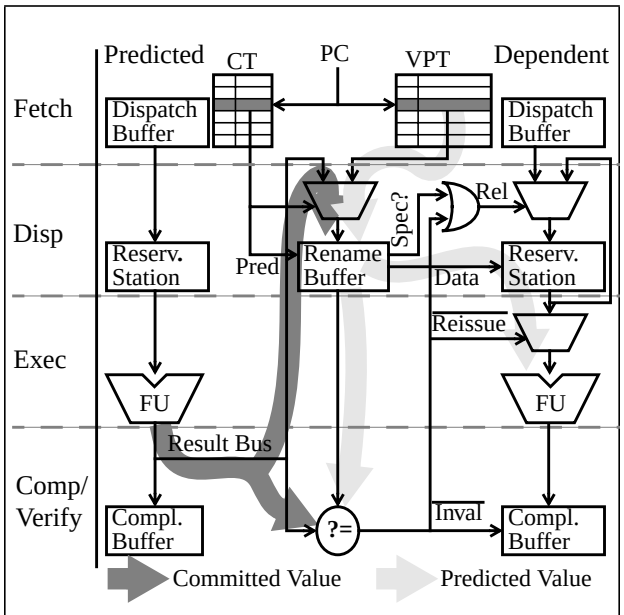

Figure 9.Example use of Value Prediction Mechanism. The dependent instruction shown on the right uses the predicted result of the instruction on the left, and is able to issue and execute in the same cycle.

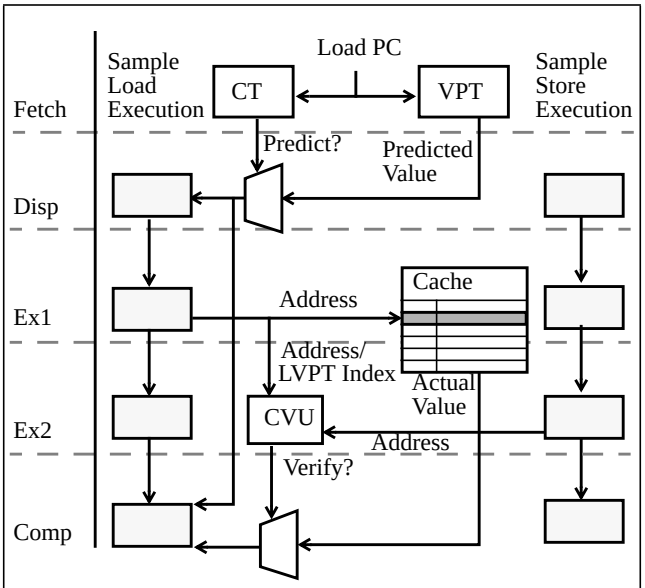

Figure 10.Block Diagram of the LVP Mechanism. The Load PC is used to index into the VPT and CT to find a value to predict and to determine whether or not a prediction should be made. Constant loads that find a match in the CVU needn't access the cache, while stores cancel all matching CVU entries. When the load completes, the predicted and actual values are compared, the VPT and CT are updated, and dependent instructions are reissued if necessary.

.

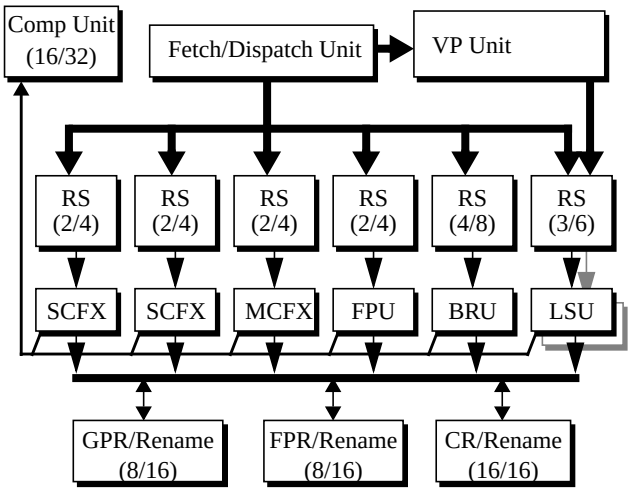

Figure 11.PPC 620 and 620+ Block Diagram. Buffer sizes are shown as (620/620+).

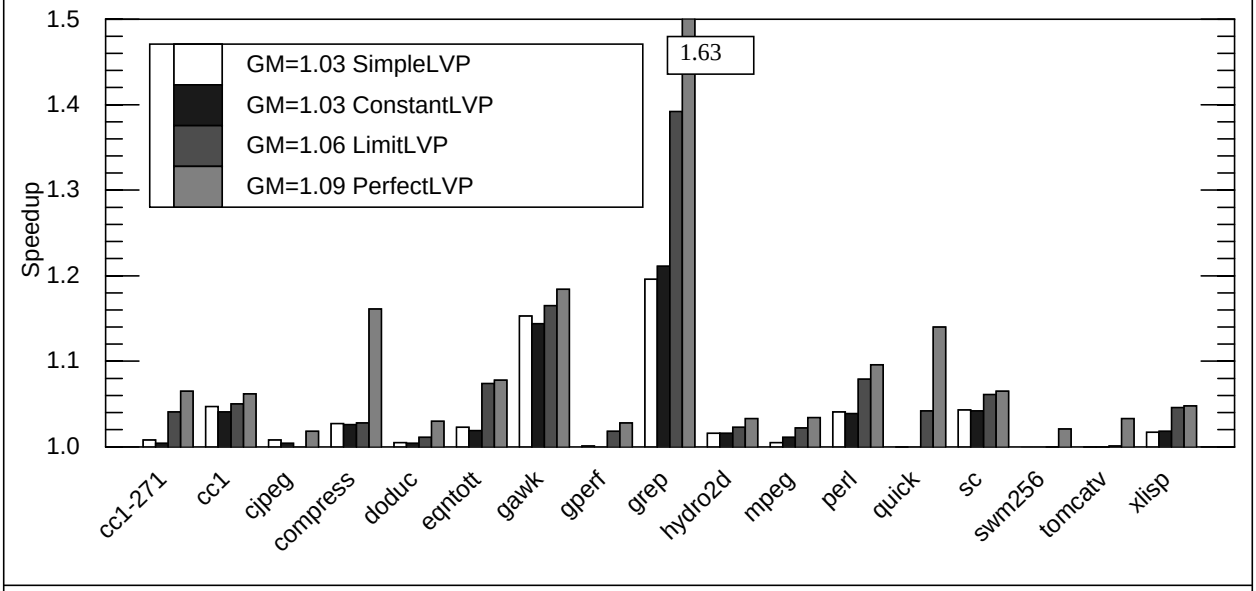

Figure 12.PowerPC 620 Speedup with Load Value Prediction.

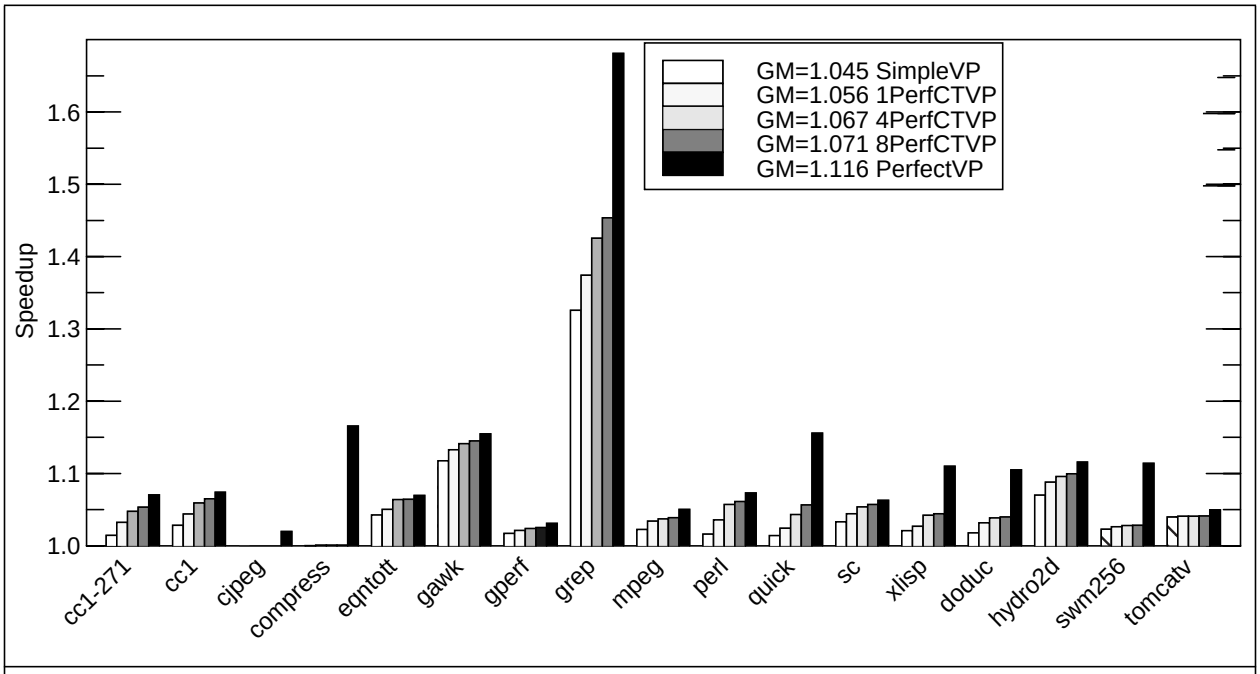

Figure 13.PowerPC 620 Speedup with Register Value Prediction

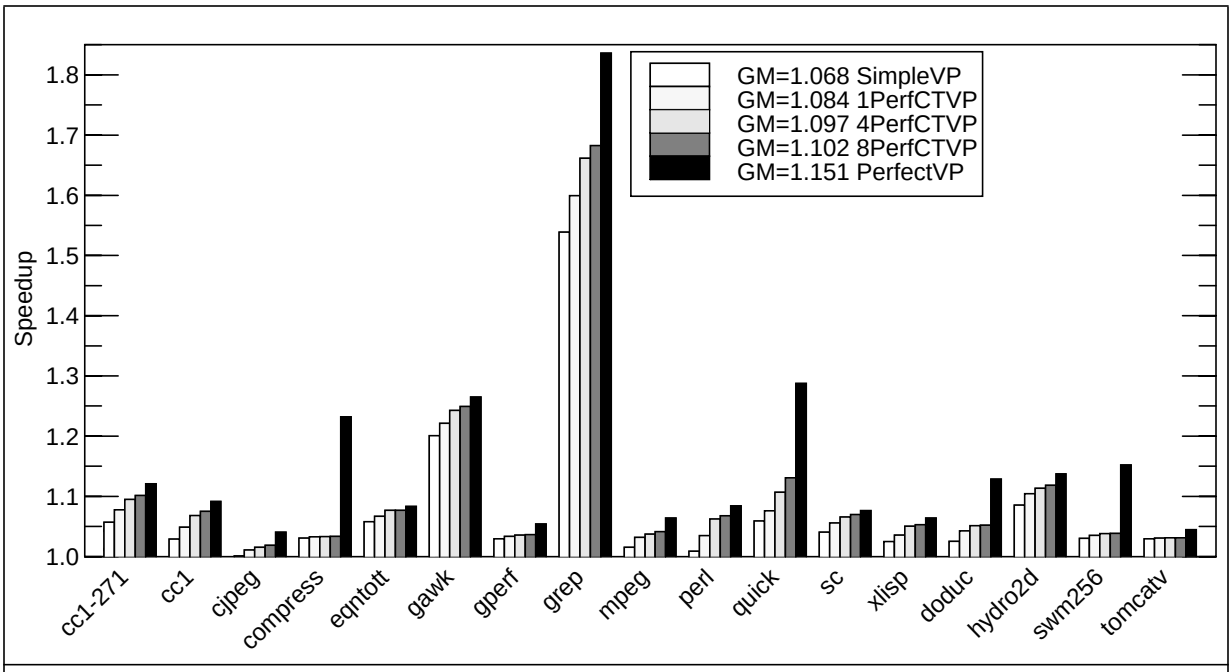

Figure 14.PowerPC 620+ Speedup with Register Value Prediction

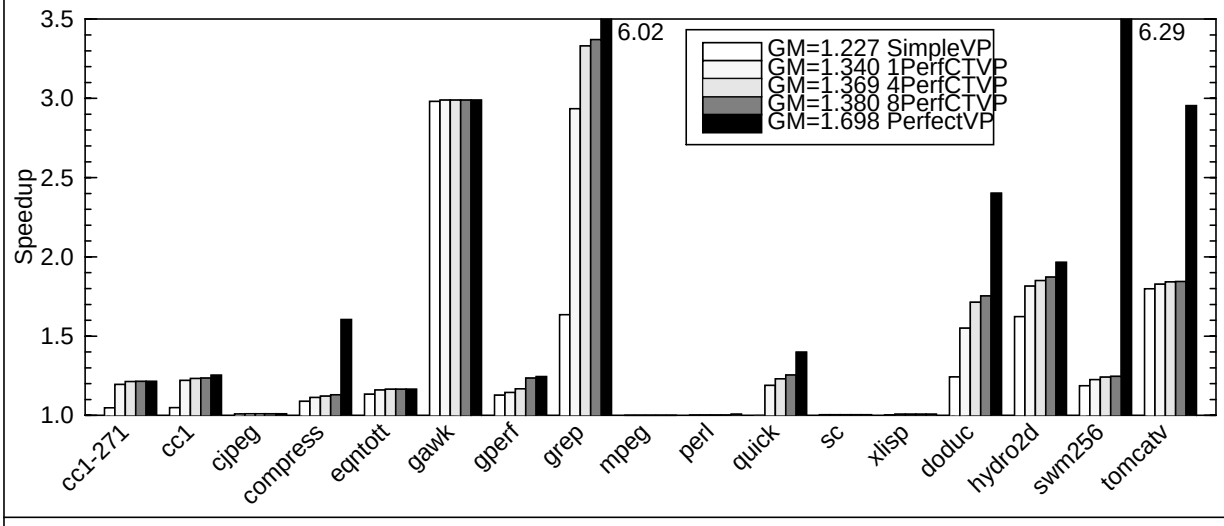

Figure 15.Infinite Machine Model Speedup with Register Value Prediction

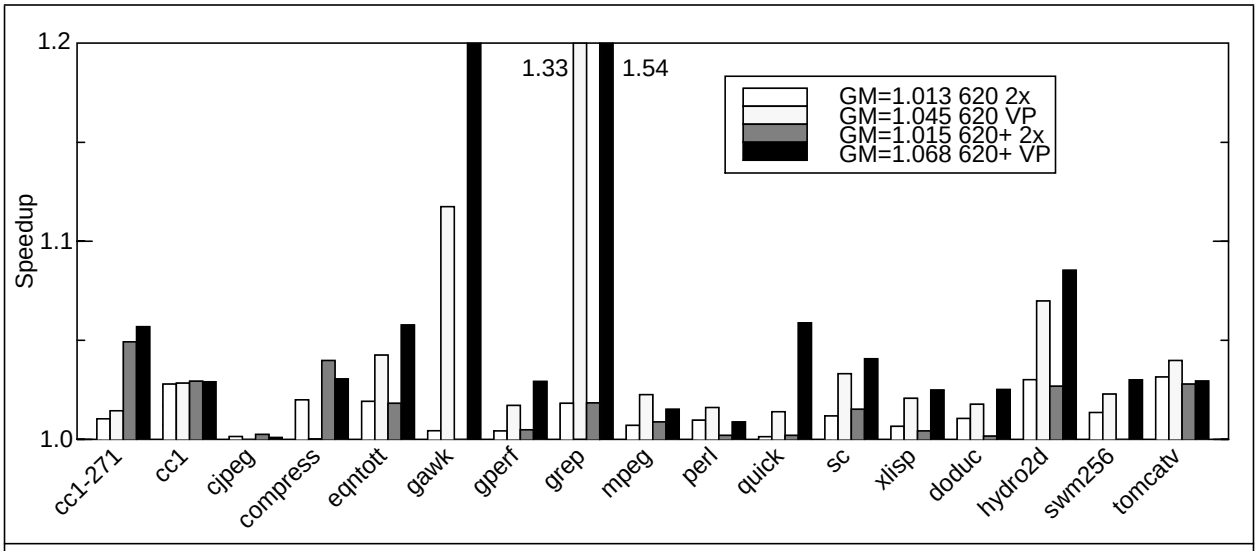

Figure 16.Doubling PowerPC 620/620+ Data Cache vs. Value Prediction