Frequent Value Locality and Value-Centric Data Cache Design *

Youtao Zhang
zhangyt@cs.arizona.edu

Jun Yang
junyang@cs.arizona.edu

Rajiv Gupta
gupta@cs.arizona.edu

Department of Computer Science
The University of Arizona, Tucson, Arizona 85721

ABSTRACT

By studying the behavior of programs in the SPECint95 suite we observed that six out of eight programs exhibit a new kind of value locality, the *frequent value locality*, according to which a few values appear very frequently in memory locations and are therefore involved in a large fraction of memory accesses. In these six programs ten distinct values occupy over 50% of all memory locations and on an average account for nearly 50% of all memory accesses during program execution. This observation holds for smaller blocks of consecutive memory locations and the set of frequent values remains quite stable over the execution of the program.

In the six benchmarks with frequent value locality, on an average 50% of all cache misses occur during the reading or writing of the ten most frequently accessed values. We propose a new data cache structure, the *frequent value cache* (FVC), which employs a *value-centric* approach to caching data locations for exploiting the frequent value locality phenomenon. FVC is a small direct-mapped cache which is dedicated to holding only frequently occurring values. The value-centric nature of FVC enables us to store data in a compressed form where the compression is achieved by encoding the frequent values using a few bits. Moreover this simple compression scheme preserves the random access to data values in a cache line.

Our experiments demonstrate that by augmenting a direct mapped cache (DMC) with a direct mapped FVC of size no more than 3 Kbytes we can obtain reductions in miss rates ranging from 1% to 68%. In fact we observed that higher reductions in miss rates can be achieved by augmenting a DMC with a small FVC as opposed to doubling the size of DMC for the `124.m88ksim` and `134.perl` benchmarks.

1. INTRODUCTION

Recent research has demonstrated that values produced by executing instructions exhibit a high degree of value locality, that is, multiple executions of the same instruction often produce the same value [3, 8]. Value locality has been exploited in the design of value reuse and prediction mechanisms for superscalar processors.

In this paper we show that another kind of locality, that we refer to as the *frequent value locality*, is also quite prevalent in programs. If we track the values involved in memory accesses, we observe that at any given point in the program's execution, a small number of distinct values occupy a vast fraction of these referenced locations. In fact we observed that in six out of eight programs in SPECint95 test suite, ten distinct values occupy over 50% of all referenced memory locations throughout the execution of the program. Therefore it is not surprising that a large fraction of memory accesses involve a small number of values. In the six programs with frequent value locality, on an average, ten distinct values account for nearly 50% of all memory accesses during program execution. The set of frequent values remains quite stable throughout the execution of the program and these values are scattered fairly uniformly throughout the memory.

One application of the phenomenon of frequent value locality that we explore in this paper is in the design of data caches. We observed that on an average 50% of cache misses for the six SPECint95 benchmarks which exhibit frequent value locality can be attributed to the reading and writing of ten most frequently accessed values. Therefore we design a *value-centric* data cache structure which caches values in memory locations only if they contain frequently accessed values. This structure is used in conjunction with a traditional data cache to achieve improved performance. Due to its special treatment of frequently accessed values, a small value-centric data cache can provide significant reductions in cache miss rates and hence off-chip traffic. Thus, it has the potential of leading to power efficient cache designs.

In this paper we specifically focus on improving the performance of on-chip Direct-Mapped data Caches (DMCs). We only consider write-back caches as opposed to write-through caches because write-through caches are known to generate much higher levels of traffic. We augment a traditional DMC with a small direct-mapped frequent value cache (FVC) to retain the identities of memory locations that contain one of the frequent values which has been recently evicted from the

* Supported by DARPA award no. F29601-00-1-0183 and NSF grants CCR-0096122, EIA-9806525, and CCR-9996362 to the University of Arizona.

ASPLOS 2000 Cambridge, MA Nov. 12-15 , 2000

DMC. Since a disproportionate number of misses are caused by evicted frequent values, a small FVC structure dedicated to frequent values provides reductions in miss rates that otherwise can only be achieved using much larger DMCs.

Since the number of highly frequent values is small, less than 10 according to our study, the values can be stored in the FVC using an encoding requiring a few bits instead of using full words to store each value. Thus, FVC retains values by using compact encoded representations of frequent values to achieve data compression. Moreover, this simple compression scheme preserves the ability to randomly access data values in a cache line. This compression allows more data to be held on-chip and therefore reduces data cache miss rates, offchip traffic and thus power consumption. The techniques we propose are particularly relevant for systems being designed today where power consumption is an important issue as the systems are often battery powered. The reduced miss rates should provide lower execution times. The reductions in traffic will directly result in corresponding reductions in power consumption. In fact code compression [2] is being widely studied today by researchers for the purpose of reducing power consumption [6, 7]. Packing of narrow width data operands in multimedia applications is also being explored by researchers [1, 9, 10]. The FVC introduces an approach to data compression centered around frequently accessed values. The technique incorporated in FVC encodes data values and is thus applicable to all data (not just narrow width data) and is useful in context of general purpose applications (not just multimedia applications).

Our experiments demonstrate that by augmenting a direct mapped cache (DMC) with a direct mapped FVC of size no more than 3 Kbytes we can obtain reductions in miss rates ranging from 1% to 68%. This is not surprising because due to the data compression scheme employed, the FVC was found to effectively cache well over four times the number of values that can be held in an equal sized DMC. In fact we observed that higher reductions in miss rates can be achieved by augmenting a DMC with a small FVC as opposed to doubling the size of DMC for the `124.m88ksim` and `134.perl` benchmarks.

In section 2 we present a study of frequent value locality in for SPECint95 benchmarks. In section 3 we describe the detailed design of *frequent value cache*. In section 4 we present a detailed evaluation of FVC in terms of miss rates and traffic.

2. FREQUENT VALUE LOCALITY

Frequently encountered values. We studied the behavior of the SPECint95 benchmarks to establish the existence of frequent value locality in real programs. Throughout this study as well as all other experiments described in this paper all benchmarks were executed on *reference* inputs. The programs were executed and by tracking the execution of all load and store instructions, the frequently encountered values were identified.

In this study we examined values that *occur* frequently in memory locations as well as values that are frequently *accessed* by memory operations. The accesses are accumulated over the entire execution of the program. However, the occurrence of values in memory locations was sampled every 10 million instructions and averaged over the entire set of collected samples. The memory locations that were considered at a given point included those that we viewed to contain values that are of interest to the program. Ideally, we consider the value in a location to be of **interest** if the memory location has been referenced (i.e., read or written) at some point in the program and has not been deallocated since. In practice we were able to track deallocations of stack memory but not that of heap memory. Therefore the results we present are based upon consideration of some uninteresting locations in addition to all interesting locations.

The results are summarized in Figure 1. The first six benchmarks exhibit high levels of *frequent value locality*. Not only do a mere ten values *occur* in at least 50% of memory locations, on an average nearly 50% of accesses involve just ten distinct values. The last two benchmarks, `129.compress` and `132.ijpeg`, exhibit very little frequent value locality. Therefore in the subsequent sections of this paper we only present results of experimental evaluations for the first six benchmarks. Although we did evaluate these two benchmarks, as expected, the improvements due to our techniques were quite small. Although we present detailed evaluations of integer benchmarks, we also performed the above study for SPECfp95 benchmarks. As the results in Figure 2 show, the floating point benchmarks also exhibit a high degree of frequent value locality.

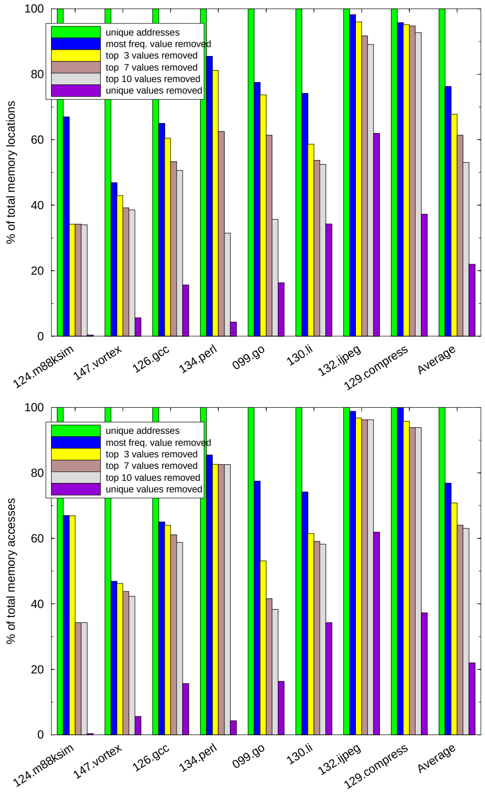

Figure 1: Frequently encountered values in SPECint95 .

The graphs in Figure 1 give us summary information. We also studied the occurrences and accesses of frequent values throughout program execution. The graphs in Figure 3 show the behavior of the `126.gcc` benchmark which terminates after executing approximately 315 million instructions on the

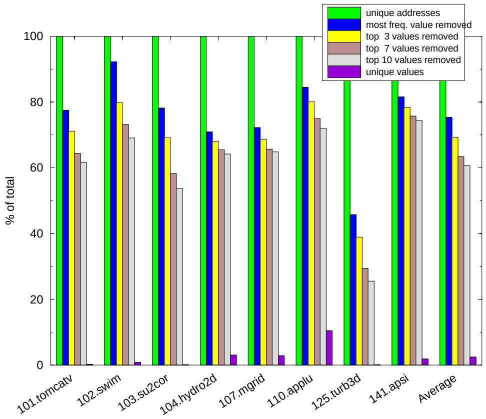

Figure 2: Frequently encountered values in SPECfp95 .

reference input. In these graphs the top most line represents the total number of locations and accesses. The subsequent curves give us an idea of how many locations and accesses correspond to the top ten most frequently occurring and accessed values. The difference between the first (top most) and second curve in the number of locations and accesses to the most frequent value. The difference between the second and third curve is the number of locations and accesses to the next two values. The difference between the third and fourth curve is the number of locations and accesses to the next four frequent values. Finally the difference between the fourth and fifth curve is the number of locations and accesses to the next three most frequently used values. From the results in Figure 3 we can see that the top ten values occupy nearly 50% of the memory locations and appear in approximately 40% of accesses. Moreover this observation holds true for the entire execution of the program.

The bottom most line represents the number of unique values that occur in memory and are accessed respectively. It is interesting to note that the number of distinct values, indicated by the bottom most curve, is only around 20% of total number of memory locations or accesses. This may be surprising considering that variables representing loop counters should typically generate a large number of distinct values. We believe that although many variables, such as loop counters, do generate a large number of distinct values, these values are often not written to memory but simply computed in registers and then discarded (e.g., a register may be assigned to a loop counter). This explanation may suggest that results we obtained may vary greatly depending upon the quality of register allocation due to two reasons. First the register allocator may not assign registers to variables that generate large numbers of distinct values. Second if the registers are not used efficiently, the loads and stores will be generated more frequently. However, in practice these reasons should not be of serious concern. Even the simplest register allocator can be expected to assign registers to loop counters since they frequently referenced. Moreover, in integer codes such as the SPECint, the vast majority of load and store instructions are generated due to imprecise aliasing information and not due to lack of registers. In any case, we performed all our experiments using optimized code - we used code generated using `gcc` under the `-O3` option.

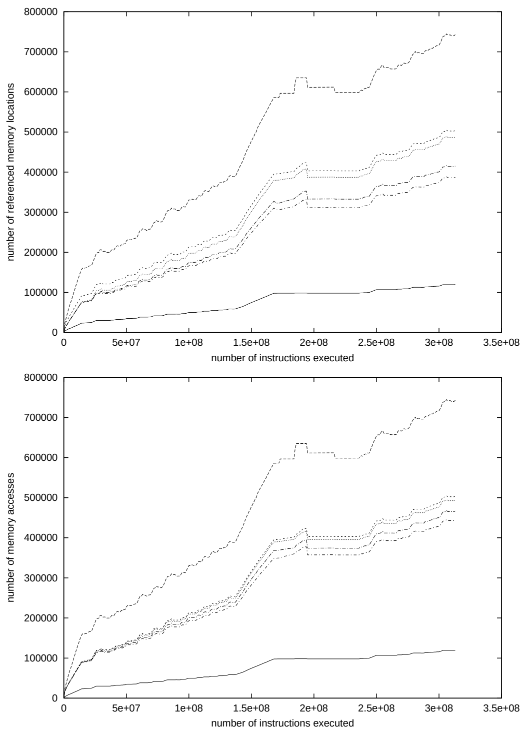

Figure 3: Frequent value locality in 126.gcc benchmark.

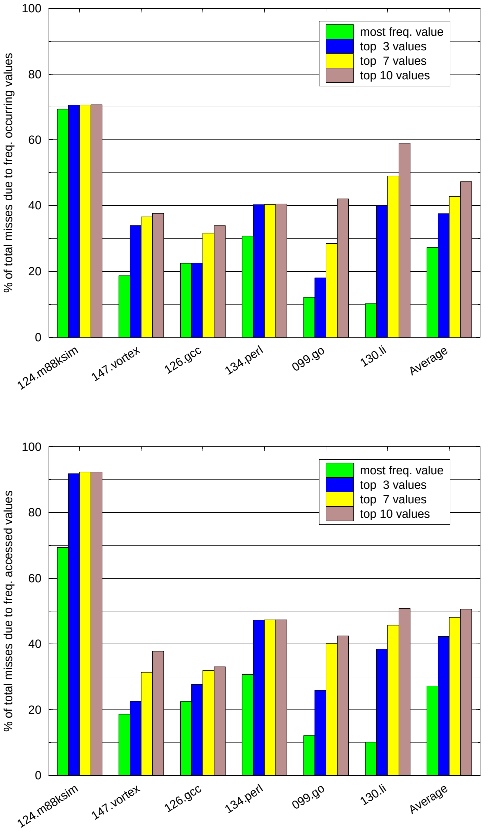

Figure 4: Cache miss behavior: 16Kb DMC and line size of 16 bytes.

`099.go`		`124.m88ksim`		`126.gcc`		`130.li`		`134.perl`		`147.vortex`	
accessed	occurring	accessed	occurring	accessed	occurring	accessed	occurring	accessed	occurring	accessed	occurring
0	ffffffff	0	0	0	0	0	0	0	0	0	0
1	0	1	3	1	e7	3	1	1	1	44034500	4
351a	1	2	1c	2	403	40230f30	3	100	20207878	1	ffffffff
2	2	401dcb90	401ddd30	4	ffffffff	4	5	40267e70	20782078	402b35bc	30
3	3	1c	401de6fc	a	1b	40233a08	40234974	2	20787820	4	50005
4	4	ffffffff	401dbfc0	40034	20000029	103	4	40267e0c	31787820	4128bdbc	40004
2ed	349	100b8	401dd5a0	3	40034	4022d0f8	6	ffffffff	78207878	402324b0	5
13	351a	40264728	8048bf7d	40204260	3	303	303	401e7594	78782078	ffffffff	30657079
ffffffff	1c1	401d5d68	60d12	ffffffff	2	401e6d5c	103	401d4068	78787878	405aba98	746e6f63
28	2ed	402050bc	1db82340	4021470c	42	40233a44	106	40269b88	78202020	44033af0	6

Table 1: Frequently occurring and accessed values (hex. integer) ordered by decreasing frequency.

Cache miss study. We also studied the role of frequently occurring and frequently accessed values in cache misses. We counted the cache misses that occurred when values that were involved were either top ten frequently occurring or top ten frequently accessed values. Figure 4 shows the distribution of cache misses attributable to these values. As we can see, on an average the cache misses attributable to frequently occurring values is slightly less than 50% and those attributable to frequently accessed values is slightly over 50%. Since the latter is slightly higher, we use the top ten frequently accessed values in evaluating our value-centric cache design.

Frequently occurring vs accessed values. Our results show that the behavior of frequently occurring values is not much different from frequently accessed values. In particular, one may expect that frequently accessed values would cause significantly greater number of cache misses in comparison to frequently occurring values. This observation can be explained as follows.

First consider the list of frequently occurring and frequently accessed values in Table 1. There is a significant overlap in these values (i.e., many of the frequently occurring values are also the frequently accessed ones). Second the distribution of frequently occurring values in memory is quite uniform. Therefore whether the program accesses a part of the memory frequently or it accesses the memory more uniformly, it is likely to encounter frequently occurring values with the same frequency. In other words, there is not a significant difference between frequently occurring and frequently accessed values. This second point is illustrated by the data in Figure 5 which represents the snapshot of memory at a point when `126.gcc` was half way through its execution. The referenced memory was broken into blocks of 800 consecutive locations each. To mimic a line size of 8 words per line, we divided the 800 words into 100 lines with 8 words each. In this graph each point represents a measure of the presence of frequent values in each line of a memory block. We computed the average number of frequent values per line, over the 100 lines in a block, and plotted this average. In this experiment we only considered the top seven frequently occurring values. As we can see, the measure is around four throughout the memory, that is, frequently occurring values are distributed quite uniformly in memory.

Sensitivity of frequently accessed values to program input. We ran the six programs also on the *test* and *train* inputs and compared the frequently accessed values for these inputs with those determined earlier for reference inputs. The results for individual benchmarks are shown in Table 2. In the table X/Y indicates that X values among top Y frequently accessed values for *test* or *train* inputs were also present in the top Y frequently accessed values for reference inputs. Therefore from these results we can see that there is approximately 50% overlap in the frequently accessed values across the three sets of inputs we considered. Upon closer examination we found that the small frequently accessed values are not input sensitive while the large values, which are typically addresses, are sometimes input sensitive.

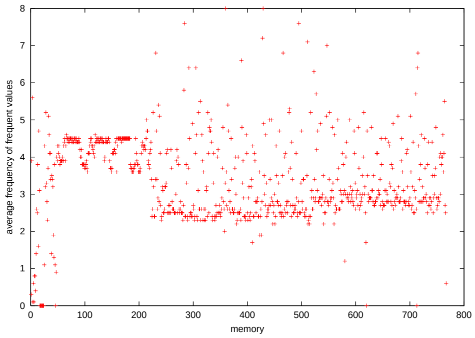

Figure 5: Frequent occurrence of some values in smaller memory blocks.

Program	Test		Train	
`099.go`	7/7	7/10	7/7	10/10
`124.m88ksim`	2/7	2/10	3/7	3 /10
`126.gcc`	7/7	10/10	4/7	4 /10
`130.li`	7/7	10/10	3/7	5 /10
`134.perl`	2/7	4/10	3/7	4 /10
`147.vortex`	3/7	4/10	3/7	4 /10

Table 2: Input sensitivity study.

Finding frequently accessed values. The values in Table 1 were obtained by running the programs to completion or near completion on reference inputs. However, to exploit the frequent value locality phenomenon in practice, we need a fast method for identifying the frequently accessed values. This is indeed possible using profiling because the set of top ten frequently accessed values remains quite stable throughout the execution of a program. In Table 3 the percentage of execution time after which the identity and the order of frequently accessed values never changes is given. We give this data for finding the top 1, 3, and 7 values. As we can see, in most cases the values are found very quickly. In case of `124.m88ksim` it takes longer to find the order of values. However, if we are interested in the identity of these values and not their relative ordering, as is the case for the value-centric cache, then we can find the frequently accessed values faster. For 124.m88ksim, after 18% of the execution the top 3 frequently accessed values appear in the top 10 list and after 39% of the execution the top 7 frequently accessed values appear in the top 10 list.

Program	Instructions executed (in billions)	Top most values found after % of execution		
		1	3	7
099.go	18.8	0%	0.07%	0.5%
124.m88ksim	4.1	0%	63%	70%
126.gcc	0.3	0%	10%	18%
130.li	1.2	0%	0.3%	0.3%
134.perl	20.9	0%	0.3%	0.4%
147.vortex	8.2	0%	9%	29%

Table 3: Finding frequently accessed values.

Frequent value locality vs value locality. The traditional notion of value locality [3, 8] is quite distinct from frequent value locality. In particular, value locality is defined in context of individual instructions and is observed by examining the values generated by multiple executions of a specific instruction. Therefore it is not surprising that value locality has found applications in uncovering instruction level parallelism by impacting the scheduling of individual instructions. In contrast, frequent value locality is observed when values involved across all load and store instructions are examined. Therefore it is more appropriate for applications such as cache design where the overall reference behavior of the program is of concern.

Inspite of the above distinction, a simple experiment we performed does indicate that for many programs frequently value locality and value locality go hand in hand. We determined the percentage of referenced addresses whose contents remain constant throughout a program's execution. For a location that was allocated multiple times each allocation were treated separately. The results in Table 4 show that for the six benchmarks this number is high indicating that they are likely to exhibit load value locality. These are also the six benchmarks that have high frequent value locality. The benchmarks for which values stored at addresses do not remain constant are 129.compress and 132.ijpeg. These two programs also do not exhibit frequent value locality.

Program	Constant Addresses
099.go	78.2%
124.m88ksim	99.3%
126.gcc	61.8%
130.li	28.8%
134.perl	80.4%
147.vortex	79.9%
129.compress	3.2%
132.ijpeg	6.7%

Table 4: Addresses with constant values.

3. FREQUENT VALUE CACHE

FVC design basics. The results of the above study motivated us to design a dedicated direct-mapped structure, the *frequent value cache*, which is used in conjunction with a traditional DMC to enhance the DMC's performance (see Figure 6). Therefore both caches are accessed in parallel and a hit in one of them results in a overall hit. The FVC is a *value-centric* structure which can hold values of memory locations as long as the locations contain one of the frequently accessed values. It allows data values to be held in a compressed form where each frequent value is represented by a few bits instead of requiring a full word.

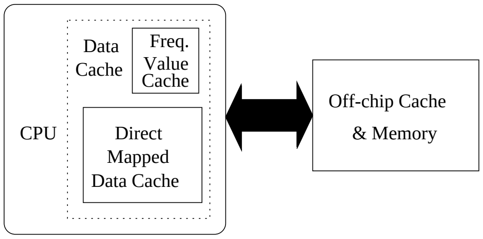

Figure 6: Frequent value cache.

In designing the FVC we have two goals. First we would like to develop a design for FVC so that it improves the performance of the DMC but as far as possible does not lower performance. Second we would like to maximize the benefits of FVC so that the benefits of adding a FVC far exceed the benefits achievable by simply increasing the size of the DMC.

To achieve the first goal we keep the operation of DMC the same and in addition we place data in the FVC only when it is evicted from DMC (there is one exception to this rule which will be discussed later; however, even in that situation misses are likely to be eliminated not introduced). Therefore a hit in DMC in the absence of FVC is almost always a hit in DMC in the presence of FVC. In addition, frequent value hits may occur in the FVC when a miss in the DMC occurs.

To achieve the second goal the FVC is so designed that a value at an address location is either cached in the DMC or, if the value at the location is a frequent value, it may be cached in the FVC; however, *the value at an address location is never cached in both DMC and FVC simultaneously*. This design choice ensures that the FVC provides a given level of performance improvement at the smallest cost in terms of its size. The second goal is also met by employing the following data compression scheme.

Data compression using frequent value encodings. The *value-centric* nature of the FVC creates an interesting opportunity for compressing the size of data stored in the FVC. Since FVC only contains frequently accessed values, we do not need to store the data values explicitly but they can be stored in a more *compact encoded* form. The encoded data cache field in FVC consists of multiple subfields, one for each word in the original DMC cache line. The value in each field identifies a specific frequent value or indicates that a non-frequent value resides at that location. If the field length is 3 bits, eight codes are available of which one can be used to indicate the absence of a frequent value and seven codes can be used for the seven top most frequent values. Therefore in its encoded form the representation of frequent values can be compressed by a factor of ten (e.g., from 32 bits to 3 bits if top 7 frequently accessed values are exploited). An example, illustrating the above is shown in Figure 7. This encoding scheme compresses data values without sacrificing random access to values in each cache line.

Frequent Value (32 Bits)	0	-1	1	2	4	8	10	infrequent values
3 Bit Encoding	000	001	010	011	100	101	110	111

0	1000	0	99999	-1	10	1	-1

DMC: 8 word uncompressed cache line (256 Bits)

000	111	000	111	001	110	010	001

FVC: 8 word compressed cache line (24 Bits)

Figure 7: Compressed encoded data in FVC.

Hits and misses in FVC. The DMC and FVC are accessed in parallel and a hit in at most one of two caches can occur (see Figure 8). A *read hit* in the FVC occurs when the address tag matches and the value being referenced is a frequently accessed value. If the tag does not match or the tag matches but the value is an infrequently accessed value, a read miss occurs. Similarly a *write hit* in the FVC occurs when there is a tag match and the value being written is a frequently accessed value. If the tag matches but the value is an infrequent one, a miss occurs.

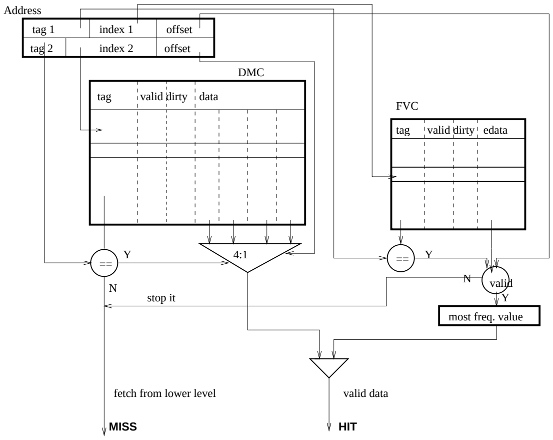

Figure 8: FVC: detailed design.

Transfer of data among DMC, FVC, and memory. It is possible that upon a read or a write to a memory location, although there is a tag match in the FVC, the value being read/written is an infrequently accessed value. Since the FVC cannot provide/store this value, in this case, the actual data cache line is brought in from memory. The other locations in the line are appropriately updated using latest values from FVC and then the line is entered into the DMC. At this time the corresponding line from FVC is evicted.

Cached lines are put into the FVC in two situations. First when a cache line is evicted from DMC it is written back to memory and at the same time we also store the identity of locations containing frequently accessed values in FVC. When a line is moved into FVC another line may be evicted which is then also written back to memory. The second situation in which new values are written to the FVC arises when there is a write miss in both the DMC and the FVC. If the value being written is a frequent value, then it is written into the FVC and all other values in the same line are marked as infrequent. The evicted line from FVC (if any) is written back to memory. If the other words in the cache line are also written over with a frequent value before they are ever read, or if they are not referenced at all, cache misses are avoided. On the other hand if a location marked as containing an infrequent value is referenced a cache miss occurs. Thus, this strategy has the effect of either eliminating or delaying the cache miss.

FVC vs DMC access times. The time for using the FVC is dominated by the FVC access time because the decoding of the value found can be carried out very quickly. Dedicated registers for holding individual values can be provided and a simple select operation can be used to decode the accessed value. We used the CACTI [5] tool to estimate the access times of FVC and DMC for 0.8 micron technology to see if the inclusion of FVC will slow down the overall access time of the on-chip cache. The results are shown in Figure 9. As we can see, there are many configurations of DMC and FVC for which the access time of FVC is less than or equal to the access time of DMC. The access times of FVC are plotted under the assumption that the top seven frequently accessed values are exploited because in our experiments described in the next section we use a maximum of seven frequently accessed values. The small variation in access times of FVC with fixed number of entries are due to the varying sizes of tags determined by the DMC configuration.

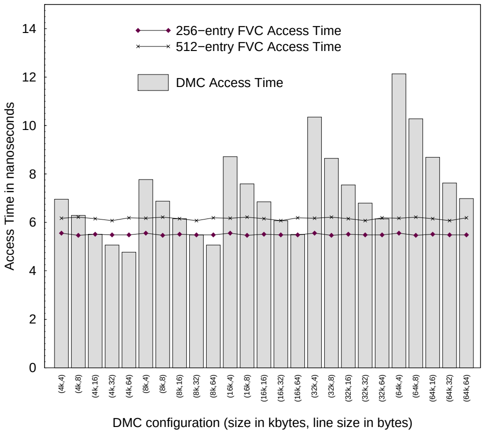

Figure 9: Access time of FVC vs DMC.

4. EXPERIMENTAL EVALUATION

The goals of the experiments were to study the performance improvements by FVC 's for varying sizes and configurations and comparing them with those that can be achieved by simply increasing the size of the DMC.

The performance enhancements of DMCs of varying configurations were determined. We considered DMCs of sizes 4, 8, 16, 32 and 64 Kbytes for line sizes of 16, 32 and 64 bytes in these experiments. The sizes of FVC were varied from 64 entries to 4096 entries. Three different configurations of FVC were considered corresponding to providing 1, 2, and 3 bits for encoding the frequent values. Therefore these FVCs exploited the most frequent, the top three, and the top seven frequently accessed values respectively. The encoded data field contains the same number of subfields as the number of words per line in the corresponding DMC. In all our experiments we ignore the effects of context switches on cache behavior.

Varying the size of FVC . We varied the number of entries in the FVC from as low as 64 to as high as 4096 and studied the changes in the enhancements of the DMC's performance. The FVC configuration used here exploited 7 frequently accessed values. The size of DMC was kept fixed at 16 Kbytes and a line size of 8 words. The results of our experiment are shown in Figure 10.

As expected the performance in terms of miss rate improves with the size of FVC. However, it is interesting to note that in some cases (`124.m88ksim` and `134.perl`) a very small sized FVC of 64 entries is sufficient to get all the performance improvements. For the other four benchmarks (`147.vortex`, `126.gcc`, `099.go`, `130.li`) the performance increases steadily as the size of the FVC is increased. The percentage reductions in miss rates due to incorporation of a FVC along with a DMC vary from a minimum of around 10% for `130.li` to maximum of well over 50% for `124.m88ksim`.

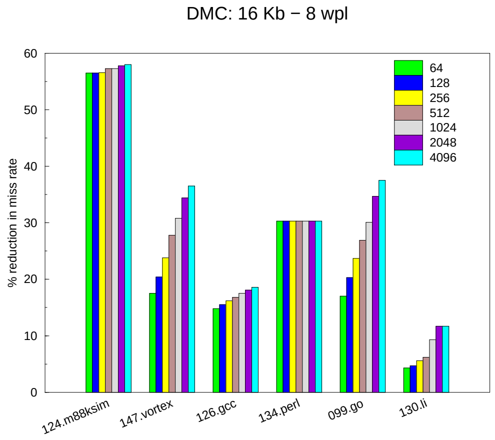

Figure 10: Miss rate reduction with FVC size.

Effectiveness of data compression. The reductions in traffic and miss rates shown above are substantial and are explained when we examine the contents of the FVC for an executing program. In Figure 11 the average percentage of frequent values in valid cache lines of the FVC are plotted. As we can see in most programs the number of such values is over 40%. Thus, a significant number of locations containing frequent values are being cached in the FVC which leads to a substantial reductions in the miss rates. More importantly in order to hold these values the FVC uses much less storage than a DMC. Given that a 32 byte DMC cache line is compressed into a 3 byte FVC cache line, and 40% of FVC locations contain frequent values, we can conclude from this experiment that FVC uses 4.27 ($= \frac{32}{3} \times 0.4$) times less storage to store cached data values than the DMC. Therefore we find that the compression scheme we employ is simple and effective while it also preserves random access to data values in a cache line.

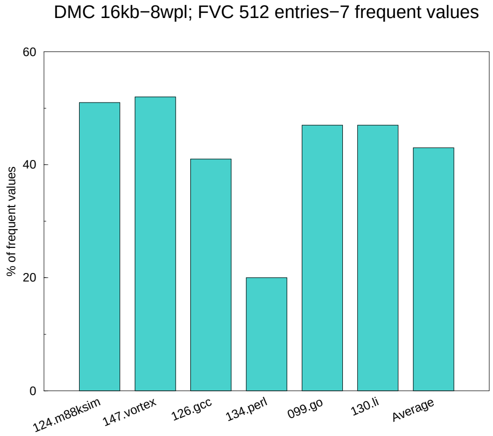

Figure 11: Frequent value content of FVC .

Exploiting varying number of frequently accessed values. Next we conducted experiments to evaluate the improvements that can be obtained by augmenting a DMC with a 512 entry FVC. We conducted these experiments to study the impact of varying the number of frequently accessed values that are exploited by the FVC (1, 3, and 7). A total of 12 DMC configurations for which access times of DMC is equal to or greater than access time of a 512 entry FVC were chosen.

The percentage reductions in miss rates are shown in Figure 12. From the graphs we can see that in some cases there is a substantial improvement in performance as we go from 1 frequent value to 3 frequent values. However, the improvements observed when we go from 3 frequent values to 7 frequent values is smaller.

All of the results clearly show that by exploiting a small number of frequently accessed values, substantial reductions in miss rates and off-chip traffic can be achieved. The percentage reductions in miss rates, and therefore also memory traffic, range from 1% to 68%.

Using a larger DMC vs using a FVC. Next we show that the performance improvements resulting from a small FVC of 0.125 Kbytes to 3 Kbytes can give large performance improvements which can sometimes exceed the performance improvement achieved by doubling the size of the DMC. As the miss rates in Figure 13 show, this observation is true for the benchmarks of `124.m88ksim` and `134.perl`. Therefore, for these benchmarks it is better to have a smaller DMC augmented with a small FVC instead of using a much larger DMC. For other benchmarks the larger DMC's provide lower miss rates. However, still the miss rates of the configurations with a FVC compare quite favorably with larger DMCs without a FVC.

Set-associativity and FVC. So far we have only considered the impact of incorporating a FVC along with a directed mapped cache. It is possible to use a FVC in conjunction with a set-associative cache. The FVC derives its improvement by eliminating a combination of *conflict* misses and *capacity* misses. It eliminates conflict misses because lines evicted from the main conventional cache (direct-mapped or set-associative) are moved to the FVC (except for the non-frequent values). It removes

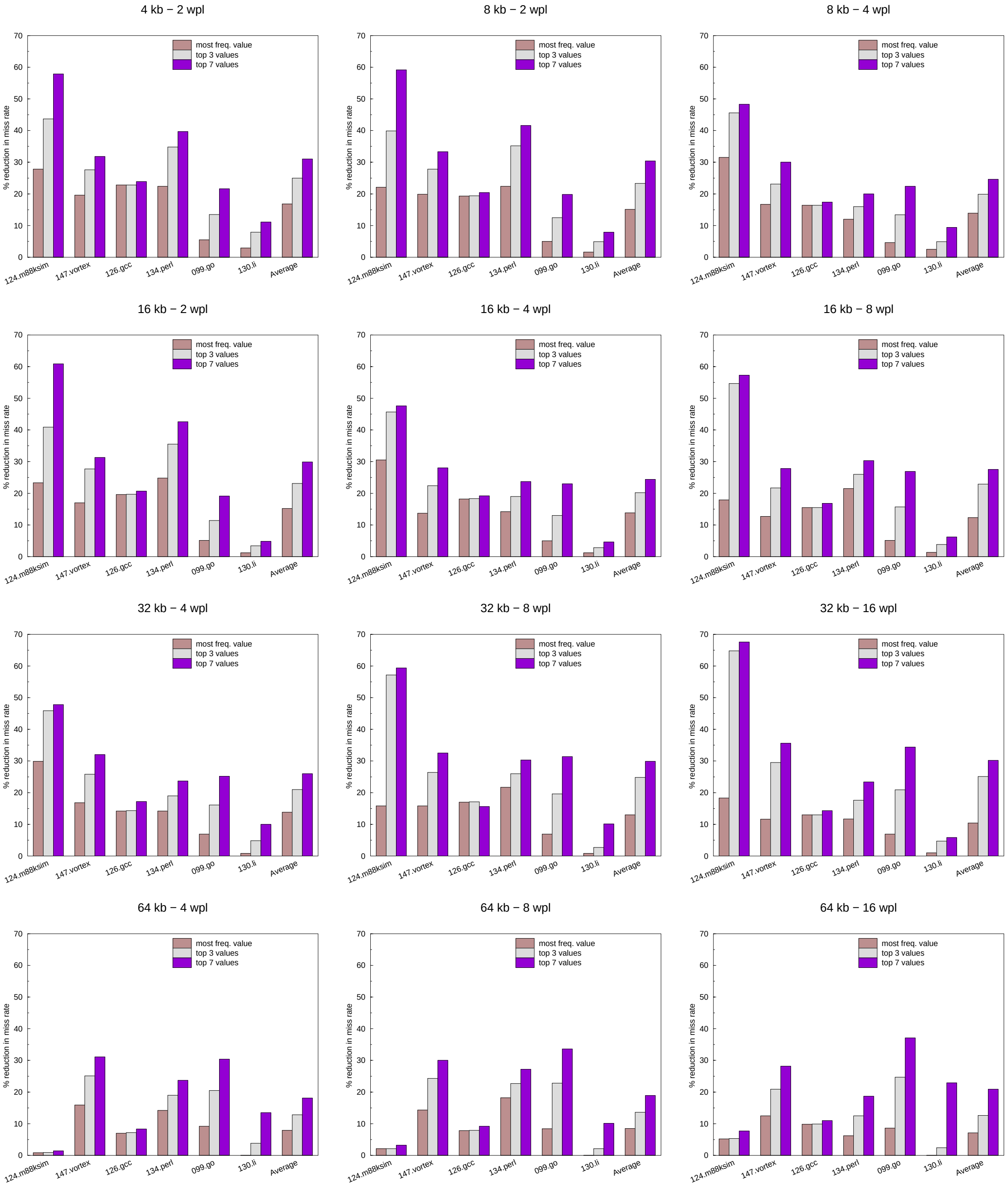

Figure 12: % reduction in miss rates: DMC vs DMC + FVC (top 1 vs 3 vs 7 frequent values).

DMC: line size = 2 words = 8 bytes		
FVC: 512 entries; 7 frequently accessed values		
benchmark	4Kb DMC + .375Kb FVC	8Kb DMC
	% misses	% misses
124.m88ksim	1.132	1.841
134.perl	4.090	5.209
FVC: 512 entries; 3 frequently accessed values		
	4Kb DMC + .25Kb FVC	8Kb DMC
124.m88ksim	1.514	1.841
134.perl	4.422	5.209
FVC: 512 entries; 1 frequently accessed value		
	4Kb DMC + .125Kb FVC	8Kb DMC
124.m88ksim	1.939	1.841
134.perl	5.261	5.209

DMC: line size = 4 words = 16 bytes					
FVC: 512 entries; 7 frequently accessed values					
8Kb DMC + .75Kb FVC	16Kb DMC	16Kb DMC + .75Kb FVC	32Kb DMC	32Kb DMC + .75Kb FVC	64 Kb DMC
% misses	% misses	% misses	% misses	% misses	% misses
0.701	1.101	0.577	1.050	0.548	1.050
3.361	3.524	2.687	3.502	2.672	3.502
FVC: 512 entries; 3 frequently accessed values					
8Kb DMC + .5 Kb FVC	16Kb DMC	16Kb DMC + .5 Kb FVC	32Kb DMC	32Kb DMC + .5 Kb FVC	64 Kb DMC
0.737	1.101	0.598	1.050	0.568	1.050
3.527	3.524	2.853	3.502	2.838	3.502
FVC: 512 entries; 1 frequently accessed value					
8Kb DMC + .25 Kb FVC	16Kb DMC	16Kb DMC + .25 Kb FVC	32Kb DMC	32Kb DMC + .25 Kb FVC	64 Kb DMC
0.929	1.101	0.765	1.050	0.736	1.050
3.698	3.525	3.024	3.502	3.004	3.502

DMC: line size = 8 words = 32 bytes				
FVC: 512 entries; 7 frequently accessed values				
benchmark	16Kb DMC + 1.5Kb FVC	32Kb DMC	32Kb DMC + 1.5Kb FVC	64Kb DMC
	% misses	% misses	% misses	% misses
124.m88ksim	0.385	0.853	0.346	0.853
134.perl	2.685	3.829	2.668	3.829
FVC: 512 entries; 3 frequently accessed values				
	16Kb DMC + 1Kb FVC	32Kb DMC	32Kb DMC + 1Kb FVC	64Kb DMC
124.m88ksim	0.408	0.853	0.366	0.853
134.perl	2.851	3.829	2.834	3.829
FVC: 512 entries; 1 frequently accessed value				
	16Kb DMC + .5Kb FVC	32Kb DMC	32Kb DMC + .5Kb FVC	64Kb DMC
124.m88ksim	0.740	0.853	0.718	0.853
134.perl	3.021	3.829	2.999	3.829

DMC: line size = 16 words = 64 bytes	
FVC: 512 entries; 7 frequently accessed values	
32Kb DMC + 3Kb FVC	64Kb DMC
% misses	% misses
0.246	0.757
2.170	2.834
FVC: 512 entries; 3 frequently accessed values	
32Kb DMC + 2Kb FVC	64Kb DMC
0.267	0.757
2.336	2.834
FVC: 512 entries; 1 frequently accessed value	
32Kb DMC + 1Kb FVC	64Kb DMC
0.618	0.757
2.502	2.834

Figure 13: DMC + FVC vs larger DMC : miss rate comparison.

capacity misses because due to the data compression technique employed a small FVC has a significant capacity (a 1.5 Kbyte FVC can hold 4 K frequent values). Since set-associativity eliminates a number of conflict misses, the improvements achieved by FVC in a set-associative configuration can be expected to be lower.

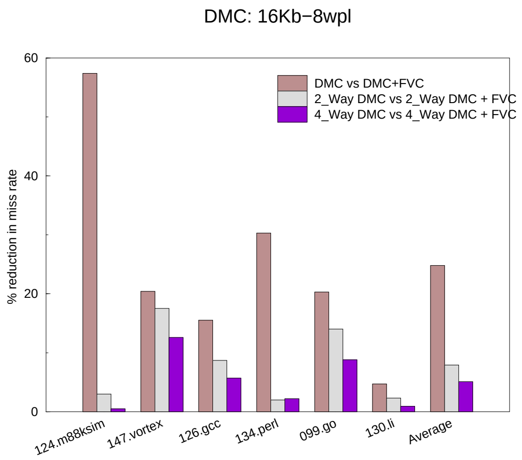

Figure 14: 2-way and 4-way set-associative FVCs (top 7 frequently accessed values).

We conducted an experiment in which we compared the performance improvements resulting from the use of FVC in a direct-mapped setting with those obtained in a 2-way and 4-way set-associative setting. The cache size used was 16 Kbytes and line size was 8 words per line. The results obtained are given in Figure 14. As we can see, in three of the benchmarks (`124.m88ksim`, `134.perl`, and `130.li`) the improvements for 2-way and 4-way set-associative caches are very small because most of the misses eliminated by FVC in the direct-mapped setting were conflict misses which are eliminated by incorporating associativity in DMC. For the other three benchmarks (`147.vortex`, `126.gcc`, and `134.go`) the miss rate reductions for 2-way and 4-way set-associative caches are significant because there were significant number of capacity misses in these benchmarks.

Victim cache vs FVC . A known approach for improving the performance of direct-mapped caches is the victim cache [4]. For small direct mapped caches, a fully associative victim cache of a few entries (eg., four entries) can greatly reduce the miss rate. The victim cache was designed to effectively hold cache lines that are evicted from a DMC and immediately refetched. This is achieved by saving cache lines evicted from DMC into VC . If a miss in DMC and a hit in VC occurs, the lines at respective cache entries are swapped. The VC is a fully associative structure and therefore it is important that its size be small.

We conducted two experiments to compare the performances of FVC and VC when used with a DMC of size 4Kb with a line size of 8 words. In the first experiment we compared the performances of equal sized victim and frequent value caches. Given a DMC of line size of eight words, the FVC requires a data cache field that is less than one word (24 bits or 3 bytes if 7 frequently accessed values are considered). In contrast, a line in VC contains eight words. A FVC with more entries than a VC will also contain more tags. In fact accounting for the tags, we found that a 128-entry FVC which exploits 7 frequently occurring values and a 16-entry VC take almost the same amount of space for a line size of 8 words. In the second experiment we compared the performances of VC and FVC with nearly same access times. The access time of a small 4-entry VC with 8 words per line is 9ns while the access time of a larger 512-entry FVC is 6ns leaving enough room for time spent on encoding and decoding of values in FVC. The access time of a small VC is comparable to a larger FVC because VC is fully associative and FVC is direct mapped.

From the results in Figure 15 we can see that in the first experiment VC outperforms the FVC while in the second experiment FVC outperforms the VC. From this experiment we can conclude that both VC and FVC are quite effective in improving the performances of small direct-mapped caches.

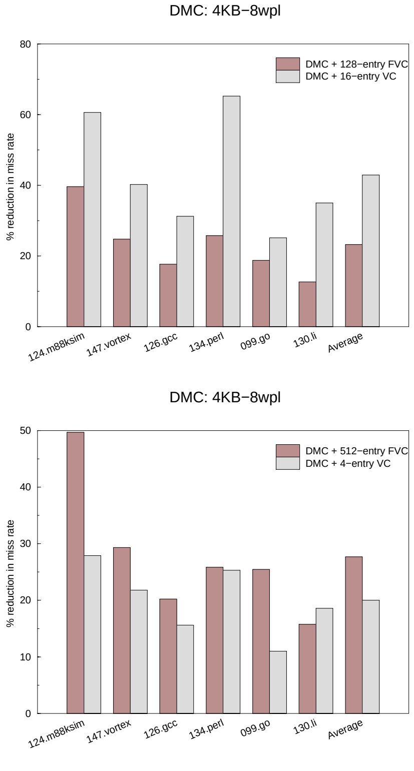

Figure 15: Fully associative VC vs direct mapped FVC (top 7 frequent values).

5. CONCLUSIONS

This paper makes two important contributions. The first contribution is the identification of the *frequent value locality* phenomenon. We have demonstrated that six out of eight SPECint95 programs exhibit frequent value locality according to which some values appear very frequently in memory locations and are involved in a large fraction of memory accesses. In these six programs ten distinct values occupy over 50% of all memory locations and on an average account for nearly 50 the frequently occurring and accessed values remain the same over the execution of the program and are distributed fairly evenly throughout the memory. This phenomenon suggested a simple data compression scheme which allowed us to achieve impressive data compression rates without sacrificing the random access to data values.

The second contribution of this paper is the design of the *frequent value cache* which clearly demonstrates that a value-centric treatment of data locations can lead to significant improvement in the overall performance of on-chip caches. In particular, we have demonstrated that augmenting a DMC with a small FVC is more cost effective solution than increasing the size of the DMC to obtain better performance. We believe that this phenomenon can be exploited in many creative ways to improve the performance of memory hierarchy. The FVC is merely one such example.

6. REFERENCES

[1] D. Brooks and M. Martonosi, “Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance,” *Proc. of the 5th International Symp. on High-Performance Computer Architecture*, Orlando, Florida, Jan. 1999.

[2] S.K. Debray, W. Evans, and R. Muth, “Compiler Techniques for Code Compaction,” *ACM Transactions on Programming Languages and Systems*, to appear.

[3] F. Gabbay and A. Mendelson, “Can Program Profiling Support Value Prediction?, ” *Proc. of the 30th ACM/IEEE International Symp. on Microarchitecture*, pages 270-280, Dec. 1997.

[4] N.P. Jouppi, “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,” *Proc. 17th Annual International Symp. on Computer Architecture*, Seattle, pages 364-373, 1990.

[5] N.P. Jouppi and S.J.E. Wilton, “An Enhanced Access and Cycle Time Model for On-Chip Caches,” Technical Report 93/5, *DEC WRL*, July 1994.

[6] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge, “Improving Code Density Using Compression Techniques,” *Proc. of the 30th Annual ACM/IEEE International Symp. on Microarchitecture*, pages 194-203, 1997.

[7] D. Kirovski, J. Kin, and W. H. Mangione-Smith, “Procedure Based Program Compression,” *Proc. of the 30th Annual ACM/IEEE International Symp. on Microarchitecture*, pages 204-217, 1997.

[8] M.H. Lipasti, C.B. Wilkerson and J.P. Shen, “Value Locality and Load Value Prediction,” *Proc. of the 7th International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 138-147, Oct. 1996.

[9] P. Ranganathan, S. Adve, and N. Jouppi, “Reconfigurable Caches and their Application to Media Processing,” *Proc. of the 27th Annual International Symp. on Computer Architecture*, Vancouver, British Columbia, Canada, June 2000.

[10] M. Stephenson, J. Babb, and S. Amarasinghe, “Bitwidth Analysis with Application to Silicon Compilation,” *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, Vancouver, British Columbia, Canada, June 2000.

[11] J. Yang, Y. Zhang, and R. Gupta, “Frequent Value Compression in Data Caches,” Technical Report, *Univ. of Arizona*, Dept. of CS, Tucson, AZ, June 2000.