Energy Efficient Frequent Value Data Cache Design *

Jun Yang [(1)]
Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA 92521

Rajiv Gupta
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Abstract

Recent work has shown that a small number of distinct frequently occurring *values often account for a large portion of memory accesses. In this paper we demonstrate how this* frequent value *phenomenon can be exploited in designing a cache that trades off performance with energy efficiency. We propose the design of the* Frequent Value Cache *(FVC) in which storing a frequent value requires few bits as they are stored in encoded form while all other values are stored in unencoded form using 32 bits. The data array is partitioned into two arrays such that if a* frequent value *is accessed only the first data array is accessed; otherwise an additional cycle is needed to access the second data array. Experiments with some of the* SPEC95 *benchmarks show that on an average a 64Kb/64-value FVC provides 28.8% reduction in L1 cache energy and 3.38% increase in execution time delay over a conventional 64Kb cache.*

1 Introduction

Due to the proliferation of mobile computing devices which operate on battery power, interest in energy efficient processor designs is growing. The on-chip cache memory is one significant source energy consumption to which many researchers have paid attention. Approaches for conserving cache energy include filtering mechanisms for reducing cache activity [7, 17], reconfiguring the cache according to the application needs [1, 14, 11], and compression techniques for reducing cache activity [4, 6]. To a significant extent these three approaches are orthogonal to each other.

We present a new approach to reducing dynamic energy consumed by a data cache which is complimentary to the above solutions. The design and evaluation of energy efficient L1 data cache, called the *frequent value cache* (FVC), which achieves energy savings through data encoding is described. Recently we have shown that a small number of distinct *frequently occurring* values often occupy a large portion of program memory and therefore account for a large portion of memory accesses [19, 15]. We exploit this *frequent value* phenomenon in designing a cache that trades off performance with energy efficiency.

In the FVC, frequent values are represented by fewer than 32 bits as they are stored in encoded form while all other values are stored in unencoded form using all 32 bits of a word. The data array is partitioned into two arrays such that if a frequent value is accessed only the first data array is accessed, while for nonfrequent values both data arrays must be accessed. Since frequent values are encountered quite often, this approach greatly reduces the energy consumed by the data cache. The reduction in energy is achieved at the cost of an additional cycle needed to access nonfrequent values. Thus, FVC represents tradeoff between lower dynamic energy consumption for frequent value accesses and higher access times for nonfrequent value accesses.

Detailed access time and energy models for FVC are developed by modifying XCACTI [10]. Studies demonstrate that for a wide range of configurations for set-associative data caches—including varying associativity, cache size, line size, and number of frequent values—the FVC compares favorably with a conventional data cache. We show that the time spent on encoding and decoding of values does not impact the cache access time for associative caches. While the reduction in energy consumed by a frequent value access ranges from 34% to 84%, the increase in the energy consumed by a nonfrequent access is less than 1% for a read and less than 5% for a write. Finally our experiments with some of the SPEC95 benchmarks show that on an average a 64Kb/64-value FVC provides 28.8% reduction in L1 cache energy and 3.38% increase in execution time delay over a conventional 64Kb cache.

The remainder of the paper we first present the design and analysis of the FVC in Section 2. In Section 3 we discuss how frequent values can be identified. Section 4 contains experimental results and Section 5 describes additional related work.

*Supported by DARPA award no. F29601-00-1-0183 and National Science Foundation grants CCR-0208756, CCR-0220334, CCR-0105355, and EIA-0080123 to the University of Arizona. (1) This work was done while Jun Yang was at Univ. of Arizona.

2 Frequent Value Cache

From the perspective of the frequent value cache, data values are divided into two categories: a small number of *frequent values*, say n, that typically ranges from 4 values to 128 values and all remaining values that are referred to as *nonfrequent values*. The frequent values are stored in encoded form and therefore can be represented in $log_2 n$ number of bits, which ranges from 2 bits for 4 frequent values to 7 bits for 128 frequent values. The nonfrequent values are stored in unencoded form in 32 bit words. The set of frequent values remains fixed for a given program run.

The cache data array is partitioned into two data arrays as shown in Figure 1. The *low-bit array* contains the lower order $log_2 n$ bits of each word and the *high-order array* contains the remaining $32 - log_2 n$ bits. Frequent values are stored in encoded form in the low-bit array while nonfrequent values are split into the lower order $log_2 n$ bits and the higher order $32 - log_2 n$ bits using the space in both arrays. To distinguish between a code for a frequent value and a trailing part for a nonfrequent value in the low-bit array, an additional flag bit is needed corresponding to each word in a cache line. The extra bit was stored along with every word in the low-bit array so that the word width becomes $log_2 n + 1$.

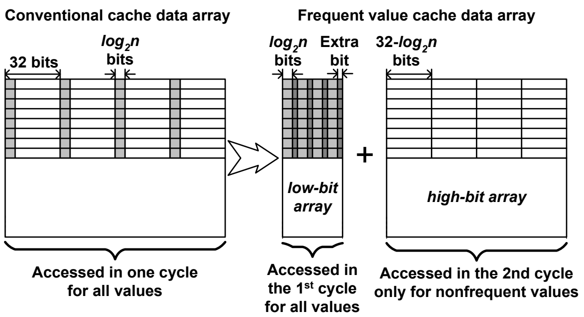

Figure 1. An example of partitioning the data array of 32 bits per word (bpw), 4 words per line (wpl) into two arrays of $log_2 n + 1$ bpw, 4 wpl and $32 - log_2 n$ bpw, 4 wpl each.

Accessing the FVC. When reading a word from the cache, initially we simply read from the low-bit array. Since every word read out contains a flag bit, we examine it to determine what comes next. If the flag is clear it means that the desired word is in unencoded form so the remaining bits should be read out from the high-bit array to form the original value. On the other hand, if the flag is set it means that the desired word is a frequent value and it is stored in encoded form. In this case we proceed to decode the value. Since the access to the high-bit array is avoided, cache activity is reduced.

Since the retrieval of $log_2 n$ bits from the low-bit array and that of $32 - log_2 n$ bits from the high-bit data array is serialized, it takes longer to read a nonfrequent value from the FVC than it would have taken to read the same value from a conventional data cache. Let us assume that upon a hit it takes a single cycle to read a value from a conventional data cache. In contrast, for FVC, in the first cycle the $log_2 n$ bits from the low-bit array are accessed and if the value is a nonfrequent one, in the second cycle the remaining bits from the high-bit data array are accessed. Thus, reading nonfrequent values takes an extra cycle.

A write to the FVC is performed as follows. Before a value is written, it is first encoded through an encoder. If encoding is successful, it means that the value is a frequent value and thus $log_2 n$ bit code is stored in the low-bit array and the flag bit is set. In this case, accessing the high-bit array is avoided. If the encoding fails, the value to be written is a nonfrequent value and thus both low-bit and high-bit data arrays are accessed as well as the flag bit is cleared.

The FVC design. We have developed a detailed FVC design shown in Figure 2. Let us further discuss in detail how the read/write operations as well as corresponding decode/encode operations are performed.

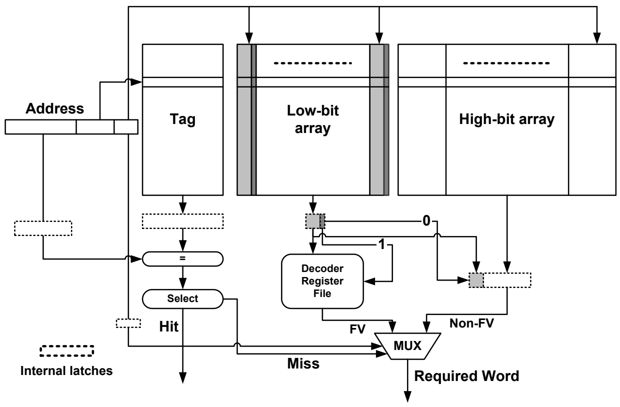

Figure 2. The FVC design.

During a *read operation* when the flag bit is set, the $log_2 n$ bit code is decoded. The decoder is in effect a multiported register file that contains frequent values. This *decoder register file* is shared among the multiple cache ways. The code is used to index the register file to retrieve the corresponding 32-bit frequent value. Reading a frequent value is done after the 32-bits flow through the output multiplexer. However, if the flag bit is clear indicating a nonfrequent value, it turns off the decoder letting the partial data value flow to an internal latch where the remaining part of the value is filled from reading the high-bit array in the second cycle. The full value is obtained by concatenating the two parts completing an infrequent value read.

In a traditional cache implementation the entire line is first read out from the data array, then the desired word within the line is selected at the time it reaches the output multiplexer. If the same scheme is used in the FVC design, the decoding of frequent values cannot begin until the required word is selected out which is the very end of a cache access. Consequently decoding will increase the cache access time which is not desirable. Therefore we adopt the subbanking scheme proposed by [5] in which the subbank containing the target word can be read independently. The width of each subbank is the physical word width and each subbank can be activated independently. This design facilitates our FVC implementation in that the decoding for frequent values can begin immediately after accessing the low-bit array since only the word at the desired location is read out instead of the whole line. The actual measurements of FVC access time can be found in section 2.1 and subbanking scheme is used in our experiments as the baseline configuration.

During a *write operation* the value must be encoded. The encoding of a frequent value to be written is carried out before the cache access since the value to be written may be known as early as the decode/operand fetch stage. The encoding hardware for write accesses is shown in Figure 3. It is a CAM that can match an incoming frequent value and output its location in binary form. The *CAM Array* shown in the figure stores the most often encountered 16 frequent values and the RAM stores the binary representation (ID) of each entry corresponding to the *CAM Array*, i.e. for 16 frequent values we need 4 bits each.

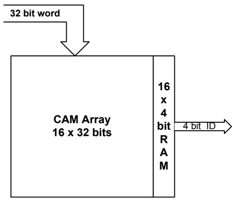

Figure 3. 16 Value FV CAM.

The frequent value encoder shown in Figure 3 is part of a slightly larger hardware that contains extra control information that is responsible for *finding frequent values*. During a given program run, frequent values are first found through this hardware by monitoring the values accessed for certain amount of time following which the values are fixed and monitoring is stopped. For remainder of the program's execution, these fixed frequent values are stored in encoded form. Therefore the values captured by the frequent value finder are sent to the *decoder register file* at this point. When frequent value encoding/decoding is being carried out the contents of the *decoder register file* and the *encoding CAM* are the same. The details of the frequent value finder are discussed in section 3.

We have modified XCACTI [10] to incorporate a model of the above FVC design. The XCACTI tool adds the energy model for cache writes, write-backs and line-fills. These are what we need in order to correctly measure the FVC energy. The baseline cache uses the subbanking technique presented in [5]. The major change in XCACTI to accommodate subbanking is the organization parameter N_{dwl} which indicates how many times the data array has been split with vertical cut lines. In our model N_{dwl} is fixed to the number of words per line. This way the address decoder to the data array will automatically pick the right subbank to drive. For FVC design, accessing a frequent value differs from a baseline access in that the word width is narrowed down to code size plus one. Accessing an infrequent value increases each word width by one which will affect the delay in decoder, wordline, bitline etc. since the number of bit columns, the wire lengths connecting different lines in a set and other parameters changed. For the *frequent value register file*, we extracted the data array access model from XCACTI and modified it so that it complies with register files. The method we used is similar to the one used in [2]. Next we present an analysis of access time and energy for read and write accesses for FVC under 0.18 μm technology.

2.1 FVC Access Time

Let us study the timing behavior of an FVC and a conventional cache in greater detail and then see how their access times compare. In a conventional cache design the decoder drives both the tag array and the data array in parallel. Tag array does the tag matching and the data array reads out the words from intended locations. Typically, the tag matching takes longer than the data read. If the tag matching fails, the results of the data read are discarded; otherwise the data word is sent to the CPU. In case of FVC, while the tag is compared against the reference address tag, the data is read out of the low-bit data array. The single bit indicating whether the value is frequent or nonfrequent is also read and if the bit indicates that we have an encoded frequent value, the decoding of the value is then initiated. Since the read is completed before the results of the tag match are known, the decoding is carried out in parallel with tag matching.

We would like to ensure that the access time of the FVC is no greater than the access time of the baseline cache. Since we have not changed the tag path, the tag matching time is the same for baseline and FVC caches. Therefore, for the access time of the FVC to be equal to that of the conventional cache, the sum of the times spent by the FVC on retrieving a frequent value from the low-bit data array ($T_{data\_path\_delay}$ without output multiplexer) and the decoding of the frequent value (T_{decode}) should be no greater than the time spent on tag matching ($T_{tag\_path\_delay}$), i.e.:

$$T_{data\_path\_delay} + T_{decode} \leq T_{tag\_path\_delay}.$$

Figure 4(a) shows the timing behavior of the FVC for frequent and nonfrequent value reads assuming that the FVC satisfies the above constraint. It should be noted that no timing constraint arises due to encoding of values during write operations because the value being written is encoded before the cache access. Figure 4(b) depicts the situation for nonfrequent value reads. Since accessing the high-bit array is serialized with the low-bit array access, the total time in reading data array can not be less than the tag matching time. Therefore it is carried in the following cycle.

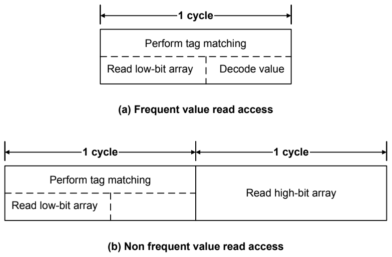

Figure 4. Read access operation.

Whether or not the above timing constraint is satisfied is determined by the configuration of the FVC. The T_{decode} is a function of decoder size which in turn depends upon the number of frequent values we use—the larger the number the longer the decoding time. The difference between the $T_{tag\_path\_delay}$ and the $T_{data\_path\_delay}$ is a function of cache size and associativity.

In Figure 5 we show the increase in access time, if any, that is caused due to the need to perform a decode operation upon a frequent value access. Recall that the access time is the critical path delay in the cache. The number above each column is the tag matching time and the increase in access time is given in each entry. In this study, we limit the variation of the cache organization parameters so that the cache data array is not further chopped up into subarrays since we have already divided it vertically into subarrays of one word width. As we can see, for a wide range of configurations shown in bold there is no increase. This is because the decode time is quite small varying from 0.41 ns for 2 bit decoding to 0.678 ns for 7 bit decoding of frequent values. Since the tag path delay is greater than the data path delay, often there is enough time left over to carry out decoding. In many other cases while there is an increase in access time, the increase is very small. Finally, for direct mapped caches typically FVC design is not attractive as the increase in access time is significant. In summary, FVC is expected to perform well for configurations used in power efficient processors—high associativity and modest size.

2.2 FVC Dynamic Energy Model

The energy estimation model is separated into two parts. In the first part we compute energy estimates on a *per cache access* basis based upon the detailed characteristics of the cache design shown in Figure 2 and the cache configuration. The second part of the model estimates the *total cache energy* expended during a program run. Therefore it depends upon runtime behavior of the program.

Per access energy. The computation of energy per access (EPA) for the FVC are presented in Figure 6. There are three main energy components for the conventional cache: energy consumed by the *data array*, energy consumed by the *tag array*, and energy needed to *drive the output multiplexer*. The energy consumed by the tag array is the same for both the baseline cache and the FVC, when we compare caches with same associativity and capacity. Moreover, this energy must be spent for both frequent and nonfrequent value accesses. The same is true for the energy that is spent on driving the output multiplexer as in all cases output is 32 bits. The two main differences in energy computations are described next.

Access times in nanoseconds.

8 Kbyte Cache

Ways →	1		2		4		8	
L →	8b	16b	8b	16b	8b	16b	8b	16b
C (bits) ↓	4.38	2.43	2.85	2.26	2.36	2.11	2.22	1.99
2	+0	0.01	+0	+0	+0	+0	+0	+0
3	+.01	+.08	+0	+0	+0	+0	+0	+0
4	+.07	+.15	+0	+0	+0	+0	+0	+0
5	+.14	+.21	+0	+0	+0	+0	+0	+0
6	+.24	+.31	+.03	+.06	+0	+0	+0	+0
7	+.32	+.40	+.11	+.14	+.04	+.04	+0	+0

16 Kbyte Cache

Ways →	1		2		4		8	
L →	8b	16b	8b	16b	8b	16b	8b	16b
C (bits) ↓	8.89	4.37	4.80	2.86	2.94	2.37	2.52	2.28
2	+0	+.02	+0	+0	+0	+0	+0	+0
3	+.01	+.09	+0	+0	+0	+0	+0	+0
4	+.09	+.16	+0	+0	+0	+0	+0	+0
5	+.15	+.22	+0	+0	+0	+0	+0	+0
6	+.25	+.32	+.02	+.09	+.01	+.02	+0	+0
7	+.33	+.41	+.10	+.17	+.09	+.10	+0	+0

32 Kbyte Cache

Ways →	1		2		4		8	
L →	16b	32b	16b	32b	16b	32b	16b	32b
C (bits) ↓	8.89	4.37	4.79	2.87	2.96	2.43	2.56	2.41
2	+.01	+.09	+0	+0	+0	+0	+0	+0
3	+.08	+.16	+0	+0	+0	+0	+0	+0
4	+.16	+.24	+0	+0	+0	+0	+0	+0
5	+.22	+.30	+.01	+.06	+0	+0	+0	+0
6	+.32	+.40	+.11	+.16	+.06	+.04	+0	+0
7	+.41	+.49	+.19	+.25	+.14	+.13	+0	+0

64 Kbyte Cache

Ways →	1		2		4		8	
L →	16b	32b	16b	32b	16b	32b	16b	32b
C (bits) ↓	22.74	8.87	9.29	4.79	4.89	3.00	3.13	2.67
2	+.02	+.11	+0	+0	+0	+0	+0	+0
3	+.09	+.18	+0	+0	+0	+0	+0	+0
4	+.17	+.26	+0	+.02	+0	+0	+0	+0
5	+.23	+.32	+.03	+.09	+0	+.01	+0	+0
6	+.34	+.42	+.13	+.19	+.09	+.12	+0	+0
7	+.42	+.51	+.22	+.28	+.18	+.21	+.06	+0

Figure 5. Finding configurations with 1-cyle frequent value access and decode.

The first difference in energy computations arises in computing the data array energy due to the word line length and number of bitlines driven. In case of the baseline cache, the wordline length, wl, is simply computed from the bits per word and the associativity. In case of a frequent value access in the FVC, wl changes to wl', where wl' is less than wl because the high-bit data array ($32 - log_2 n$ bits per word) is not accessed and it also offsets the energy spent on the extra flag bit. Similarly for a nonfrequent value ac-

Energy spent on a single access on a hit in a baseline cache

$$
\begin{array}{|lcl|}
\hline
EPA_{cache} & = & EPA_{data}(\mathbf{wl}) + EPA_{tag} + EPA_{output}(\mathbf{wl}) \\
\hline
EPA_{data} & = & EPA_{d-decoder}(\mathbf{wl}) + EPA_{d-wordline}(\mathbf{wl}) + EPA_{d-bitline}(\mathbf{wl}) + EPA_{d-sense-amp}(\mathbf{wl}) + \\
 & & EPA_{d-sense-ext-driver}(\mathbf{wl}) \\
\hline
EPA_{tag} & = & EPA_{t-decoder} + EPA_{t-wordline} + EPA_{t-bitline} + EPA_{t-sense-amp} + EPA_{compare} + \\
 & & EPA_{valid-driver} + EPA_{drive-mux} + EPA_{select-block} \\
\hline
\multicolumn{3}{|l|}{where,\ EPA_{data}(\mathbf{wl})\ means\ EPA_{data}\ increases\ with\ \mathbf{wl}} \\
\hline
\mathbf{wl} & = & word\ line\ wire\ length\ in\ bits\ =\ 32 \times Associativity \\
\hline
\end{array}
$$

Energy spent on a single **frequent value** access on a **hit** in FVC

$$
\begin{array}{|lcl|lcl|}
\hline
EPA^{Read}_{fv} & = & EPA_{fv} + E_{fv-decode} & EPA^{Write}_{fv} & = & EPA_{fv} + E_{fv-encode} \\
\hline
EPA_{fv} & = & EPA^{fv}_{data}(\mathbf{wl}') + EPA_{tag} + EPA_{output}(\mathbf{wl}) & & & \\
\hline
EPA^{fv}_{data}(\mathbf{wl}') & = & \multicolumn{4}{l|}{EPA_{d-decoder}(\mathbf{wl}') + EPA_{d-wordline}(\mathbf{wl}') + EPA_{d-bitline}(\mathbf{wl}') + EPA_{d-sense-amp}(\mathbf{wl}') +} \\
 & & \multicolumn{4}{l|}{EPA_{d-sense-ext-driver}(\mathbf{wl}')} \\
\hline
\multicolumn{6}{|l|}{where,\ \mathbf{wl}' = FVC\ low-bit\ array\ word\ line\ wire\ length\ in\ bits\ =\ (log_2 n + 1) \times Associativity < \mathbf{wl}} \\
\hline
\end{array}
$$

Energy spent on a single **nonfrequent value** access on a **hit** in FVC

$$
\begin{array}{|lcl|lcl|}
\hline
EPA^{Read}_{nfv} & = & EPA_{nfv} & EPA^{Write}_{nfv} & = & EPA_{nfv} + E_{fv-encode} \\
\hline
EPA_{nfv} & = & EPA^{nfv}_{data}(\mathbf{wl}'') + EPA_{tag} + EPA_{output}(\mathbf{wl}) & & & \\
\hline
where, & \multicolumn{5}{l|}{(\mathbf{wl}'')\ =\ Total\ FVC\ word\ line\ wire\ length\ in\ bits\ =\ 33 \times Associativity > \mathbf{wl}} \\
\hline
\end{array}
$$

Figure 6. Per access energy model.

cess, wl changes to wl'', where wl'' is greater than wl since the flag bits need to be accessed irrespective of the frequent property of the value. Similar to the reduction in word line length, the bitlines are reduced from 32 to $log_2 n + 1$ bits per word access. For example, if the codes for frequent values are only 4 bits, the amount of decrease in driving the bitlines is $(32 - 4 - 1)/32 = 84\%$. For nonfrequent values, there is an extra bitline which is a $1/32 = 3\%$ increase.

The second difference in energy computation arises due to the decoding and encoding operations that must be carried out during read and write operations. Decoding is associated with every *frequent value read* access and each decoding is an access to the *decoder register file*. Reading nonfrequent values does not trigger decoding since it is blocked by the flag bit. Encoding is associated with *every* write access since the information of the value being encoded is not known a prior. The energy spent in encoding, E_{fv_encode}, is the energy of a CAM access which differs greatly from the decoding energy, E_{fv_decode}, and we will compute them separately.

It is clear now that there is savings in energy when a frequent value is accessed while there is an increase in energy consumed when a nonfrequent value is accessed. However, the savings are much greater than the increase. We also analyzed the per access energy characteristics of several configurations. In Figure 7 the percentage reductions in energy consumed on a frequent value read access are given (L is the line size and C is the number of bits for encoding frequent values). As we can see from the results, the reduction per access ranges from 34% to 84%. For writes, it is a little bit lower. The reductions are higher for greater degree of associativity.

8 Kbyte Cache

Ways →	1		2		4		8	
L →	8b	16b	8b	16b	8b	16b	8b	16b
C (bits) ↓								
2	42.6	55.5	60.3	66.9	70.6	74.9	78.0	80.1
3	41.9	54.3	58.3	64.7	68.2	72.5	75.4	77.5
4	39.5	51.5	56.1	62.2	65.7	69.8	72.7	74.8
5	38.4	49.6	53.6	59.4	63.0	66.9	70.0	71.9
6	35.6	46.1	50.9	56.3	60.1	63.9	67.1	68.9
7	34.1	43.6	47.9	52.9	57.0	60.6	64.0	65.7

16 Kbyte Cache

Ways →	1		2		4		8	
L →	8b	16b	8b	16b	8b	16b	8b	16b
C (bits) ↓								
2	43.8	52.4	56.9	67.4	69.2	75.0	76.7	80.0
3	43.1	51.3	55.0	65.1	66.9	72.6	74.2	77.3
4	40.8	48.8	53.0	62.7	64.5	70.0	71.6	74.6
5	39.9	47.3	50.8	60.1	61.9	67.2	68.9	71.8
6	37.4	44.5	48.6	57.3	59.2	64.3	66.1	68.9
7	36.2	42.6	46.1	54.2	56.4	61.2	63.2	65.9

32 Kbyte Cache

Ways →	1		2		4		8	
L →	16b	32b	16b	32b	16b	32b	16b	32b
C (bits) ↓								
2	53.4	63.8	65.0	75.1	74.9	79.6	79.8	82.6
3	52.3	62.2	62.9	72.6	72.4	77.0	77.2	80.0
4	49.8	59.5	60.6	70.0	69.9	74.3	74.5	77.3
5	48.5	57.6	58.2	67.2	67.2	71.5	71.8	74.5
6	45.8	54.6	55.7	64.3	64.4	68.5	69.0	71.6
7	44.3	52.4	53.1	61.3	61.5	65.5	66.1	68.6

64 Kbyte Cache

Ways →	1		2		4		8	
L →	16b	32b	16b	32b	16b	32b	16b	32b
C (bits) ↓								
2	55.4	65.0	66.2	74.1	73.9	81.2	80.1	84.0
3	54.3	63.3	64.0	71.7	71.5	78.5	77.5	81.4
4	51.7	60.7	61.7	69.1	68.9	75.8	74.9	78.7
5	50.5	58.8	59.4	66.5	66.4	73.0	72.2	76.0
6	47.8	56.0	57.1	63.8	63.7	70.1	69.4	73.2
7	46.5	54.0	54.6	61.0	61.0	67.2	66.6	70.3

Figure 7. Percentage of energy savings for frequent value accesses.

As expected, the reduction in per access energy is greater when fewer number of bits are used to encode frequent values. However, this does not imply that we should use the least number of bits to encode frequent values. When the cumulative cache energy is considered, this trend need not hold. This is because as we increase the number of bits we can encode greater number of values and therefore the fraction of total accesses that are frequent value accesses also increases.

The increase in per access energy consumed for a nonfrequent value access is small in comparison to the reduction achieved for a frequent value. We computed these increases separately for read and write operations. This is because a write is more expensive as it requires an associative search in the CAM encoder. We found that the increase in energy for a *nonfrequent value read* ranges from 0.0005% to 0.0678% for the configurations considered. This increase is mainly due to the access to the extra flag bit that is required in FVC. In contrast, the increase in energy used by a nonfrequent value write ranges from 0.776% to 4.679%. As we can see, these increases are small in comparison to the energy savings that result during frequent value accesses.

Total cache energy. Given the per access energy costs computed in Figure 6, we can now derive equations shown in Figure 8 for computing the total energy consumed by the cache in a given program run. These computations are based upon the number of read and write accesses which are hits, number of references that are misses, and the memory update policy. We assume that we have a writeback policy and a single dirty bit is maintained for each cache line to decide when the writeback is needed.

The total energy is divided into three parts: energy consumed by read hits (E_{Rhit}), write hits (E_{Whit}), and misses (E_{RWmiss}). Since we have already computed the energy per read/write access on a hit for a frequent/nonfrequent value (EPA_{fv}^{Read}, EPA_{fv}^{Write}, EPA_{nfv}^{Read}, EPA_{nfv}^{Write}), the computation of total read/write energy on hits for a program run can be easily computed by measuring the total number of read/write hits to frequent/nonfrequent values (H_{fv}^{Read}, H_{nfv}^{Read}, H_{fv}^{Write}, and H_{nfv}^{Write}) as shown in Figure 8.

The energy consumed by misses consists of two parts: energy for performing the write back operation ($E_{writeback}$) and energy consumed by line fill of the new line fetched from the L2 cache ($E_{linefill}$). The writeback energy is computed separately for dirty lines and non-dirty lines. If the cache line is not dirty, we still pay some price for reading it from the cache. This is because the reading of the low-bit array is carried out in parallel with tag matching. Therefore even if there is a miss, and the line is not dirty, we have already spent EPA_{fv} energy in reading the portion of a word line. If the line is dirty, then we read the entire cache line, including the portion in high bit array, and expend EPA_{nfv} energy.

Before performing the writeback operation we must decode the frequent values. By measuring the number of misses of dirty lines ($M_{dr-lines}$), misses of non-dirty lines ($M_{ndr-lines}$), and number of frequent values in dirty lines ($DR_{fv-values}$), we can compute the total energy expended by the writeback operations performed in the program run. We assume that this process does not take extra time than usual since preparing for writeback and reading new lines from lower level memory is usually carried out in parallel. The lower level memory access time is a lot longer than the higher level memory. Therefore there is enough time to prepare the lines for writing back. The energy for writing in the new line is simply the cost of updating the entire cache line (EPA_{nfv}) and encoding the values in the line. Note that even if a value is a nonfrequent value, energy is spent on the encoding operation because the way we determine that a value is nonfrequent is by attempting to encode it. Note that line fill does not take longer time than usual since if it was a read operation that initiated the line fill, the referenced value is passed directly to the CPU whenever it is read out from the lower level memory. Performing encoding is done only afterwords so the process does not affect program speed. On the other hand, if the line fill is initiated by a write operation, the instruction finishes execution once it has deposited its value into the cache. The actions inside the cache usually do not affect program speed unless there are back to back cache accesses.

3 Finding Frequent Values

A hardware FV Finder is provided to monitor the values in initial stages of program execution. the frequent values found using limited monitoring are used by FVC for the remainder of the program execution. Next the hardware FV Finder design is introduced and the impact of monitoring time on the quality of frequent values found is studied.

3.1 Frequent Value Finder

The Design.A small table of frequent values is used by the finder. To find the top n values, a table of $2n$ entries is used with each entry having a *value* field and a *counter* field. The *value* field stores the data value encountered and the *counter* field contains an c bit saturating counter. Each time a data value is involved in an access by the CPU, the table is updated follows. If the value is present in entry i, its counter is incremented by one. When the counter saturates, the entry i is *swapped* with entry $i-1$. Swapping causes frequent values to percolate to the top part of the table.

When a new value is encountered, and there is no free entry in the table, a victim entry needs to be selected to free up the space. We select an entry from the bottom half of the table with the smallest *counter* since the bottom half contains less frequent values. When the counter field is short,

$E_{fvc} = E_{Rhit} + E_{Whit} + E_{RWmiss}$	$E_{Rhit} = \underbrace{H_{fv}^{Read} \times EPA_{fv}^{Read}}_{frequent\ value\ read} + \underbrace{H_{nfv}^{Read} \times EPA_{nfv}^{Read}}_{nonfrequent\ value\ read}$
$E_{Whit} = \underbrace{H_{fv}^{Write} \times EPA_{fv}^{Write}}_{frequent\ value\ write} + \underbrace{H_{nfv}^{Write} \times EPA_{nfv}^{Write}}_{nonfrequent\ value\ write}$	$E_{RWmiss} = E_{writeback} + E_{writein}$
$E_{writeback} = \underbrace{M_{ndr-lines} \times EPA_{fv}}_{cache\ line\ not\ dirty} + \underbrace{M_{dr-lines} \times EPA_{nfv} + DR_{fv-values} \times E_{fv-decode}}_{cache\ line\ dirty}$	
$E_{writein} = (wpl \times E_{fv-encode} + EPA_{nfv}) \times (M_{dr-lines} + M_{ndr-lines})$	

Figure 8. Total energy consumed by FVC.

selecting smallest counter is not expensive to implement either. We have experimented on the appropriate number of bits used in *counter* and found that the performance of a 2-bit counter is satisfactory. Therefore, selecting the smallest counter could be implemented as a *priority multiplexer*—selecting from the four states, $\{0,\ 1,\ 2,\ 3\}$, in descending priority. When there are multiple qualifying entries, we simply pick an arbitrary one. After certain amount of monitoring time, the frequent values are obtained from the top half of the table.

Our method is inspired by the software *value profiling* technique of [3]. However, the algorithm in [3] does not use *swapping*. It simply maintains frequency counts for values in the table and periodically clears half of the table to allow new values to enter into the table. The values are sorted according to their associated frequency counts and half of the values with counts lower than the other half are removed. The sorting operation makes this existing technique unsuitable for hardware implementation. Our algorithm does not require sorting; instead it uses *swapping* to approximate the effect of sorting. The swapping process approximately sorts the list such that bottom half contains less frequently seen values. In [15] we show that swapping is nearly as good as sorting for finding frequent values.

Combining Finder with Encoder. Due to the nature of finding frequent values, the FV finder is implemented as a CAM. We have mentioned in section 2 that this CAM can be combined with the encoder CAM. The complete hardware is used for different functionalities as follows. During the early period of program execution it acts as a *FV finder*. Later the monitoring stops and the hardware acts as a *frequent value encoder*. This is feasible since when the FV finder identified the set of frequent values on the top half of the table, the lower half can be shut off and the top half can continue as the encoder (see Figure 9). The encoder needs the content of the *FV finder* and there is no need in replicating storage for frequent values. However, it is necessary to pass the captured frequent values to the decoder shown in Figure 2. Decoder initialization is a one time activity and can be done in a few cycles.

Transition in cache behavior. We do not intend to sweep through the cache to encode the FV as soon as monitoring

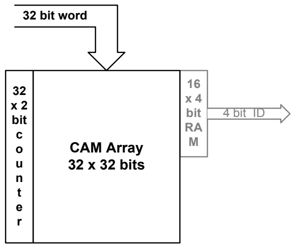

Figure 9. 32-entry finder and 16-FV encoder.

is turned off since this activity is too costly. Instead, after the FV finder is turned off, the cache will gradually turn itself into an FVC. The frequent values are recognized and encoded during write and cache line-fill operations. Later read operations benefit from the encodings.

3.2 Study of Monitoring Time

To study the effectiveness of the FV finder, we ran it on 11 out of 18 `SPEC95` benchmarks using reference inputs. We varied the duration of monitoring from millions to billions of instructions. The collected frequent value sets are compared with the *ideal set* which is obtained by software monitoring with unlimited resources through complete program run. The metric we use for comparison is the percentage of cache accesses involving the captured frequent values. We varied the number of frequent values used from 1 to 128. The results are presented in Figure 10.

In these figures, the legend shows the monitoring duration of the FV finder in terms of the number of instructions. In parenthesis the monitoring duration is given in terms of the percentage of the total executed instructions. The highest curve represents the ideal one so if most of the curves are close to that it means that the FV finder is very effective. From the graphs there are 8 out of the 11 tested benchmarks belong to this category. For these benchmarks the degree of monitoring has little impact on the quality of frequent value set identified. This is because we have found that for these programs frequent values can be divided into two categories: a small number of extremely frequent values seen throughout the program's execution and a much larger set of values that are encountered nearly equally frequently. A very small amount of monitoring is needed to identify val-

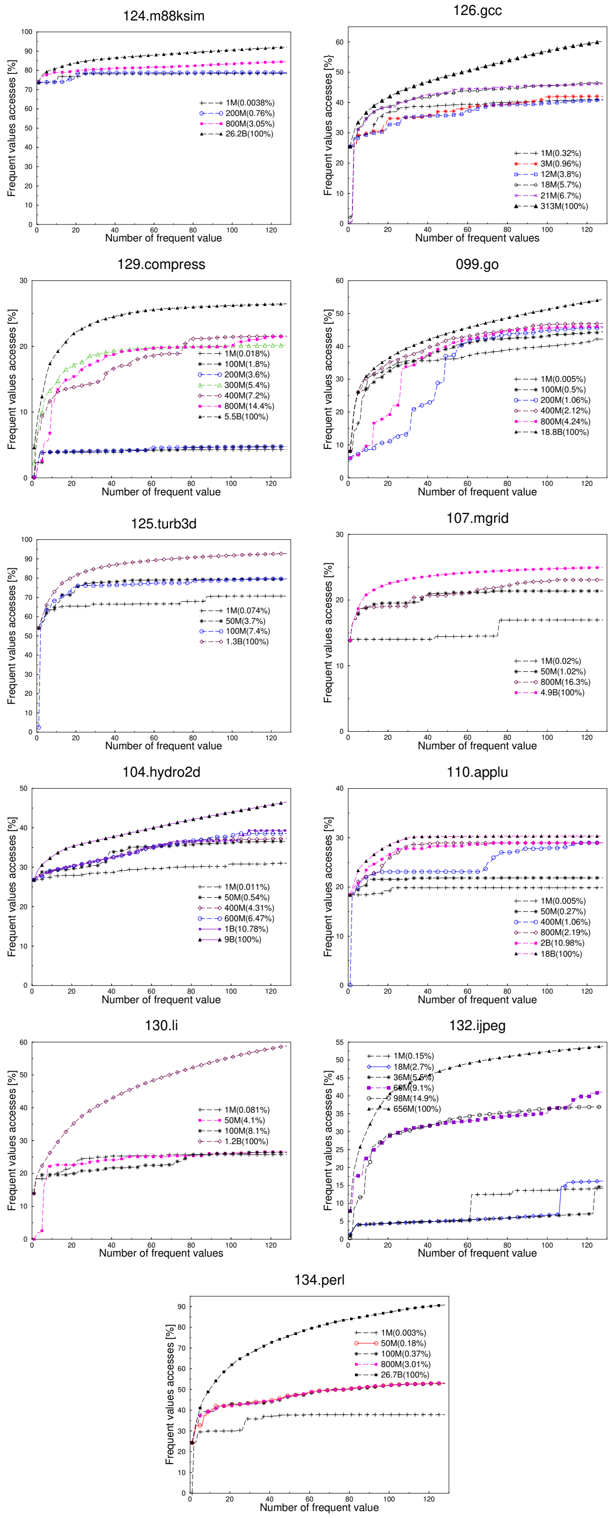

Figure 10. Comparing with frequent value set identified by the FV finder with an ideal set.

ues in the first category. A small amount of variation is due to the difference in the subset of values identified from the second category.

For the remaining three benchmarks—`130.li`, `132.ijpeg and 134.perl`, the results from the FV finder differ from the ideal result greatly. To find the cause behind this observation we ran an additional experiment in which we ran the FV finder allowing unlimited number of entries so no value is ever ejected from the FV table. In Figure 11 we see that for `130.li` and `134.perl`, with unlimited resources the top frequent values do appear as early as first 5% of program. Therefore, the inaccuracy in Figure 10 is due to the small number of entries and the replacement policy used by the FV finder. On the other hand, Figure 11 also shows that for `132.ijpeg`, even with unlimited entries, the frequent values do not become stable after about 15% of program run. For this case, the FV finder only needs to run longer to capture the right set of the frequent values as some of these values may show up later in the program execution.

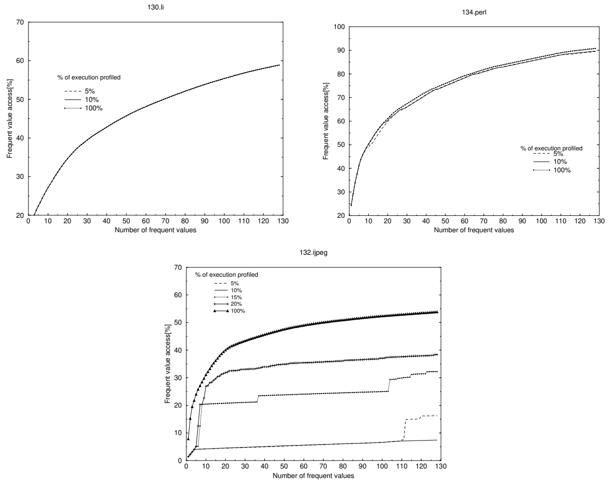

Figure 11. Comparing frequent value set identified using software method.

Our study shows that when the FV finder is effective, it needs to run for a short duration. Thus, in future experiments we run FV finder for the first 5% of memory accesses.

4 Experiments

Experimental setup. Low power processors being developed at present (e.g., StrongArm), have simple in-order pipelines and reasonably small associative caches. The simplicity of these designs is the key to achieving low power. Thus we decided to evaluate the FVC in the above context.

Our baseline processor supports a five stage in-order pipeline with the MIPS ISA. The simulator is based upon the system described in [8] into which we have integrated

XCACTI 2.0 with modifications necessary for FVC. The XCACTI 2.0 is also modified to model energy on every write, energy in write back and line fill operation so that the formulas in total energy model in Figure 8 are accurately applied. All the benchmarks are compiled with *gcc* version 2.7.2 for a MIPS compatible processor using optimization level *-O3*. The data cache parameters used in this study are given below in Figure 12. In addition, the L1 data cache latency is 1 cycle for the baseline and 1-2 cycle for the FVC corresponding to frequent-nonfrequent value accesses. The number of frequent values used are 16, 32 and 64. The cache sizes used vary from small (8 Kbyte) to modest (64 Kbyte). For these L1 cache configurations of FVC there is no increase in access times as the decoding of frequent values can be carried out according the timing constraints presented earlier.

	Level	Size (KB)	Line Size (B)	Ways
Data Cache	L1	8	16	4
		64	32	8
	L2	128	64	8

Figure 12. Configurations used.

We ran the configurations for the same set of benchmarks in Section 3 using reference inputs. During the first 5% of the memory accesses the cache acts as a traditional cache and the FV finder catches the frequent values. After that, the cache acts as a FVC. We observe that the FV finder performs better than the study whose results were presented in the preceding section because it is now being run for longer duration of time.

Experimental results. Figure 13 plots the percent of frequent value hits, including reading and writing of frequent values to the cache, in the 95% of the memory accesses performed after frequent values have been found. Bigger caches have more hits therefore a higher FV hit ratio. We found that the results for `130.li` and `134.perl` improved by about 10% which means monitoring for longer time is beneficial. As expected, for the rest benchmarks we have slightly better frequent value hit rate than Figure 10.

The saving in cache energy, increase in program execution time, saving in energy delay product and power are shown in Figures 14–17. Generally, programs with higher FV hit rates save more energy (see Figure 14) and slow down less (see Figure 15). As we can see, while on on average this increase is 3.7%/3.5%/3.3% for 16/32/64 frequent values respectively for a 64Kb FVC. The savings in energy energy×delay, and power are substantial. For a 64Kb FVC, on an average, we see 26%/27.6%/28.8% reduction in cache energy, 23%/24.8%/26.1% reduction in energy×delay, and 28.9%/30.3%/31.3% reduction in power for 16/32/64 frequent values respectively. The savings are larger for larger number of frequent values and for larger sized cache. The only benchmark that presents high slowdown (8∼9%) is `107.mgrid`. It does not have abundant frequent value cache accesses as shown in our study in Figure 10 so it cannot benefit much from our FVC design. The other two benchmarks `132.ijpeg` and `129.compress` present increase in energy×delay. This is also due to the low FV hit rates as we can see from Figure 13. However, the increases are within only 1.75%.)

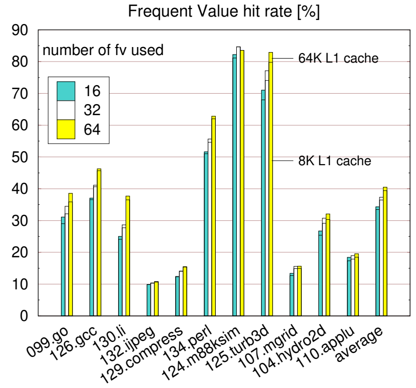

Figure 13. FV hits for 95% of cache accesses.

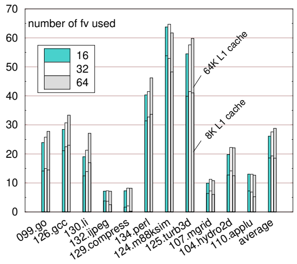

Figure 14. Cache Energy Reduction.

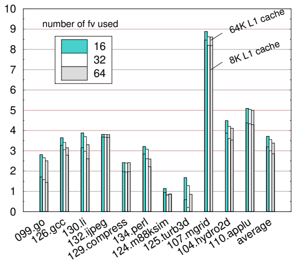

Figure 15. Delay Increase.

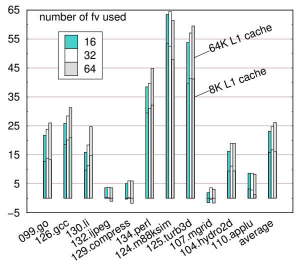

Figure 16. Energy×Delay Reduction.

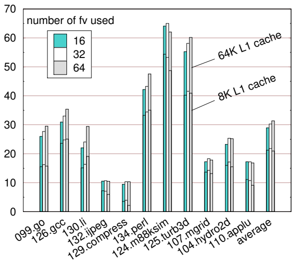

Figure 17. Power Reduction.

We also used Wattch to estimate the energy spent for the FV finder. We applied their CAM model and 2-bit register for the counters. The energy for the FV finder ranges from 0.3% to 6.07% of the total data cache energy used by an 8K to 64K FVC. This is because the FV finder runs for only first 5% of the memory accesses therefore does not introduce much overhead.

Load marking for minimizing delay increase. For the programs that have lower frequent value cache access rates, such as `107.mgrid`, the delay increases are higher than the other programs as shown in Figure 15. This is because most of the nonfrequent value accesses take two cycles, lengthening the average cache access time. We have further developed a scheme that minimizes the number of cache access that require two cycles instead of one.

We mark the loads that are involved in nonfrequent value accesses most of the time. When these loads are encountered at runtime, the two data arrays are read simultaneously. Thus, this approach enables reading of nonfrequent values in one cycle by the marked loads. Therefore, the overall increase in delay is reduced. If only those loads that are rarely involved in frequent value accesses, the energy savings will be effected minimally.

The potential of the above idea is indicated by the study whose results are presented in Figure 18. For each statically distinct load, through profiling, we collected the percentage of dynamic accesses by the load that involve nonfrequent value accesses. Next we set a threshold value for the maximum percentage of nonfrequent value accesses allowed by the dynamic instances of a static load. We refer to this threshold as the *nonfrequent value access threshold* (T). Each static load is classified as a *nonfrequent value load* if fewer than T% of its dynamic instances perform nonfrequent value accesses; otherwise it is classified as a *frequent value load*. In Figure 18, the values along x-axis represent different threshold values. The values along y-axis represent the number of load instances, as a percentage of total number of dynamic loads, that are due to the loads that are classified as *nonfrequent value loads* for a given threshold.

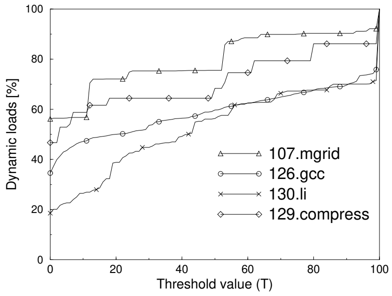

Figure 18. Load marking.

By examining the data in Figure 18 we make two observations. First that most loads are involved in some nonfrequent value accesses. Second and even more important observation is that there are some static loads that always access nonfrequent values. We choose to mark these loads as *nonfrequent loads*. By accessing both data arrays simultaneously for these loads, no reduction in data cache energy savings will occur. At the same time the increase in delay will be reduced as values will be obtained in a single cycle by these loads.

We reran the experiments for some of the programs to see if there is a significant difference in the delay increase due to load marking. Figure 19 shows the results for the benchmarks we have tested. The second column shows what percentage of the dynamic loads that arise from marked static loads. The third column shows the increase in delay before applying the load-marking scheme (taken from Figure 15). The last column shows the increase in delay after applying the load-marking scheme. We used a 64K-byte FVC with 64 frequent values in this experiment.

Benchmarks	loads marked	Delay Increase	
		before	after
`107.mgrid`	56%	8.6%	5.1%
`129.compress`	47%	2.4%	1.4%
`126.gcc`	34%	3.1%	1.8%
`130.li`	18%	3.3%	2.6%

Figure 19. Effects of load-marking on delay.

As we can see, the results of load marking are quite positive. The benchmark having the highest slowdown (i.e., `107.mgrid`) without load marking now has a significantly lower increase in delay—5.1% as opposed to 8.6%. Other two programs also show significantly reduced delay increases of 1.8% and 2.6%. Note that the load marking scheme does not affect the energy consumption since all the marked loads consume the same amount of energy as they did before load marking. Thus, the load marking scheme is an effective way to reduce the FVC delay increase.

5 Related Work

Our prior works propose other cache designs that exploit presence of frequent values [19, 18]. However, none of those designs are low power cache designs but rather they are aimed at improving the hit rate of the data cache. By storing data in compressed form, greater amounts of data is held in the caches to increase the hit rate. Furthermore the above designs increase the *cache cycle time*. In contrast, FVC does not increase the cache cycle time. Unlike the above designs, the miss rate for the FVC stays the same as the original cache. In [12] cache energy reduction based upon encoding zero value was introduced. In contrast, our approach considers a set of values for encoding.

We have also developed another application taking advantage of the FV temporal locality on the *off-chip data bus* [15, 16]. We found that on average 32% of the values transmitted on the off-chip data bus are frequent values. An FV "one-hot" bus encoding was therefore proposed to reduce the switching activity on the data bus. The encoder and decoder structure is shown in Figure 20. FV encoding achieved an average switching reduction of 30% with on-chip data cache and 49% without on-chip data cache for `SPEC95` and `mediabench` programs.

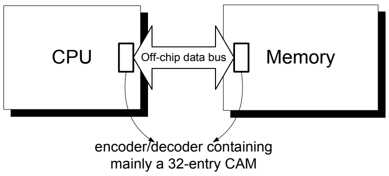

Figure 20. The frequent value off-chip data bus encoding architecture.

6 Conclusion

We have introduced a frequent value cache design which is based on recent research on frequent values in memory systems. Our FVC stores frequent values in encoded form so that they only need a few bits instead of full 32 bits in representation. With this encoding, the data array part of the cache is split into a smaller array and a larger array. The smaller array holds the short encoded form for frequent values and partial word of the same width for the nonfrequent values. The larger array holds the rest part of the word for both frequent and nonfrequent values. We carried out a detailed access time and access energy study of FVCs with varying configurations. The hardware design of a frequent value finder was also developed and shown to be simple and effective. From additional experiments we conclude that for a wide range of configurations for set-associative data caches, the FVC compares favorably to a conventional data cache design.

Acknowledgement. We would like to thank Josep Torrellas and Jose Renau for their generosity in providing modified XCACTI 2.0 which incorporates write energy, write back energy and line fill energy into the original version.

References

[1] D.H. Albonesi, "Selective Cache Ways: On Demand Cache Resource Allocation," *IEEE/ACM International Symposium on Microarchitecture* (MICRO-32), pages 248–259, 1999.

[2] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *International Symposium on Computer Architecture* (ISCA), pages 83–94, 2000.

[3] B. Calder, P. Feller and A. Eustace, "Value Profiling," *IEEE/ACM International Symposium on Microarchitecture* (MICRO-30), pages 259–269, 1997.

[4] R. Canal, A. Gonzalez, and J.E. Smith, "Very Low Power Pipelines Using Significance Compression," *IEEE/ACM International Symposium on Microarchitecture* (MICRO-33), pages 181–190, 2000.

[5] K. Ghose and M. B. Kamble "Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers, and Bit Line Segmentation," *IEEE/ACM International Symposium. on Low Power Electronics and Design* (ISPLED), pages 70–75, 1999.

[6] J-S. Lee, W-K. Hong, and S-D. Kim, "Design and Evaluation of a Selective Compressed Memory System," *International Conference. on Computer Design* (ICCD), pages 184–191, 1999.

[7] J. Kin, M. Gupta, and W.H. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," *IEEE/ACM International Symposium on Microarchitecture* (MICRO-30), pages 184–193, 1997.

[8] S. Onder and R. Gupta, "Automatic Generation of Microarchitecture Simulators," *IEEE International Conference on Computer Languages*, pages 80–89, 1998.

[9] M.D. Powell, S-H. Yang, B. Falsafi, K. Roy, T.N. Vijaykumar, "Gated Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories," *IEEE/ACM International Symposium on Low Power Electronics and Design* (ISLPED), pages 90–95, 2000.

[10] M. Huang, J. Renau, S. M. Yoo and J. Torrellas, "L1 Data Cache Decomposition for Energy Efficiency," *IEEE/ACM International Symposium on Low Power Electronics and Design* (ISLPED), pages 10–15, 2001.

[11] P. Ranganathan, S. Adve, and N. Jouppi, "Reconfigurable Caches and their Application to Media Processing," *International Symposium on Computer Architecture* (ISCA), pages 214–224, 2000.

[12] L. Villa, M. Zhang, K. Asanovic, "Dynamic Zero Compression for Cache Energy Reduction," *IEEE/ACM International Symposium on Microarchitecture* (MICRO-33), pages 214–220, 2000.

[13] S. Wilton and N.Jouppi "An Enhanced Access and Cycle Time Model for On-Chip Caches," *Technique Report, Western Research Lab.*, May 1993.

[14] S-H. Yang, M.D. Powell, B. Falsafi, K. Roy, and T.N. Vijaykumar, "An Integrated Circuit/Architecture Approach for Reducing Leakage in Deep-Submicron High-Performance I-Caches," *IEEE International Symposium on High-Performance Computer Architecture* (HPCA), pages 147–157, 2001.

[15] J. Yang and R. Gupta, "Frequent Value Locality and its Applications," to appear *ACM Transactions on Embedded Computing Systems*, 2002.

[16] J. Yang and R. Gupta, "FV Encoding for Low-Power Data I/O," *ACM/IEEE International Symposium on Low Power Electronics and Design* (ISLPED), pages 84–87, 2001.

[17] J. Yang and R. Gupta, "Energy-Efficient Load and Store Reuse," *IEEE/ACM International Symposium on Low Power Electronics and Design* (ISLPED), pages 72–75, 2001.

[18] J. Yang, Y. Zhang, and R. Gupta, "Frequent Value Compression in Data Caches," *IEEE/ACM International Symposium on Microarchitecture* (MICRO-33), pages 258–265, 2000.

[19] Y. Zhang, J. Yang, and R. Gupta, "Frequent Value Locality and Value-Centric Data Cache Design," *ACM 9th International Conference. on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 150–159, 2000.