Symbolic Cache: Fast Memory Access Based on Program Syntax Correlation of Loads and Stores

Qianrong Ma
Server Globalization Tech.
Oracle Corporation
qianrong.ma@oracle.com

Jih-Kwon Peir
Lu Peng
CISE Department
University of Florida
{peir,lpeng}@cise.ufl.edu

Konrad Lai
Microprocessor Research Lab
Intel Corporation
konrad.lai@intel.com

Abstract

An increasing cache latency in next-generation processors incurs profound performance impacts in spite of advanced out-of-order execution techniques. One way to circumvent this cache latency problem is to predict load values at the onset of pipeline execution by exploiting either the load value locality or the address correlation of stores and loads. In this paper, we describe a new load value speculation mechanism based on the program syntax correlation of stores and loads. We establish a Symbolic Cache, which is accessed by the content of memory load and store instructions in early pipeline stages to achieve a zero-cycle load. Performance evaluation using SPEC95 and SPEC2000 integer programs with SimpleScalar tools shows that the symbolic cache provides higher accuracy than both the memory renaming and the value prediction scheme, especially when hardware resources are limited.

1 Introduction

Today's high-performance processor pipeline permits overlapping instruction execution to achieve more than one Instruction Per Cycle (IPC) average execution rate. The available Instruction-Level Parallelism (ILP) constrains this parallel execution because dependent instructions must wait for the data produced by the source instructions. The severity in terms of execution delays depends primarily on the speed that the producer instruction can generate the needed data.

Memory load latency presents a classical pipeline bottleneck even when the data is located in the first-level cache. This is because the load data is not ready until late stages of the pipeline while the dependent instruction normally requires the data at an earlier stage. This load-to-use delay exacerbates in recent high-performance microprocessors in which multi-cycle, first-level caches become the norm [10, 13, 12, 8, 7]. As the cache size, clock frequency and complexity of microarchitecture continue to increase in next-generation processors, it is estimated that the first-level cache accesses may consume two to five cycles [1]. This increasing load latency from caches will further lengthen the load-to-use delay and will have profound performance impact in spite of advanced out-of-order execution techniques [1, 2].

In Figure 1, a conceptual out-of-order execution pipeline is partitioned into two phases. First, an instruction is fetched, decoded, renamed, and issued through the *front-end* of pipeline stages. Afterwards, the instruction is executed (including memory access) and committed through the *back-end* of pipeline stages. In order to be hazard-free, a source instruction must produce the data early before executions of its dependent instructions. In other words, a critical producer, when it is fetched and issued at the same cycle with its dependent instructions, needs to generate the result in the front-end of the pipeline in order to avoid any stall of the dependent instructions. Such a hazard-free memory load instruction is called a *zero-cycle* load.

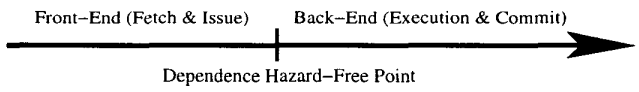

Figure 1: Pipeline partitioning and hazard-free point

There have been several attempts to achieve a zero-cycle load by predicting and speculating the load value in the front-end of the processor pipeline. The most aggressive way is to predict the load value at the onset of pipeline execution. A load-value history table is established and accessed using the Program Counter

0-7695-1200-3/01$10.00 © 2001 IEEE

(PC) of the load. This scheme allow loads to bypass caches completely to achieve a zero-cycle load. However, the lack of close correlation between the instruction address and the value of the load makes the prediction accuracy low [9, 15, 11].

Another way to circumvent pipeline hazards caused by the cache latency is to predict the load address at the onset of pipeline execution so that a cache access can start speculatively without going through the normal decode, rename, and address generation stages [5, 6, 4, 1]. Existing address prediction methods exploit regular patterns, such as stride-based address patterns, and irregular but repeated patterns, such as addresses for traversing link-based data structure. However, the difficulty of predicting a significant portion (over 30% [1]) of load addresses that do not fall into these two categories remains. Furthermore, a lengthy cache access is still required even with a correctly predicted address.

Memory renaming techniques establish dynamic dependence correlations between stores and loads [14]. A separate storage element called a *value file (VF)* is used to save the correlated data. When a memory load instruction is fetched, an indirect access to the value file based on the PC of the load can retrieve the data without going through a lengthy cache access. Studies show that there are many more loads that consume the value from the same producer than those loads which repeat the same value or address from the previous instance of the same load. Therefore, there is a better chance to obtain the correct load value using memory renaming through the value file than that based on the load value/address locality. This approach, however, requires additional hardware to establish the correct dependence links among stores and loads.

In this paper, we exploit a new avenue to speculatively obtain the load value in front-end stages of the pipeline. First, we observe that store-load and load-load correlations are established in software and often displayed in the program syntax in the form of a base register ID plus a displacement value. Therefore, instead of establishing accurate dependence links in hardware, we can use contents of store/load instructions to approximate the dependence. Second, applications exhibit spatial locality among memory references. Such locality can also be observed in the program syntax. A typical example is found in stack accesses during procedure calls as consecutive stack accesses differ only by a small displacement value. Therefore, it is reasonable to establish store/load dependences on a large block granularity to capture the spatial data reference locality.

Based on these observations, we propose a *Symbolic Cache (SC)*. An SC is a small separate cache that is accessed at early front-end stages based on the content of memory reference instructions. The speculative data retrieved from the SC can trigger the execution of dependent instructions to avoid delays. In comparison, the SC works similar to value prediction schemes. Instead of predicting the load value based on the past history of the load, the value can be more accurately predicted based on the syntax (i.e. the base register ID and the displacement) of the loads. Similarly, the general function of the SC is also comparable to that of the VF in the memory renaming approach. Using the content of loads, the SC approach can eliminate additional complexity of establishing correct dependences and extra tables to allow correct accesses to the VF.

Performance evaluations using SimpleScalar tools and SPEC95/SPEC2000 integer programs show higher prediction accuracy in the SC approach in comparison with other data speculation methods. This is especially true when hardware resources are limited for constructing the additional tables and caches. The remaining paper is organized as follows. We will first provide motivations and observations for the proposed method in the next section. This is followed by discussions of design and related issues for establishing the SC in Section 3. In Section 4, performance evaluations of three data speculation methods are given. Finally, Section 5 concludes the paper.

2 Background and Motivation

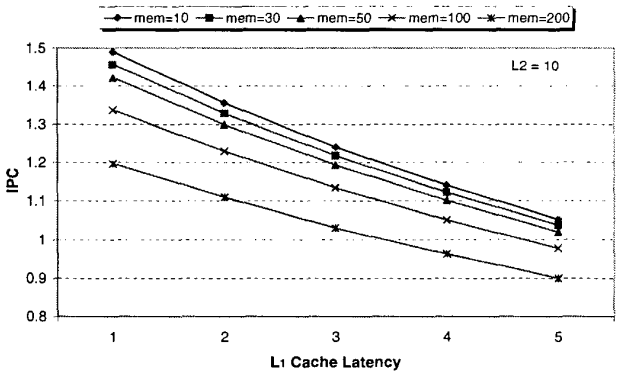

Figure 2: IPCs of various cache/memory latencies

It is well-known that increasing cache and memory latencies deteriorate pipeline performance. Figure 2 shows average IPCs of several SPEC95 and SPEC2000 integer benchmarks with various cache and memory latencies. The simulation is performed using the SimpleScalar tool on a Sun/Solaris environment [3]. We

vary first-level data cache (L_1) access latencies from 1 to 5 cycles with memory latencies of 10, 30, 50, 100, and 200 cycles. The second-level cache (L_2) latency is fixed at 10 cycles and other parameters follow the default values. We simulate an 8-wide, out-of-order pipeline, with 32KB, 4-way, instruction and data L_1 caches and a 1MB, 4-way, combined L_2 cache. The results indicate that each cycle reduction of the L_1 access delay improves the IPC by about 7–10%. Techniques based on load value speculation to hide the cache latency are the main focus in this paper.

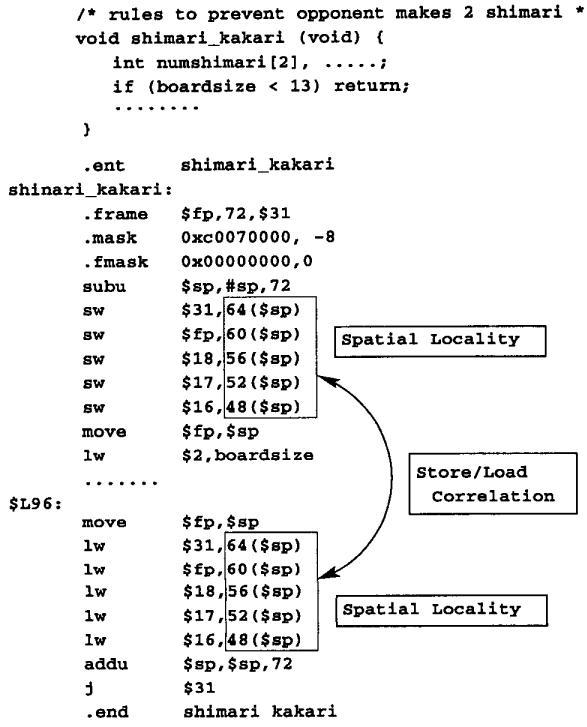

Figure 3: Syntax corrections of store/load and their spatial locality

There are two key observations that establish the foundation of the *Symbolic Cache (SC)*. First, store-load and load-load correlations are often displayed in the program syntax in the form of a base register ID plus a displacement value. Second, such a simple memory reference syntax also exhibits spatial locality. Both of these properties can be observed in accessing global-allocated variables, local stacks for procedure calls, as well as the heap for dynamic allocated variables. Figure 3 shows the source and the assembly codes of a simple function from *Go* of SPEC95. This function is invoked many times to determine a strategy for the next move. Note that we omit the function body and only show stack accesses for saving and restoring registers. The store/load dependence can be established perfectly with a matching pair of base register (*sp*) and displacement. Meanwhile, the spatial locality among stores and among loads is also evident from the displacement values. In general, these properties exist in all functions. Other program constructs such as stores/loads in unrolled loops, handling spoiling code, etc. also show similar syntax correlations and locality properties.

Given these syntax correlations among stores and loads in applications, we can establish a small SC to save and speculate load values in the front-end of pipeline stages.

3 Establishing Symbolic Caches

An SC is a small cache which is addressed by the content of load/store instructions. The SC can be accessed once loads/stores are fetched out of the instruction cache. As a result, pipeline stages involving register file access, address generation/translation, and cache access for loads can be bypassed to shorten the load-to-use latency. The impact of pipeline performance using an SC is very similar to that of using the VF in memory renaming techniques [14], where the speculative load data is fetched out of the VF indirectly through a pointer table.

It is essential to properly extract the *symbolic address* from the content of load/store instructions in order to capture the syntax correlation among stores and loads. Meanwhile, the aliasing problem among symbolic addresses need to be considered. A typical memory instruction consists of an opcode, a register source/destination, and a memory source/destination. Intuitively, we can use the memory source/destination to form a 32-bit symbolic address as illustrated in Figure 4. The least significant 16 bits are extracted from the displacement value, and the base register ID (5 bits) are inserted next to the displacement. Although simple, this approach suffers aliasing problems because multiple memory addresses can be mapped to the same symbolic address. In addition, this simple symbolic address formation creates other access and alignment problems:

- **Aliasing of Symbolic Address:** With the simple address mapping shown in Figure 4, a 32-bit memory address is represented by a 21-bit symbolic address. Potentially, many different memory addresses can be expressed by the same symbolic address. An obvious example can be found in stack accesses for saving and restoring

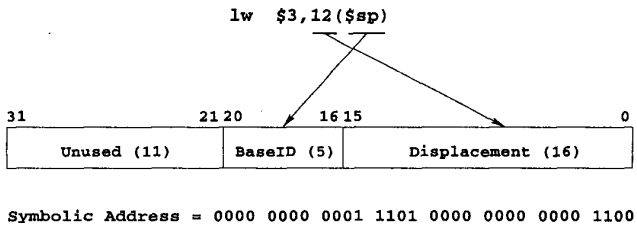

Figure 4: Extracting symbolic address from memory instructions

registers during procedure calls. Although accessing different stack frames in each procedure, the same stack pointer (*sp*) with similar ranges of offset values are commonly used. The content in the SC for the saving registers is likely to be destroyed by nested procedure calls and cannot be saved for restoring the same registers.

- **Uneven SC Index Distribution:** It is well-known that displacement values in memory references are unevenly distributed with a high percentage of '0'. Using a portion of the displacement bits as the index of the SC has a potential to generate many conflict misses.
- **Word/Byte Alignment:** The most difficult problem lies in the difference of the line boundary between a symbolic and a L_1 cache lines. This is due to the fact that offset bits of a cache line are not the same between the symbolic and the real addresses. It is essential to properly align the data layout in the symbolic cache according to the symbolic address to capture the spatial locality of memory references.

3.1 Procedure Coloring and Index Randomization

In order to alleviate the stack access aliasing problem in different procedure calls, we apply a simple procedure coloring technique. A global counter called *P-color* is implemented. The P-color is incremented whenever a procedure call is encountered. The P-color is decremented after returning from a procedure call. The P-color can be incremented contiguously in nested or recursive procedures before being decremented. Stack accesses between a caller and its callees can be differentiated by the P-color to avoid conflicts in accessing the SC.

The P-color can be concatenated with the symbolic address for stack accesses. The width of the P-color counter is flexible as long as it can be fitted into the remaining bits of the symbolic address. Figure 5 (a) illustrates the symbolic address after adding a 6-bit P-color. It is important to know that the P-color is only applied to stack accesses which use *sp* as the based register. Other memory accesses do not add the P-color to allow sharing of global variables among different procedures.

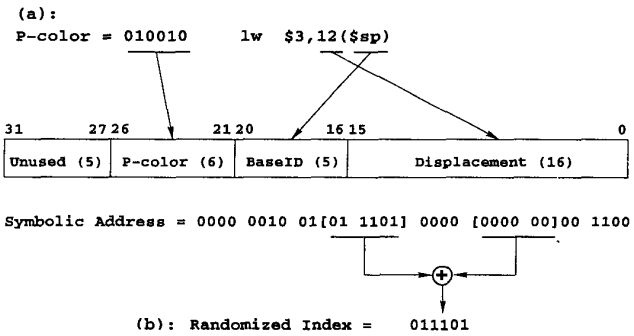

Figure 5: (a) Adding procedure color to symbolic address; (b) Index randomization in SC

An uneven distribution of the index bits extracted directly from the displacement value has a potential to create heavy conflict misses to the SC. This is due to the fact that high-order displacement bits are often all zeros. This problem can be dealt with by a simple randomization technique. Instead of extracting index bits from the symbolic address directly, randomized index bits can be formed by *exclusive-ORing* the original index bits from the displacement with the bits from the base register ID and the P-color as illustrated in Figure 5 (b). In this example, we assume the SC has 64 sets with 64-byte line size. The six index bits are obtained by *exclusive-ORing* normal index bits in position 6 to 11 with the base register ID and partial P-color bits starting at position 16 through 21.

3.2 Word/Byte Alignment

One remaining issue is the data alignment between the SC and the L_1 data cache. The symbolic address within a cache line, i.e. the last few offset bits, may not be the same as the offset bits in the real address. In order to exploit spatial reference locality, the cache line fetched from L_1 need to be rearranged in the SC such that the data layout can be aligned with the symbolic address. The basic alignment algorithm works as follows. When a memory request misses the SC, the target cache line is fetched from the memory hierarchy and loaded into the SC. The target byte/word is placed in the SC according to offset bits of the symbolic address. For example, assume there are eight

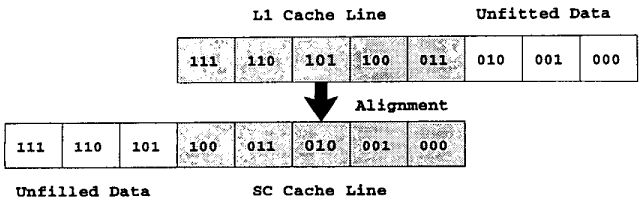

Figure 6: Data alignment in symbolic cache

access units in a cache line as shown in Figure 6. The symbolic offset of the target unit is *010* while the offset of the real address is *101*. In this case, the target data *101* is loaded into unit *010* in the SC. The remaining units are loaded according to the location of the target unit. There are thus two aspects to consider for achieving a proper data alignment.

- **Granularity of data alignment:** Depending on memory access granularity, it is conceivable that the data alignment can be performed at byte, half-word, word, or double-word level.
- **Handling underflow/overflow data:** Since the line boundary of the SC and the L_1 cache may be different, only partial line can be filled on each SC miss. In addition, there are excessive data from the target L_1 cache line that cannot fit into the requested line location in the SC. The simplest and most natural solution is to only fill a partial SC line and drop those unfitted data. Other options include fetching two adjacent L_1 lines for each requested SC line, and/or to search and place the overflow L_1 data into the correct second SC line.

Performance evaluation on these design options will be given in the next section. Note that the SC is updated by every store instruction. Data coherency between the SC and the L_1 cache is unnecessary due to the speculative nature of the SC. It is also important to keep in mind that the primary goal of establishing the SC is to provide a fast load to shorten the load-to-use latency. The simplicity of design is essential to minimize adverse effects of managing a separate SC.

4 Performance Evaluation

Performance evaluations of three load value speculation methods including the last/stride based load value prediction (*VP*), the memory renaming (*MR*), and the proposed symbolic cache (*SC*) are given. Our primary focus is to compare the prediction accuracy among the three mechanisms. Therefore, detailed pipeline and memory hierarchy cycles are not considered in this paper. All simulations are carried out on the functional model of SimpleScalar. Ten integer programs, *Gcc, Go, Li, M88k, Perl*, and *Vortex* from SPEC95 and *Gzip, Mcf, Parse*, and *Twolf* from SPEC2000 are used for this study. For each workload, we use the first 50 million instructions to warm up the caches and tables, then collect simulation statistics from the next 500 million instructions.

4.1 Data Alignment

In order to confine our performance evaluation space, we first study the alignment granularity. Table 1 shows matches of the least-significant two bits between the symbolic and the real addresses with different memory access granularity in the ten integer programs. The two numbers in each entry represent the percentages of matched and mismatched memory addresses. On the average, 82.5%, 4.5% and 13.0% of memory references are accessing word, half-word and byte respectively. Mismatches of the least two bits for the three access granularity are about 0%, 0.8% and 5.7%. SPEC2000 programs show higher percentages of sub-word memory references with more mismatches of the least 2 bits. For simplicity, we decided to apply only word alignment, which can capture about 94% of correct data alignment regardless the granularity of memory accesses.

	Byte	H-word	Word	Total
Gcc	4.3/3.1	12.2/0.9	79.5/0	96.0/4.0
Go	0/0	0/0	100/0	100/0
Li	17.8/3.4	0/0	78.8/0	96.6/3.4
M88k	6.0/0	0.2/0	93.8/0	100/0
Perl	7.0/9.0	2.8/0.9	80.3/0	90.1/9.9
Vortex	0.5/0.6	2.4/0	96.5/0	99.4/0.6
Gzip	5.7/7.0	9.2/1.6	76.5/0	91.4/8.6
Mcf	5.2/7.7	1.8/2.3	83.0/0	90.0/10.0
Parse	15.6/13.5	3.8/1.2	65.9/0	85.3/14.7
Twolf	10.9/12.9	4.3/0.8	71.1/0	86.3/13.7
Aver.	7.3/5.7	3.7/0.8	82.5/0	93.5/6.5

Table 1: Percentage of match/mismatch of least two bits for accessing byte, half-word and word

With regards to line-fills on SC misses, preliminary studies show that options of filling the entire SC line by fetching potentially more than one L_1 cache lines, and/or placing the entire targeted L_1 line into the SC provide very limited benefit. Therefore, we consider only simple partial SC line-fills and drop any unfitted data in the following evaluation.

	Gcc	Go	Li	M88k	Perl	Vortex	Gzip	Mcf	Parse	Twolf	Average
No-color	71.9	47.8	67.0	86.9	68.6	70.5	70.2	82.3	60.9	74.4	70.0
2-bit	74.2	48.7	69.2	88.2	71.9	74.5	70.3	82.4	62.9	74.4	71.7
4-bit	74.2	48.7	69.2	88.2	71.8	74.5	70.3	82.4	62.8	74.4	71.7

Table 2: Accuracy with/without the P-color

	Gcc	Go	Li	M88k	Perl	Vortex	Gzip	Mcf	Parse	Twolf	Average
4-way	68.5	43.9	65.2	83.1	65.7	62.6	70.1	80.9	55.8	70.8	66.6
Fully-asso	74.8	49.4	69.7	88.5	72.5	76.5	70.3	82.5	65.8	74.5	72.5
4-way, random	74.2	48.7	69.2	88.2	71.8	74.4	70.3	82.4	62.8	74.5	71.7

Table 3: Accuracy with index randomization

4.2 Sensitivity of P-color and Index Randomization

Table 2 shows the accuracy of load value speculation using a 8KB SC with different P-color bits. In general, we observe about 2% average improvement by adding the P-color. However, three out of four SPEC2000 workload show no improvement with P-color. After examining dynamic procedure calls in these programs, we found that there are very few nested calls. For instance, in Gzip, about 98% of the calls are labeled at level 6. We also observe that there is no benefit by increasing the number of bits in the P-color. Deep analysis of application programs reveals that perfectly-nested or deeply-recursive procedures that benefit with more P-colors rarely exist. The actual execution path traverses back and forth among different levels of procedures. With more P-colors, more levels of procedure invocation can be differentiated. However, due to a small SC, the data from ancient ancestors is difficult to hold anyway.

The benefit of index randomization is more evident in Table 3, in which the accuracy of three 8KB SC configurations are displayed. By randomizing the index, a 4-way set-associative SC can achieve the accuracy approaching to that of a fully-associative SC. On the other hand, without index randomization, the 4-way design degrades the accuracy by about 9%.

4.3 Comparison of Three Data Speculation Methods

The accuracy of three load value speculation mechanisms are evaluated. For a fair comparison, we simulate comparable hardware with respect to additional storage requirement among the three mechanisms. The VP scheme establishes a value history table with tags for matching the PC of a load. In addition, an increment value is needed in each entry to accommodate a stride-based predictor. Besides the value file, two additional tables are constructed in the MR scheme. The Store/Load Cache (*SLC*) saves pointers to the value file. The SLC is addressed by the PC of loads and stores with tags for matching the correct PC for indirect accesses to the value file. The Store-Address Cache (*SAC*) also records pointers to the value file. The SAC is accessed by load/store addresses for establishing load/store correlations. Again, address tags are necessary to make a correct correlation. The SC is simply a data cache addressed by symbolic addresses. There is no extra hardware except for a small tag array in which each tag is associated with a 64-byte symbolic line.

We consider six configurations for accuracy comparisons as shown in Table 4. The hardware requirement is represented by the total number of entries in the respective tables and caches. Because of the additional tag arrays, the storage requirement for the VP and the MR are about 40-50% and 10-15% more than that of the SC in each configuration. Note that in this first-cut estimation, extra control hardware is not considered.

Figure 7 plots the accuracy curves for the three data speculation methods. General speaking, the SC has the highest accuracy, especially with small configurations. The results indicate that the SC can capture syntax correlations and spatial locality effectively with small cache sizes as the prediction accuracy approaches 70% by using a tiny 2KB SC. The MR scheme, on the other hand, requires more than 8 times of hardware storage to achieve a comparable accuracy. The MR scheme performs poorly with small configurations primarily because of heavy misses to the

Config.	VP (word)	MR VF (word)	MR SLC (link)	MR SAC (link)	SC (line)
1	128	64	128	128	16
2	256	128	256	256	32
3	512	256	512	512	64
4	1024	512	1024	1024	128
5	2048	1024	2048	2048	256
6	4096	2048	4096	4096	512

Table 4: Six configurations for accuracy comparisons

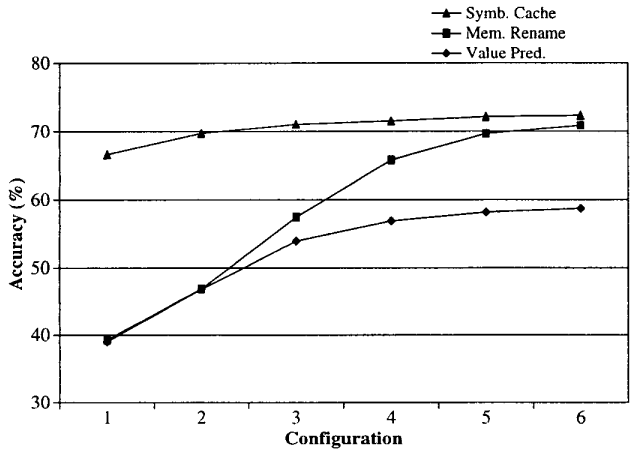

Figure 7: Average accuracy of three data speculation methods

small SLC/SAC for establishing correct links. The MR scheme shows better improvement when the configuration size increases. With bigger SLC/SAC, data dependence links can be built more precisely than those approximated by the symbolic address. However, the SC still maintains the edge of capturing spatial locality. The last/stride value predictor has the worse accuracy as we expected. This confirms a poor correlation between the load value and its instruction address.

For example, considering the third configuration with a 4KB SC, the average prediction accuracy are 53.9%, 57.4% and 71.1% for the VP, the MR, and the SC respectively as shown in Figure 8. For the ten integer programs, *Gcc, M88k, Perl, Vortex, Gzip, Mcf,* and *Twolf* show good syntax correlations with over 70% of prediction accuracy. Among them, *Gcc, Perl, Vortex,* and *Twolf* demonstrate a large improvement over the MR scheme. Again, this is because of heavy SLC/SAC misses. *Go,* and *Parse* show worse syntax correlations. These two programs have very dynamic procedure call patterns with small amount of register

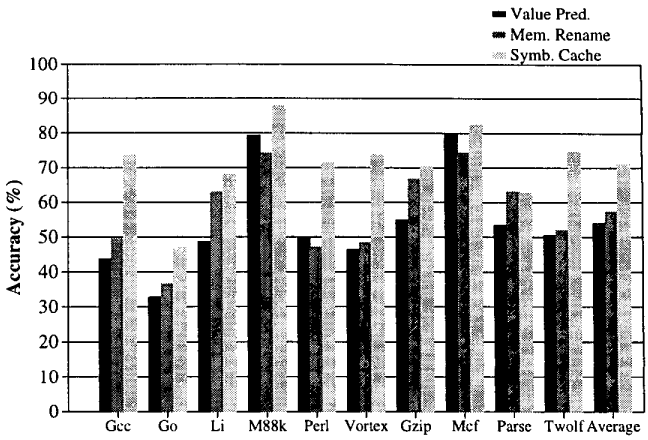

Figure 8: Accuracy of three data speculation methods of individual programs (configuration 3)

saving/restoring in each procedure to benefit the SC approach.

5 Conclusion

A new load data speculation method based on instruction syntax correlations of stores and loads has been introduced in this paper. Instead of hardware establishing the store/load correlation during runtime, the proposed method establishes a small symbolic cache to capture existing syntax correlations and memory reference locality. The symbolic cache is addressed by the content of store/load instructions to enable data accesses in the front-end of the processor pipeline to shorten load-to-use latency. Performance evaluation of SPEC integer programs has demonstrated that the proposed method can achieve 70% accuracy with a small 2KB symbolic cache. Such accuracy requires much larger storages using other memory renaming techniques.

Acknowledgment:

This work is supported in part by NSF grants MIP-9624498, EIA-0073473 and by Intel research donations. Anonymous referees' comments are very helpful to improve the paper.

References

[1] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated Load-Address Predictors," *Proc. of 26th Annual Int'l Symp. on Computer Architecture*, Atlanta, GA, May 1999, pp. 54–63.

[2] M. Bekerman, A. Yoaz, F. Gabbay, S. Jourdan, M. Kalaev, and R. Ronen, "Early Load Address Resolution Via Register Tracking," *Proc. of 27th Annual Int'l Symp. on Computer Architecture*, Vancouver, Canada, June 2000, pp. 306–315.

[3] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0", Technical Report #1342, CS Department, Univ. of Wisconsin-Madison, June 1997.

[4] C. Chen and A. Wu, "Microarchitecture Support for Improving the Performance of Load Target Prediction," *Proc. of 30th annual international symposium on Microarchitecture*, Triangle Park, NC, Dec. 1997, pp. 228–234.

[5] R. Eickemeyer and S. Vassiliadis, "A Load-Instruction Unit For Pipelined Processors," *IBM Journal of Research and Development*, Vol. 37(4), pp. 547–564, July 1993.

[6] J. Gonzalez, and A. Gonzalez, "Speculative Execution via Address Prediction and Data Prefetching," *ACM 1997 Int'l Conf. on Supercomputing*, Vienna, Austria, Aug. 1997, pp. 196–203.

[7] T. Horel and G. Lauterbach, "UltraSPARC-III: Designing Third-Generation 64-Bit Performance", *IEEE Micro*, May/June 1999, pp. 73–85.

[8] R. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro*, Vol. 19(2), March/April 1999, pp. 24–36.

[9] M. Lipasti, C. Wilkerson and J. Shen, "Value Locality and Load Value Prediction", *Proc. of the 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Oct. 1996, pp. 138–147.

[10] D. Papworth, "Tuning the Pentium Pro Microarchitecture," *IEEE Micro*, Vol. 16(2), April 1996, pp. 8–15.

[11] Y. Sazeides and J. Smith, "The Predictability of Data Values," *Proc. of 30th annual international symposium on Microarchitecture*, Triangle Park, NC, 1997, pp. 248–258.

[12] T. Slegel, et al., "IBM's S/390 G5 Microprocessor Design," *IEEE Micro*, Vol. 19(2), March/April 1999, pp. 12–23.

[13] P. Song, "IBM's Power3 to Replace P2SC," *Microprocessor Report*, Vol. 11(15), Nov. 1997, pp. 1–11.

[14] G. Tyson and T. Austin, "Improving the Accuracy and Performance of Memory Communication Through Renaming," *Proc. of 30th annual international symposium on Microarchitecture*, Triangle Park, NC, 1997, pp. 218–227.

[15] K. Wang, and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors," *Proc. of 30th annual international symposium on Microarchitecture*, Triangle Park, NC, Dec. 1997, pp. 281–290.