Value Locality and Load Value Prediction

Mikko H. Lipasti, Christopher B. Wilkerson[1], and John Paul Shen
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh PA, 15213
{mhl,shen}@ece.cmu.edu

Abstract

Since the introduction of virtual memory demand-paging and cache memories, computer systems have been exploiting spatial and temporal locality to reduce the average latency of a memory reference. In this paper, we introduce the notion of *value locality*, a third facet of locality that is frequently present in real-world programs, and describe how to effectively capture and exploit it in order to perform *load value prediction*. Temporal and spatial locality are attributes of storage locations, and describe the future likelihood of references to those locations or their close neighbors. In a similar vein, *value locality* describes the likelihood of the recurrence of a previously-seen value within a storage location. Modern processors already exploit *value locality* in a very restricted sense through the use of control speculation (i.e. branch prediction), which seeks to predict the future value of a single condition bit based on previously-seen values. Our work extends this to predict entire 32- and 64-bit register values based on previously-seen values. We find that, just as condition bits are fairly predictable on a per-static-branch basis, full register values being loaded from memory are frequently predictable as well. Furthermore, we show that simple microarchitectural enhancements to two modern microprocessor implementations (based on the PowerPC 620 and Alpha 21164) that enable *load value prediction* can effectively exploit *value locality* to collapse true dependencies, reduce average memory latency and bandwidth requirements, and provide measurable performance gains.

1. Introduction and Related Work

The gap between main memory and processor clock speeds is growing at an alarming rate [RD94]. As a result, computer system performance is increasingly dominated by the latency of servicing memory accesses, particularly those accesses which are not easily predicted by the temporal and spatial locality captured by conventional cache memory organizations [Smi82]. Conventional cache memories rely on a program's *temporal* and *spatial locality* to reduce the average memory access latency. *Temporal locality* describes the likelihood that a recently-referenced address will be referenced again soon, while *spatial locality* describes the likelihood that a close neighbor of a recently-referenced address will be referenced soon. Designing the physical attributes (e.g. size, line size, associativity, etc.) of a cache memory to best match the temporal and spatial locality of programs has been an ongoing challenge for researchers and designers alike. Some have proposed adding additional features such as non-blocking fetches [Kro81], victim caches [Jou90], and sophisticated hardware prefetching [CB94] to alleviate the access penalties for those references that have locality characteristics that are not captured by more conventional designs.

Others have proposed altering the behavior of programs to improve the data locality of programs so that it better matches the capabilities of the cache hardware. Such improvements have primarily been limited to scientific code with predictable control flow and regular memory access patterns, due to the ease with which rudimentary loop transformations can dramatically improve temporal and spatial locality [ASKL81,CMT94]. Explicit prefetching in advance of memory references with poor or no locality has also been examined extensively in this context, both with [CMCH91,CB94] and without additional hardware support [CKP91,MLG92]. Dynamic hardware techniques for controlling cache memory allocation that significantly reduce memory bandwidth requirements have also been proposed [TFMP95]. In addition, alternative pipeline configurations that reduce average memory access latency via early execution of loads have been examined [Jou88,AS95].

The most relevant prior work related to ours is the *Tree Machine* [Har80,Har82], 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* [ASU86] in hardware). Richardson follows up on this concept in [Ric92] 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 this paper, we introduce *value locality*, a concept related to *redundant computation*, and demonstrate 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.

2. Value Locality

In this paper, we introduce the concept of *value locality*, which we define 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

1. Currently with Intel Corporation in Portland, Oregon.

Preprint of paper to appear in
ASPLOS-VII, October 1996

have limited our current study to examine only the value locality of general-purpose or floating-point registers immediately following memory loads that target those registers. A plethora of previous work on dynamic branch prediction (e.g. [Smi81,YP91]) has focused on an even more restricted application of value locality, namely the prediction of a single condition bit based on its past behavior. This paper can be viewed as a logical continuation of that body of work, extending the prediction of a single bit to the prediction of an entire 32- or 64-bit register.

Intuitively, it seems that it would be a very difficult task to discover any useful amount of value locality in a 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 load individually, the task becomes much easier, and we are able to accurately predict a significant fraction of register values being loaded from memory.

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* [SIG91] 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.
- **Error-checking**: Checks for infrequently-occurring conditions often compile into loads of what 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. [SW94, KEH93]). 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 two modern RISC ISAs that we examined, both of which provide state-of-the-art compilers and run-time systems. It is worth pointing out, however, that the value locality of particular static loads 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 [ASW+93].

TABLE 1. **Benchmark Descriptions**

Benchmarks	Description	Input Set	Instr. Count	
			PPC	Alpha
cc1-271	GCC 2.7.1; SPEC95 flags	genoutput.i from SPEC95	102M	117M
cc1	GCC 1.35 from SPEC92	insn-recog.i from SPEC92	146M	N/A
cjpeg	JPEG encoder	128x128 BW image	2.8M	10.7M
compress	SPEC92 file compression	1 iter. with 1/2 of SPEC92	38.8M	50.2M
eqntott	SPEC92 Eqn to truth table	Mod. input from SPEC92	25.5M	44.0M
gawk	GNU awk; result parser	1.7M simulator output file	25.0M	53.0M
gperf	GNU hash fn generator	gperf -a -k 1-13 -D -o dict	7.8M	10.8M
grep	gnu-grep -c "st*mo"	Same as compress	2.3M	2.9M
mpeg	Berkeley MPEG decoder	4 frames w/ fast dithering	8.8M	15.1M
perl	SPEC95 Anagram search	find "admits" in 1/8 of input	105M	114M
quick	Quick sort	5,000 random elements.	688K	1.1M
sc	Spreadsheet from SPEC92	Short input from SPEC92	78.5M	107M
xlisp	SPEC92 LISP interpreter	6 queens	52.1M	60.0M
doduc	Nuclear reactor simulator	Tiny input from SPEC92	35.8M	38.5M
hydro2d	Computation of galactic jets	Short input from SPEC92	4.3M	5.3M
swm256	Shallow water model	5 iterations (vs. 1,200)	43.7M	54.8M
tomcatv	Mesh generation program	4 iterations (vs. 100)	30.0M	36.9M
Total			720M	721M

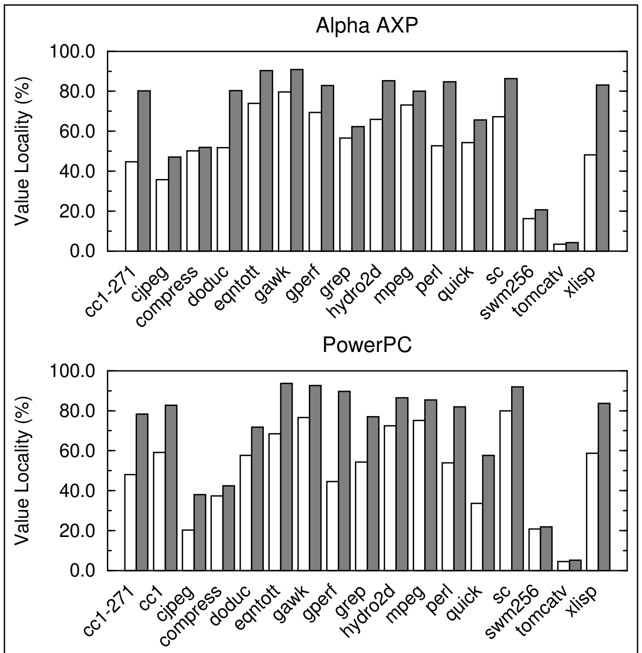

FIGURE 1. **Load Value Locality**.The light bars show value locality for a history depth of one, while the dark bars show it for a history depth of sixteen.

The benchmark set we use to explore value locality and quantify its performance impact is summarized in Table 1 . We have chosen thirteen integer benchmarks, five of them 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 have chosen four of the SPEC '92 floating-point benchmarks. All benchmarks are compiled at full optimization with each manufacturer's reference compilers, with the exception of gperf (our token C++ benchmark), which is compiled with IBM's CSET compiler under AIX, and GNU's g++ under OSF/1. All benchmarks are run to completion with the input sets described, but do not include supervisor-state instructions, which our tracing tools are unable to capture.

Figure 1 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 the course of a program's execution. Unfortunately, three of our benchmarks (cjpeg, swm256, and tomcatv) demonstrate poor load value locality.

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.

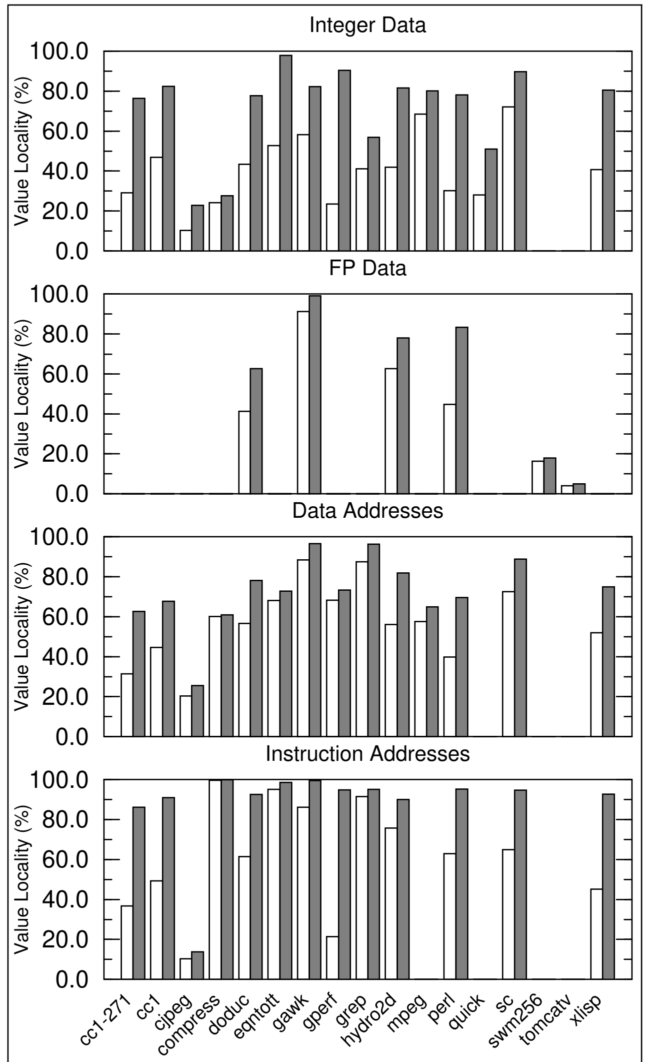

FIGURE 2. **PowerPC Value Locality by Data Type**.The light bars show value locality for a history depth of one, while the dark bars show it for a history depth of sixteen.

To further explore the notion of value locality, we collected data that classifies loads based on the type of data being loaded: floating-point data, non-floating-point data, instruction addresses, and data addresses (pointers). These results are summarized in Figure 2 (the results shown are for the PowerPC architecture only). 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. Exploiting Value Locality

The fact that memory loads in many programs demonstrate a significant degree of value locality opens up exciting new possibilities for the microarchitect. In this paper, we describe and evaluate the *Load Value Prediction Unit*, a hardware mechanism which addresses both the memory latency and memory bandwidth problems in a novel fashion. First, by exploiting the affinity between load instruction addresses and the values being loaded, we are able to reduce load latency by two or more cycles. Second, we can reduce memory bandwidth requirements by identifying highly-predictable loads and completely bypassing the conventional memory hierarchy

TABLE 2. **LVP Unit Configurations.** For history depth greater than one, a hypothetical perfect selection mechanism is assumed.

LVP Unit Configuration	LVP Table		LCT		CVU
	Entries	**History Depth**	**Entries**	**Bits/ Entry**	**Bits/ Entry**
Simple	1024	1	256	2	32
Constant	1024	1	256	1	128
Limit	4096	16/Perf	1024	2	128
Perfect	∞	∞/Perf	∞	Perfect	0

for these loads. The LVP Unit consists of a load value prediction table or LVPT (Section 3.1) for generating value predictions, a load classification table or LCT (Section 3.2. and Section 3.3) for deciding which predictions are likely to be correct, and a constant verification unit or CVU (Section 3.3) that replaces accessing the conventional memory hierarchy for verifying highly-predictable loads.

3.1. Load Value Prediction Table

The LVPT is used to predict the value being loaded from memory by associating the load instruction with the value previously loaded by that instruction. The LVPT is indexed by the load instruction address and is not tagged, so both constructive and destructive interference can occur between loads that map to the same entry (the LVPT is direct-mapped). Table 2 shows the number of entries (column 2) as well as the history depth per entry (column 3) for the four LVPT configurations used in our study. Configurations with a history depth greater than one assume a hypothetical perfect mechanism for selecting the correct value to predict, and are included to explore the limits of history-based load value prediction.

3.2. Dynamic Load Classification.

Load value prediction is useful only if it can be done accurately, since incorrect predictions can lead to increased structural hazards and longer load latency (the misprediction penalty is discussed further in Section 4). In our experimental framework we classify static loads into three categories based on their dynamic behavior. There are loads whose values are unpredictable with the LVPT, those that are predictable, and those that are almost always predictable. By classifying these separately we are able to take full advantage of each case. We can avoid the cost of a misprediction by identifying the unpredictable loads, and we can avoid the cost of a memory access if we can identify and verify loads that are highly-predictable.

In order to determine the predictability of a static load instruction, it is associated with a set of history bits. Based on whether or not previous predictions for a given load instruction were correct, we are able to classify the loads into three general groups: *unpredictable*, *predictable*, and *constant* loads. The load classification table or LCT consists of a direct-mapped table of n-bit saturating counters indexed by the low-order bits of the instruction address. Table 2 shows the number of entries (column 4) as well as the size of each saturating counter (column 5) for the LCT configurations used in our study. The 2-bit saturating counter assigns the four available states 0-3 as *"don't predict"*, *"don't predict"*, *"predict"* and *"constant,"* while the 1-bit counter assigns the two states as "*don't predict*" and "*constant*." The counter is incremented when the predicted value is correct and decremented otherwise. In Table 3, we show the percentage of all unpredictable loads the LCT is able to classify as unpredictable (columns 2, 4, 6, and 8) and the percentage of predictable loads the LCT is able to correctly classify as predictable (columns 3, 5, 7, and 9) for the *Simple* and *Limit* configurations.

TABLE 3. **LCT Hit Rates.** Percentages shown are fractions of unpredictable and predictable loads identified as such by the LCT.

Benchmark	PowerPC				Alpha AXP			
	Simple		Limit		Simple		Limit	
	Unpr	Pred	Unpr	Pred	Unpr	Pred	Unpr	Pred
cc1-271	86%	64%	58%	90%	86%	57%	64%	86%
cjpeg	97%	61%	92%	61%	93%	75%	93%	82%
compress	99%	94%	97%	90%	98%	56%	97%	94%
doduc	83%	75%	82%	92%	84%	68%	78%	92%
eqntott	91%	85%	88%	99%	68%	80%	83%	97%
gawk	85%	92%	44%	95%	74%	86%	59%	93%
gperf	93%	75%	76%	97%	77%	79%	77%	91%
grep	93%	88%	67%	81%	85%	82%	92%	92%
hydro2d	82%	85%	63%	91%	86%	80%	60%	89%
mpeg	86%	90%	78%	93%	84%	88%	85%	93%
perl	84%	71%	65%	93%	83%	66%	74%	93%
quick	98%	84%	93%	89%	98%	95%	96%	95%
sc	77%	90%	59%	97%	86%	85%	78%	95%
swm256	99%	89%	99%	93%	99%	86%	99%	90%
tomcatv	100%	89%	100%	98%	99%	68%	99%	70%
xlisp	88%	83%	77%	93%	90%	74%	76%	93%
GM	90%	81%	75%	90%	86%	78%	81%	90%

3.3. Constant Verification Unit

Although the LCT mechanism can accurately identify loads that retrieve predictable values, we still have to verify the correctness of the LVPT's predictions. For *predictable* loads, we simply retrieve the value from the conventional memory hierarchy and compare the predicted value to the actual value (see Figure 3). However, for highly-predictable or *constant* loads, we use the constant verification unit, or CVU, which allows us to avoid accessing the conventional memory system completely by forcing the LVPT entries that correspond to constant loads to remain coherent with main memory.

TABLE 4. **Successful Constant Identification Rates.** Percentages shown are ratio of constant loads to all dynamic loads.

Benchmark	PowerPC		Alpha AXP	
	Simple	Limit	Simple	Limit
cc1-271	13%	23%	10%	14%
cjpeg	4%	7%	17%	17%
compress	33%	34%	36%	42%
doduc	5%	20%	5%	15%
eqntott	19%	44%	21%	35%
gawk	10%	28%	31%	31%
gperf	21%	39%	38%	56%
grep	16%	24%	18%	22%
hydro2d	2%	8%	3%	10%
mpeg	12%	25%	10%	28%
perl	8%	19%	7%	8%
quick	0%	0%	31%	31%
sc	32%	46%	26%	31%
swm256	8%	17%	12%	12%
tomcatv	0%	0%	1%	1%
xlisp	14%	45%	8%	30%
GM	6%	11%	12%	18%

For the LVPT entries that are classified as constants by the LCT, the data address and the index of the LVPT are placed in a separate, fully-associative table inside the CVU. 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 LVPT 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 LVPT 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 4 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. As a second-order effect, we also observe a decrease in the L2 cache bandwidth for our 21164 machine model (see Section 6.1). 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.

3.4. The Load Value Prediction Unit

The interactions between the LVPT, LCT, and CVU are described in Figure 3 for both loads and stores. When a load instruction is fetched, the low-order bits of the load instruction address are used to index the LVPT and LCT in parallel. The LCT (analogous to a branch history table) determines whether or not a prediction should be made, and the LVPT (analogous to a branch target buffer) forwards the value to the load's dependent instructions. Once the address is generated, in stage EX1 of the sample pipeline, the cache access and CVU access progress in parallel. When the actual value returns from the L1 data cache, it is compared with the predicted data, and the dependent speculative instructions are consequently either written back or reissued. Since the search on the CVU can not be performed in time to prevent initiating the memory access, the only time the CVU is able to prevent the memory access is when a bank conflict or cache miss occurs. In either case, a CVU match will cancel the subsequent retry or cache miss. During the execution of a store, a fully-associative lookup is performed on the store's address and all matching entries are removed from the CVU.

3.5. LVP Unit Implementation Notes

An exhaustive investigation of LVP Unit design parameters and implementation details is beyond the scope of this paper. However, to demonstrate the validity of the concept, we analyzed sensitivity to a few key parameters, and then selected several design points to use with our microarchitectural studies (Section 6). We realize that the designs we have selected are by no means optimal, minimal, or very efficient, and could be improved significantly. For example, we reserve a full 64 bits per value entry in the LVP Table, 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. The intent of this paper is not to present the details of such a design; rather, our intent is to explore the larger issue of the impact of load value prediction on microarchitecture and instruction-level parallelism, and to leave such details to future work.

However, we note that the LVP Unit has several characteristics that make it attractive to a CPU designer. First of all, since the LVPT and LCT lookup index is 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). Finally, we reiterate that the LVP Unit, though speculative in nature, actually reduces memory bandwidth requirements, rather than aggravating them.

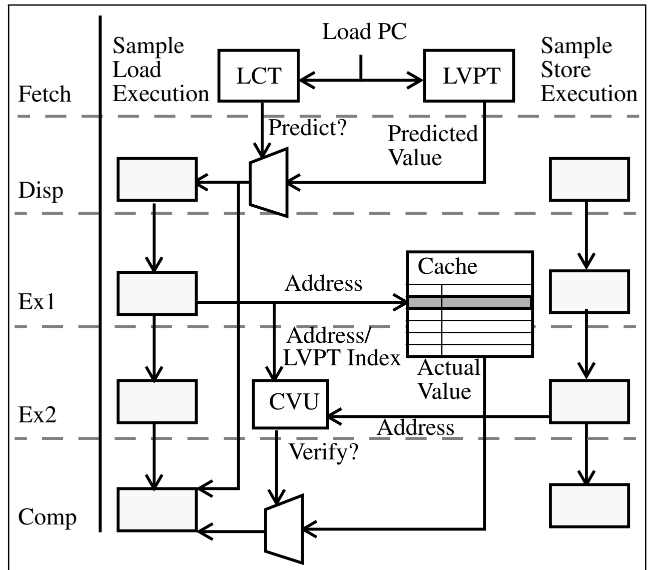

FIGURE 3. **Block Diagram of the LVP Mechanism**. The Load PC is used to index into the LVPT and LCT to find a value to predict and to determine whether or not a prediction should be made. Constant loads that find a match in the CVU needn't access the cache, while stores cancel all matching CVU entries. When the load completes, the predicted and actual values are compared, the LVPT and LCT are updated, and dependent instructions are reissued if necessary.

4. Microarchitectural Models

In order to validate and quantify the performance impact of load value prediction and constant identification, we implemented trace-driven timing models for two significantly different modern microprocessor implementations--the PowerPC 620 [DNS95, LTT95] and the Alpha AXP 21164 [BK95]; one aggressively out of order, the other clean and in order. We chose to use two different architectures in order to alleviate our concern that the value locality behavior we observed is perhaps only an artifact of certain idioms in the instruction set, compiler, run-time environment, and/or operating system we were running on, rather than a more universal attribute of general-purpose programs. We chose the PowerPC 620 and the AXP 21164 since they represent two extremes of the microarchitectural spectrum, from complex "brainiac" CPUs that aggressively and dynamically reorder instructions to achieve a high IPC metric, to the clean, straightforward, and deeply-pipelined "speed demon" CPUs

TABLE 5. **Instruction Latencies**

Instruction Class	PowerPC 620		Alpha AXP 21164	
	Issue	Result	Issue	Result
Simple Integer	1	1	1	1
Complex Integer	1-35	1-35	16	16
Load/Store	1	2	1	2
Simple FP	1	3	1	4
Complex FP	18	18	1	36-65
Branch(pred/mispr)	1	0/1+	1	0/4

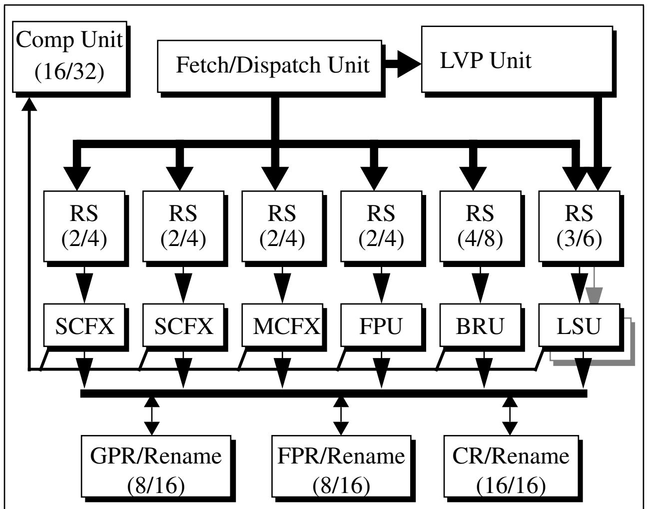

FIGURE 4. **PowerPC 620 and 620+ Block Diagram**. Buffer sizes are shown as (620/620+).

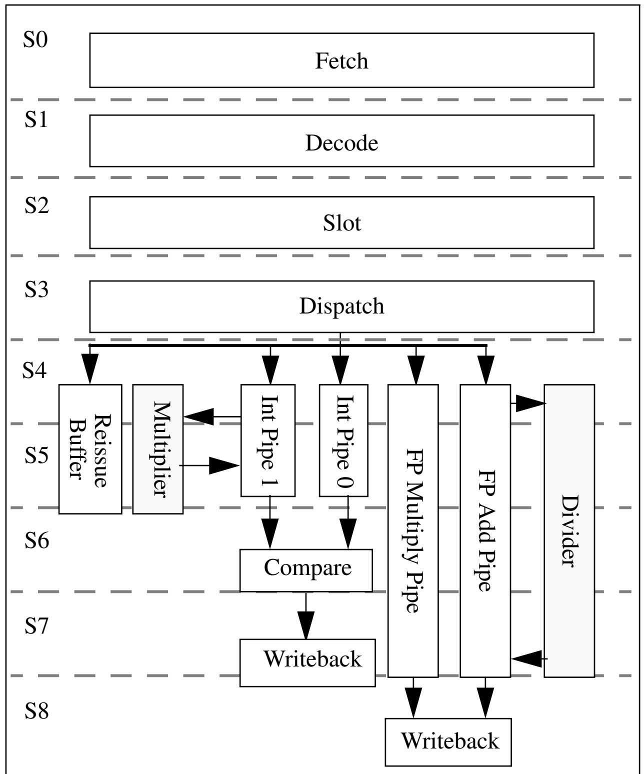

FIGURE 5. **Alpha AXP 21164 Block Diagram**.

that rely primarily on clock rate for high performance [Gwe94]. The issue and result latencies for common instruction types on both machines are summarized in Table 5 .

4.1. The PowerPC 620 Machine Model

The microarchitecture of the PowerPC 620 is summarized in Figure 4. Our model is based on published reports on the PowerPC 620 [DNS95, LTT95] and accurately models all 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 instruction refetching, and in-order completion. To alleviate some of the bottlenecks we found in the 620 design, we also model an aggressive "next-generation" version of the 620, which we term the 620+. The 620+ differs from the 620 by doubling the number of reservation stations, FPR and GPR rename buffers, 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 add a LVP Unit that predicts load values by keeping a value history indexed by load instruction addresses.

The LVP Unit predicts the values during dispatch, then forwards them speculatively to subsequent operations via the 620's rename busses. 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. Speculatively-forwarded values are tagged with the uncommitted loads they depend on, and these tags are propagated to the results of any subsequent dependent instructions. Meanwhile, uncommitted loads execute in the load/store pipe, and the predicted values are verified by either a CVU address match or a comparison against the actual values retrieved by the loads. Once a load is verified, all the dependent operations are either ready for in-order completion and can release their reservation stations (in the case of a correct prediction), or restart execution with the correct load values (if the prediction is incorrect). Since the load/store unit supports multiple non-blocking loads on cache misses, verifying a predicted value can take up to dozens of cycles, allowing the processor to speculate several levels down the dependency chain beyond the load, executing instructions and resolving branches that would otherwise be blocked by true dependencies.

The worst-case penalty for an incorrect load 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 the load value to be committed or corrected. Since the load value comparison takes an extra cycle beyond the standard two-cycle load latency, the dependent instruction will reissue and execute with the correct load value one cycle later than it would have had there been no prediction. In addition, the earlier incorrect speculative issue may cause a structural hazard that prevents 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 dependencies), there is no penalty, since the dependent instruction can issue as soon as the loaded value is available, in parallel with the value comparison in the load/store pipeline. In any case, due to the LCT which accurately prevents incorrect predictions, 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 may end up occupying their reservation stations for one cycle longer than they would have had there been no prediction

4.2. The Alpha AXP 21164 Machine Model

Our in-order processor model, summarized in Figure 5, differs from the actual AXP 21164 [BK95] in three ways. First, we do not fully model the MAF (miss address file) which enables nonblocking L1 cache misses on the 21164. Rather, we assume an abstract pipelined memory subsystem which allows any number of nonblocking misses (this is more aggressive than the actual 21164 and will tend to understate our results for LVP). Second, in order to allow speculation to occur in our LVP configurations, we must compare the actual data returned by the data cache and the predicted data. Since the distance between the data cache and the writeback is already a critical path in hardware, the comparison requires an extra stage before writeback. The third modification, the addition of the *reissue* *buffer*, allows us to buffer instruction dispatch groups that contain predicted loads. With this feature, we are able to redispatch instructions when a misprediction occurs with only a single-cycle penalty. The latter modifications apply only to our LVP configurations, and not to the baseline 21164 model.

In order to keep the AXP 21164 model as simple as possible, when any one of two dispatched loads is mispredicted then all of the eight possible instructions in flight are squashed and reissued from the reissue buffer regardless of whether or not they are dependent on the predicted data. Since the 21164 is unable to stall anywhere past the dispatch stage, we are unable to predict loads that miss the L1 data cache. However, when an L1 miss occurs, we are able to return to the non-speculative state before the miss is serviced. Hence, there is no penalty for doing the prediction. The inability of our LVP Unit to speculate beyond an L1 cache miss in most cases means that the LVP Unit's primary benefit is the provision of a zero-cycle load [AS95].

Typically, we envision the CVU as a mechanism for reducing bandwidth to the cache hierarchy (evidence of this is discussed in Section 6.1 and Section 6.5). However, since the 21164 is equipped with a true dual-ported cache and two load/store units it is largely unaffected by a reduction in bandwidth requirement to the L1 cache. In addition to reducing L2 bandwidth, the primary benefit of the CVU in the 21164 model is that it enables those predictions identified as constants to proceed regardless of whether or not they miss the L1 data cache. Hence, the only LVP predictions to proceed in spite of an L1 cache miss are those that are verified by the CVU.

5. Experimental Framework

Our experimental framework consists of three main phases: trace generation, LVP Unit simulation, and microarchitectural simulation. All three phases are performed for both operating environments (IBM AIX and DEC OSF/1).

For the PowerPC 620, traces are collected and generated with the TRIP6000 instruction tracing tool. TRIP6000 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. For the Alpha AXP 21164, traces are generated with the ATOM tool [SE94], which also captures user state instruction, value and address references only. The instruction, address, and value traces are fed to a model of the LVP Unit described earlier, which annotates each load in the trace with one of four value prediction states: no prediction, incorrect prediction, correct prediction, or constant load. The annotated trace is then fed in to a cycle-accurate microarchitectural simulator that correctly accounts for the behavior of each type of load. All of our microarchitectural models are implemented using the VMW framework [DS95], which enables significant productivity gains by allowing us to reuse and retarget existing models. The LVP 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 load to the microarchitectural simulator, rather than the full 32/64 bit values being loaded.

6. Experimental Results

We collected four types of results from our microarchitectural models: cycle-accurate performance results for various combinations of LVP Unit configurations and microarchitectural models for both the 620 and 21164; distribution of load latencies for the 620; average data dependency resolution latencies for the 620; and reductions in bank conflicts for the 620.

6.1. Base Machine Model Speedups with Realistic LVP

In Figure 6, we show speedup numbers relative to the baseline 620 for two LVP Unit configuration that we consider realistic (i.e. buildable within one or two processor generations) as well as two idealized LVP Unit configurations. The two realistic configurations, *Simple* and *Constant*, are described in Table 2. To explore the limits of load value prediction, we also include results for the *Limit* and *Perfect* LVP Unit configurations (also described in Table 2). The former is similar to the *Simple* 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, *Perfect*, 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.

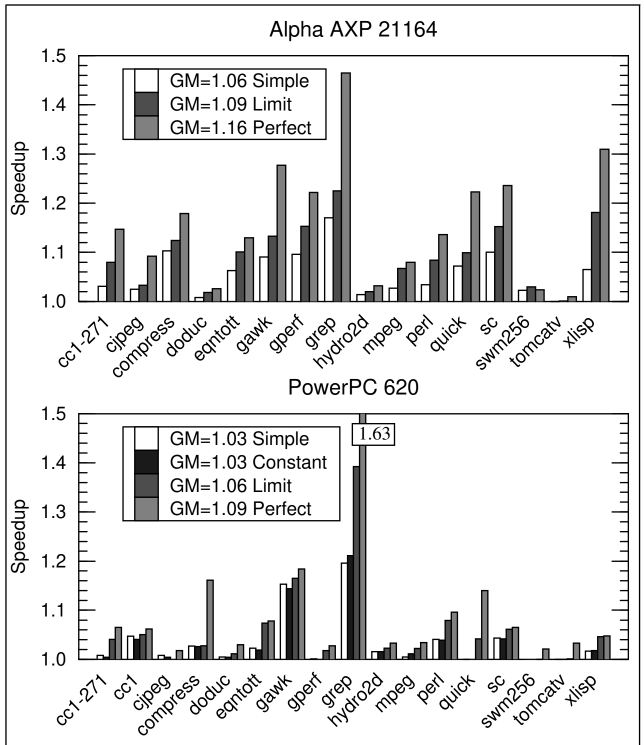

FIGURE 6. **Base Machine Model Speedups**.

Figure 6 also shows three of these four LVP configurations for the Alpha AXP 21164. We omit the *Constant* configuration from our 21164 simulations because it does not differ significantly from the *Simple* configuration on the 620 and because we have limited access to native Alpha CPU cycles for collecting traces.

In general, the 21164 derives roughly twice as much performance benefit from LVP as does the 620. We attribute this to two factors: its small first-level data cache (8K direct-mapped vs. the 620's 8-way associative 32K cache) benefits more from the CVU, and its in-order issuing policy makes it more sensitive to load latency, since it is forced to depend solely on the compiler to try to overlap it with other useful computation. The 620, on the other hand, is able to find other useful computation dynamically due to its out-of-order core.

Two benchmarks (*grep* and *gawk*) stand out for the dramatic performance increases they achieve on both models. 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.

The bandwidth-reducing effects of the CVU manifest themselves as lower first-level data cache miss rates for several of the benchmarks running on the 21164. For example, the miss rate for *compress* drops from 4.3% to 3.4% per instruction, a 20% reduction. Likewise, *eqntott* and *gperf* experience ~10% reductions in their miss rates, which translate into the significant speedups shown in Figure 6. Even *cjpeg* and *mpeg*, which gain almost nothing from LVP on the 620, eke out measurable gains on the 21164 due to the 10% reduction in primary data cache miss rate brought about by the CVU.

6.2. Enhanced Machine and LVP Model Speedups

To further explore the interaction between load value prediction and the PowerPC 620 microarchitecture, we collected results for the 620+ enhanced machine model described earlier in conjunction with four LVP configurations.

The results for these simulations are summarized in Table 6, where the third column shows the 620+'s average speedup of 6.1% over the base 620 with no LVP, and columns 4 through 7 show average additional speedups of 4.6%, 4.2%, 7.7%, and 11.3% for the *Simple*, *Constant*, *Limit*, and *Perfect* LVP configurations, respectively. In general, we see 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 baseline 620. The most dramatic examples of this trend are *grep* and *gawk*, which show very little speedup from the increased machine parallelism without LVP, but nearly double their relative speedups with LVP (with the *Simple LVP* configuration, *grep* increases from 20% to 33%, while *gawk* increases from 15% to 30%).

TABLE 6. **PowerPC 620+ Speedups.** Column 3 shows 620+ speedup relative to 620 with no LVP; columns 4-7 show additional LVP speedups relative to baseline 620+ with no LVP.

Bench	Base Cyc	620+	Simple	Constant	Limit	Perfect
cc1-271	93,371,808	1.057	1.006	1.003	1.045	1.071
cc1	117,571,998	1.112	1.012	1.006	1.021	1.041
cjpeg	2,818,987	1.126	1.001	1.011	1.000	1.021
compress	33,436,604	1.092	1.006	1.006	1.019	1.175
doduc	43.796,620	1.030	1.007	1.008	1.016	1.039
eqntott	18,823,362	1.049	1.029	1.037	1.083	1.082
gawk	28,741,147	1.009	1.293	1.240	1.327	1.272
gperf	4,893,966	1.108	1.026	1.019	1.045	1.034
grep	2,169,697	1.018	1.329	1.310	1.531	1.789
hydro2d	5,398,363	1.024	1.018	1.019	1.028	1.041
mpeg	5,394,984	1.192	1.012	1.023	1.036	1.031
perl	102,965,698	1.050	1.046	1.007	1.099	1.116
quick	704,262	1.019	1.000	0.999	1.051	1.170
sc	62,227,728	1.061	1.035	1.056	1.061	1.088
swm256	51,327,965	1.044	1.000	1.000	1.000	1.025
tomcatv	32,838,452	1.018	1.003	1.003	1.004	1.050
xlisp	44,844,605	1.052	1.022	1.026	1.059	1.058
GM		1.061	1.046	1.042	1.077	1.113

6.3. Distribution of Load Verification Latencies

In Figure 7 we show the distribution of load verification latencies for each of the four LVP configurations (*Simple*, *Constant*, *Limit*, and *Perfect*) on the 620 and 620+ machine models. That is, we show the percentage of correctly-predicted loads that are verified a given number of cycles after they are dispatched. The numbers shown are the sum over all the benchmarks. These results provide an intuitive feel for the number of cycles of load latency being eliminated by load value prediction. Clearly, if a larger percentage of loads have longer latency, LVP will prove more beneficial. Interestingly enough, the distributions for all four LVP configurations look virtually identical, which indicates that more aggressive LVP implementations (like *Limit* and *Perfect*) are uniformly effective, regardless of load latency. One would expect that a wider microarchitecture like the 620+ would reduce average load latency, since many of the structural dependencies are eliminated. These results counter that expectation, however, since there is a clear shift to the right in the distribution shown for the 620+. This shift is caused by the time dilation brought about by the improved performance of the 620+, which in turn is caused by its microarchitectural improvements as well as the relative improvement in LVP performance noted in Section 6.2.

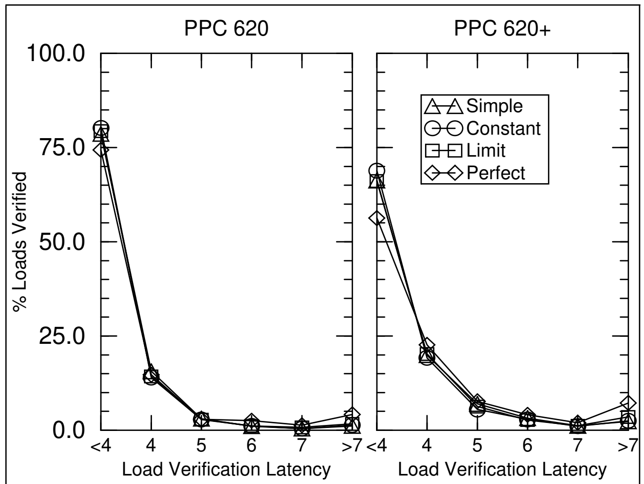

FIGURE 7. **Load Verification Latency Distribution**. Numbers shown are the percentage of correctly-predicted loads that are verified a given number of cycles after they are dispatched.

6.4. Data Dependency Resolution Latencies

The intent of load value prediction is to collapse true dependencies by reducing memory latency to zero cycles. To confirm that this is actually happening and to quantify the dependencies being collapsed, we measured the average amount of time an instruction spends in a reservation station waiting for its true dependencies to be resolved. The results are summarized in Figure 8, which categorizes the waiting time reductions by functional unit type. The numbers shown are the average over all the benchmarks, normalized to the waiting times without LVP. We see that instructions in the branch (*BRU*) and multi-cycle integer (*MCFX*) units experience the least reductions in true dependency resolution time. This makes sense, since both branches and move-from-special-purpose-register (*mfspr*) instructions are waiting for operand types (link register, count register, and condition code registers) that the LVP mechanism does not predict. Conversely, the dramatic reductions seen for floating-point (FPU), single-cycle fixed point (SCFX), and load/store (LSU) instructions correspond to the fact that operands for them are predicted. Furthermore, the relatively higher value locality of address loads shown in Figure 2 corresponds well with the dramatic reductions shown for load/store instructions in Figure 8. Even with just the *Simple* or *Constant* LVP configurations, the average dependency resolution latency for load/store instructions has been reduced by about 50%.

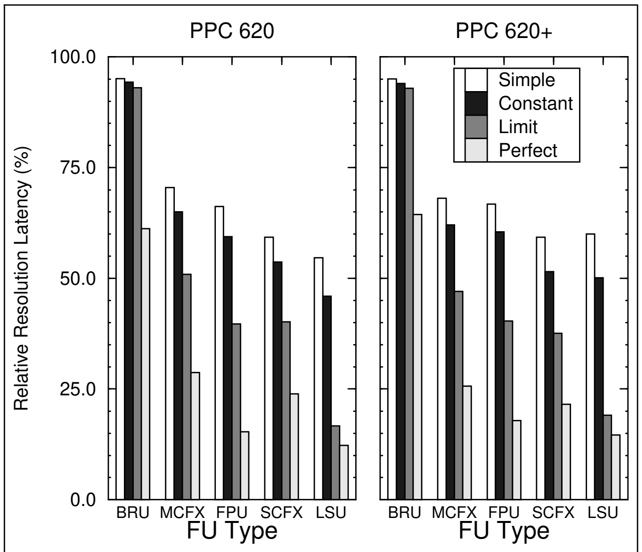

FIGURE 8. **Data Dependency Resolution Latencies.** Cycles spent waiting for operands are normalized to baseline 620/620+ models.

6.5. Bank Conflicts

The purpose of the CVU is to reduce memory bandwidth by eliminating the need for constant loads to access the conventional memory hierarchy. In our 620 and 620+ models, this benefit manifests itself as a reduction in the number of bank conflicts to the two banks of the first-level data cache. On the 620, in any given cycle, both a load and a store can attempt to access a data cache port. If both accesses are to the same bank, a conflict occurs, and the store must wait to try again the next cycle. On our 620+ model this problem is aggravated, since up to two loads and a store can attempt to access the two available banks in each cycle.

In Figure 9, we show the fraction of cycles in which a bank conflict occurs for each of our benchmarks running on the 620 and 620+ models. Overall, bank conflicts occur in 2.6% of all 620 simulation cycles for our benchmark set, and 6.9% of all 620+ cycles. Our *Simple* LVP Unit configuration is able to reduce those numbers by 8.5% and 5.1% for the 620 and 620+, respectively, while our *Constant* configuration manages to reduce them by 14.0% and 14.2% (we are pleased to note that these reductions are relatively higher than those shown in Table 4, which means the CVU tends to target loads that are, on average, more likely to cause bank conflicts).

Interestingly enough, a handful of benchmarks (*gawk*, *grep*, *hydro2d*) experience a slight increase in the relative number of cycles with bank conflicts as shown in Figure 9. This is actually brought about by the time dilation caused by the increased performance of the LVP configurations, rather than an increase in the absolute number of bank conflicts. One benchmark--*tomcatv*--did experience a very slight increase in the absolute number of bank conflicts on the 620+ model. We view this as a second-order effect of the perturbations in instruction-level parallelism caused by LVP, and are relieved to note that it is overshadowed by other factors that result in a slight net performance gain for *tomcatv* (see Table 6).

7. Conclusions and Future Work

We make three major contributions in this paper. First, we introduce the concept of *value locality* in computer system storage locations. Second, we demonstrate that load instructions, when examined on a per-instruction-address basis, exhibit significant amounts of value locality. Third, we describe *load value prediction,* a microarchitectural technique for capturing and exploiting load value locality to reduce effective memory latency as well as bandwidth requirements. We are very encouraged by our results. We have shown that measurable (3% on average for the 620, 6% on average for the 21164) and in some cases dramatic (up to 21% on the 620 and 17% on the 21164) performance gains are achievable with simple microarchitectural extensions to two current microprocessor implementations that represent the two extremes of superscalar design philosophy.

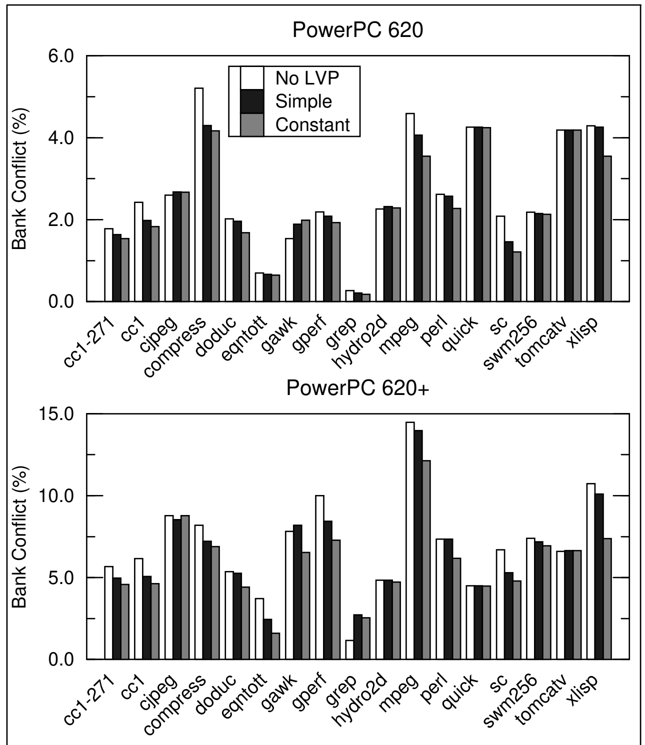

FIGURE 9. **Percentage of Cycles with Bank Conflicts**.

We envision future work proceeding on several different fronts. First of all, we believe that the relatively simple techniques we employed for capturing value locality could be refined and extended to effectively predict a larger share of load values. Those refinements and extensions might include allowing multiple values per static load in the prediction table by including branch history bits or other readily available processor state in the lookup index; or moving beyond history-based prediction to computed predictions through techniques like value stride detection. Second, our load classification mechanism could also be refined to correctly classify more loads and extended to control pollution in the value table (e.g. removing loads that are not latency-critical from the table). Third, the microarchitectural design space should be explored more extensively, since load value prediction can dramatically alter the available program parallelism in ways that may not match current levels of machine parallelism very well. Fourth, feedback-directed compiler support for rescheduling loads for different memory latencies based on their value locality may also prove beneficial. Finally, more aggressive approaches to value prediction could be investigated. These might include speculating down multiple paths in the value space or speculating on values generated by instructions other than loads.

Acknowledgments

This work was supported in part by ONR grant no. N00014-96-1-0928. We also gratefully acknowledge the generosity of the Intel Corporation for donating numerous fast Pentium Pro-based workstations for our use. These systems reduced our simulation turn-around-time by more than an order of magnitude. We also wish to thank the IBM Corporation for letting us use the TRIP6000 instruction tracing tool in our work.

References

[AS95] Todd M. Austin and Gurindar 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.

[ASKL81] Walid Abu-Sufah, David J. Kuck, and Duncan H. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Transactions on Computers*, C–30(5):341–356, May 1981.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.

[ASW+93] 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.

[BK95] Peter Bannon and Jim Keller. Internal architecture of Alpha 21164 microprocessor. *COMPCON 95*, 1995.

[CB94] Tien-Fu Chen and Jean-Loup Baer. A performance study of software and hardware data prefetching schemes. In *21st Annual International Symposium on Computer Architecture*, pages 223–232, 1994.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, April 1991.

[CMCH91] 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.

[CMT94] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, October 1994.

[DNS95] Trung A. Diep, Christopher Nelson, and John 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.

[DS95] Trung A. Diep and John Paul Shen. VMW: A visualization-based microarchitecture workbench. *IEEE Computer*, 28(12):57–64, 1995.

[Gwe94] Linley Gwennap. Comparing RISC microprocessors. In *Proceedings of the Microprocessor Forum*, October 1994.

[Har80] Samuel P. Harbison. *A Computer Architecture for the Dynamic Optimization of High-Level Language Programs*. PhD thesis, Carnegie Mellon University, September 1980.

[Har82] Samuel 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.

[Jou88] N. P. Jouppi. Architectural and organizational tradeoffs in the design of the MultiTitan CPU. Technical Report TN-8, DEC-wrl, December 1988.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *17th Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, May 1990.

[KEH93] David Keppel, Susan J. Eggers, and Robert R. Henry. Evaluating runtime-compiled, value-specific optimizations. Technical report, University of Washington, 1993.

[Kro81] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *8th Annual International Symposium on Computer Architecture*, pages 81–87. IEEE Computer Society Press, 1981.

[LTT95] David Levitan, Thomas Thomas, and Paul Tu. The PowerPC 620 microprocessor: A high performance superscalar RISC processor. *COMPCON 95*, 1995.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, 1992.

[RD94] K. Roland and A. Dollas. Predicting and precluding problems with memory latency. *IEEE Micro*, 14(4):59–67, 1994.

[Ric92] Stephen E. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. Technical report, Sun Microsystems Laboratories, 1992.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[SIG91] SIGPLAN. *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26, Cambridge, MA, September 1991. SIGPLAN Notices.

[Smi81] J. E. Smith. A study of branch prediction techniques. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 135–147, June 1981.

[Smi82] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3):473–530, 1982.

[SW94] Amitabh Srivastava and David 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.

[TFMP95] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A modified approach to data cache management. In *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 93–103, December 1995.

[YP91] 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.