Frequent Value Locality and its Applications

JUN YANG and RAJIV GUPTA
University of Arizona

By analyzing the behavior of a set of benchmarks, we demonstrate that a small number of distinct values tend to occur very frequently in memory. On an average, only eight of these frequent values were found to occupy 48% of memory locations for the benchmarks studied. In addition, we demonstrate that the identity of frequent values remains stable over the entire execution of the program and these values are scattered fairly uniformly across the allocated memory. We present three different algorithms for finding frequent values and experimentally demonstrate their effectiveness. Each of these algorithms is designed to suit a different application scenario. Since the contents of memory exhibit frequent value locality, it is expected that frequent values will be observed in data streams that flow across different points in the memory hierarchy. We exploit this observation for developing two low-power designs: a low-power level-one data cache and a low-power external data bus. In each of these applications a different form of encoding of frequent values is employed to obtain a low-power design. We also experimentally demonstrate the effectiveness of these designs.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*cache memories*; B.7.1 [**Integrated Circuits**]: Type and Design Styles—*input/output circuits*

General Terms: Design, Measurement, Performance

Additional Key Words and Phrases: Frequently occurring values, value profiling, encoding techniques, low power data bus, low power data cache

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 [Gabbay and Mendelson 1997; Lipasti et al. 1996]. Value locality has been exploited in the design of value reuse and prediction mechanisms for superscalar processors.

In this article we show that another kind of locality, which we refer to as the *frequent value locality*, is also quite prevalent in programs. The first aspect of the frequent value locality phenomenon is that 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 large fraction of these referenced locations. In fact, we observed that on an average, in fifteen of the `Spec95`

This work was supported by DARPA award no. F29601-00-1-0183 and National Science Foundation grants CCR-0208756, CCR-0105355, and EIA-0080123 to the University of Arizona.
Authors' address: J. Yang and R. Gupta, Department of Computer Science, University of Arizona, Tucson, AZ 85721.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2002 ACM 1539-9087/02/0011-0079 $5.00

programs, eight distinct values occupy 48% of all allocated memory locations throughout the execution of the program. This observation was first reported by us in Zhang et al. [2000] and has also been independently made by Larin [2000]. The second aspect of this phenomenon is that the set of frequent values remains quite stable throughout the execution of the program. The third and final aspect of frequent value locality is that frequent values are scattered fairly uniformly throughout the memory.

Although the frequent value locality phenomenon is related to the recent concepts of value locality and value prediction [Gabbay and Mendelson 1997; Lipasti et al. 1996], there are important differences between them. Frequent value locality characterizes the behavior of values present across the memory allocated to the program, throughout the execution of the program. In contrast, value locality and prediction characterize the behavior of values encountered during multiple executions of specific instructions in the program. As a consequence, the type of applications that can exploit these phenomenon are also quite different. Value locality is exploited for carrying out value prediction and thus, speculative execution of instructions to speed up a program's execution. Frequent value locality can be exploited in designing the memory hierarchy to achieve better power or performance behavior.

The three major contributions and the organization of this article are outlined below.

- We present experimental evidence for establishing the existence of frequent value locality phenomenon in many programs in Section 2.
- We present three different algorithms for finding frequent values in Section 3. Each of the algorithms is shown to be effective and appropriate for use in a different real-life setting.
- We discuss two applications that are able to successfully exploit the frequent value phenomenon by using the various frequent value finding algorithms in Section 4. These applications are low-power designs of a data cache and an external data bus that are suitable for the embedded systems domain.

2. FREQUENT VALUE LOCALITY

The *frequent value locality* phenomenon characterizes the behavior of values being held in live memory locations of running programs. The following three properties of the values characterize frequent value locality. We demonstrate these properties by analyzing the behavior of 15 `Spec95` benchmarks when run on *reference inputs*.

2.1 Frequent Value Occurrence in Memory

A small number of frequently occurring values, called *frequent values*, occupy a substantial fraction of memory locations allocated to an executing program.

To establish the above property we ran the benchmarks and examined the values in memory locations every 10 million instructions and averaged the frequencies of the values over the entire set of collected samples. During each sampling point, we scanned through the entire memory space and ranked every

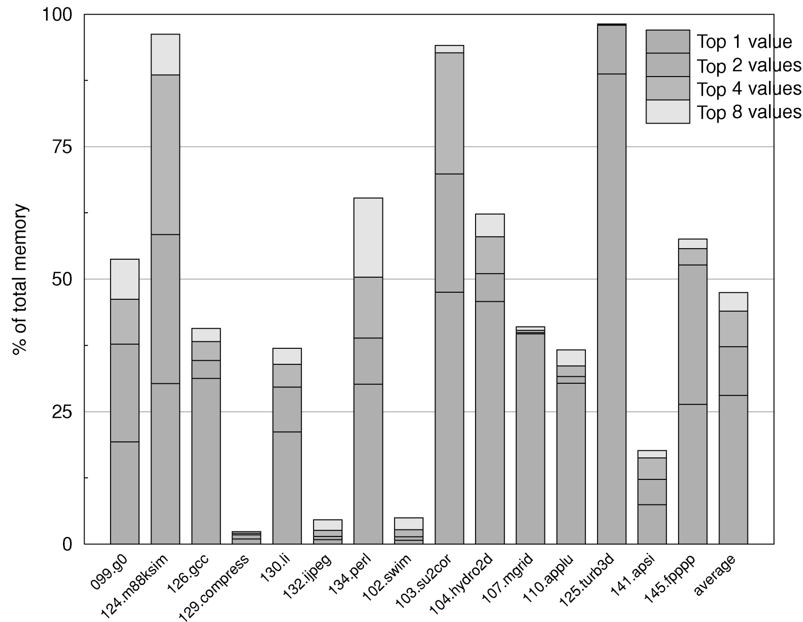

Fig. 1. Amount of memory occupied by top 8 frequent values.

distinct value according to its occurrence frequency. The memory locations that were considered at a given point included those that were of interest to the program. In particular, we considered currently allocated stack-and-heap memory locations. After the completion of the program's execution, for each encountered value, we computed its average frequency across all sampling points. The resulting average frequencies of all values were sorted in descending order. Values at the top of the list are more frequent than the values that appear later in the list. We spent a significant amount of time collecting this data as the program runs typically involved execution of billions of instructions.

Figure 1 shows that 12 out of 15 benchmarks exhibit this property, and on an average around 48% of memory locations are occupied by top eight frequently occurring values in the 15 `Spec95` benchmarks that were used in this study. The top eight frequent values are also listed in Figure 2. Examination of these values shows that there is a mix of small values (that can be represented using 16 bits) and large values (which require more than 16 bits). While the same small values (e.g., zero) are often observed across different programs, the same is not true for large values. This is because the large values are often memory addresses or string constants. Figure 3 shows what fraction of locations were occupied by small frequent values and large frequent values. In some programs the large values occupy a substantial number of locations.

2.2 Frequent Value Distribution in Time

The set of frequently occurring values remains fairly stable over a program run, which implies that frequent values can be identified and exploited during a program run.

Benchmark	1	2	3	4
099.go	0	351a	4	1
124.m88ksim	0	4022ada0	1c	40229030
126.gcc	0	e7	403	80004
129.compress	0	fffffff	65687420	20656874
130.li	0	3	1	4
132.ijpeg	0	1	ffff	10000
134.perl	78787878	78207820	0	20787820
102.swim	47435000	47435001	47434fff	47435002
103.su2cor	0	3fe00000	40040000	807bcdaf
104.hydro2d	0	3feccccc	cccccccc	3fe33333
107.mgrid	0	80000000	3c300000	bc300000
110.applu	0	2752547	4189374c	bfe16c8b
125.turb3d	0	80000000	20202020	3ff00000
141.apsi	0	3fb99be4	d443f3ee	d443f3ef
145.fpppp	9999999a	3fc99999	0	33333333

Benchmark	5	6	7	8
099.go	2	3	349	1c1
124.m88ksim	4022a610	60d12	8048bf7d	1db82340
126.gcc	40252734	10001	20	1b
129.compress	61687420	90a0a0a	68742065	20656820
130.li	6	1000000	5	40280df4
132.ijpeg	ffff0000	fffffff	10001	1ffff
134.perl	78207878	78782078	20782078	7878
102.swim	47434ffe	47435003	47434ffd	47435004
103.su2cor	3fd5f8e1	40290000	3fec71bc	390cf5ba
104.hydro2d	33333333	3ff00000	30303030	bc400000
107.mgrid	3c400000	bc400000	bc200000	40000000
110.applu	43958106	3f7b089a	3f90e560	80000000
125.turb3d	3a3a3a3a	1	6	3bc79ca1
141.apsi	d443f3f0	3fb99999	9999999a	20202020
145.fpppp	3fd33333	47ae147c	3fb47ae1	3fa47ae1

Fig. 2. Frequently occurring values ordered by decreasing frequency.

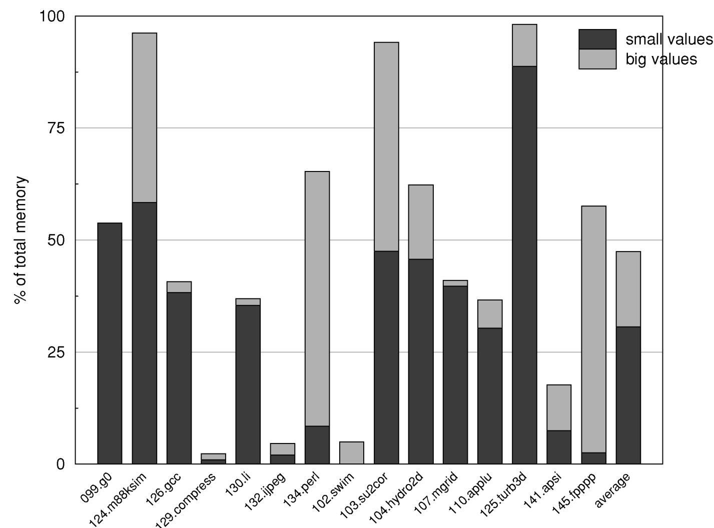

Fig. 3. Amount of memory occupied by small vs. big values.

To observe this property, we studied the occurrences of frequent values throughout the program execution. The graphs in Figures 4 and 5 show the behaviors of the benchmarks over the entire execution. To generate these graphs, we first found the top ten frequent values using the experiment already described above. Next, we ran the programs again, and during these runs, at each sampling point, for each of the top ten frequent values, we make note of the number of memory words that contain the frequent value.

We plotted the above data in the graphs where the X-axis represents time and the Y-axis represents the memory occurrence characteristics of the various frequent values. To make the graphs more readable, we did not plot these graphs for the entire execution of the program but, instead, carefully reduced the duration of a period in the middle of the program's execution. However, when we narrowed the scope of the data presented, we first examined the data for the entire execution and only narrowed the duration for which the data is presented if the behavior of the program was similar for the remainder of the execution. In these graphs, the topmost line represents the total number of allocated memory locations. The subsequent curves give us an idea of how many locations correspond to the top ten most frequently occurring values. The difference between the first (topmost) and second curves is the number of locations with the topmost frequent value. The difference between the first and third curves is the number of locations containing the top two frequent values and so on. From the results we can see that the fraction of allocated locations occupied by a given number of frequent values remains fairly stable throughout the program execution. This is because the same values continue to occur frequently over the entire execution of the program.

2.3 Frequent Value Distribution in Memory

The frequent values are distributed fairly uniformly throughout memory, which implies that no matter which part of memory is accessed, we are likely to encounter these values.

To establish the above property, we plotted the distribution of frequently occurring values in memory as shown in Figures 6 and 7. The data in these graphs represent the snapshot of memory at a point when the programs were nearly halfway through their execution. The referenced memory was broken into blocks of 800 consecutive locations each and the percentage of frequent values in each block of 800 locations was plotted as a point in the graph. We have selected a threshold of top eight frequent values in these graphs. As we can see, for nearly all of the programs, the frequent values are scattered across the memory, and for many programs the distribution of frequent values across the memory is quite uniform.

3. FINDING FREQUENT VALUES

The data in Figure 2 showed that the large frequent values always vary from program to program and small frequent values can also differ across programs. Since there is no universal set of frequent values, we must develop methods for

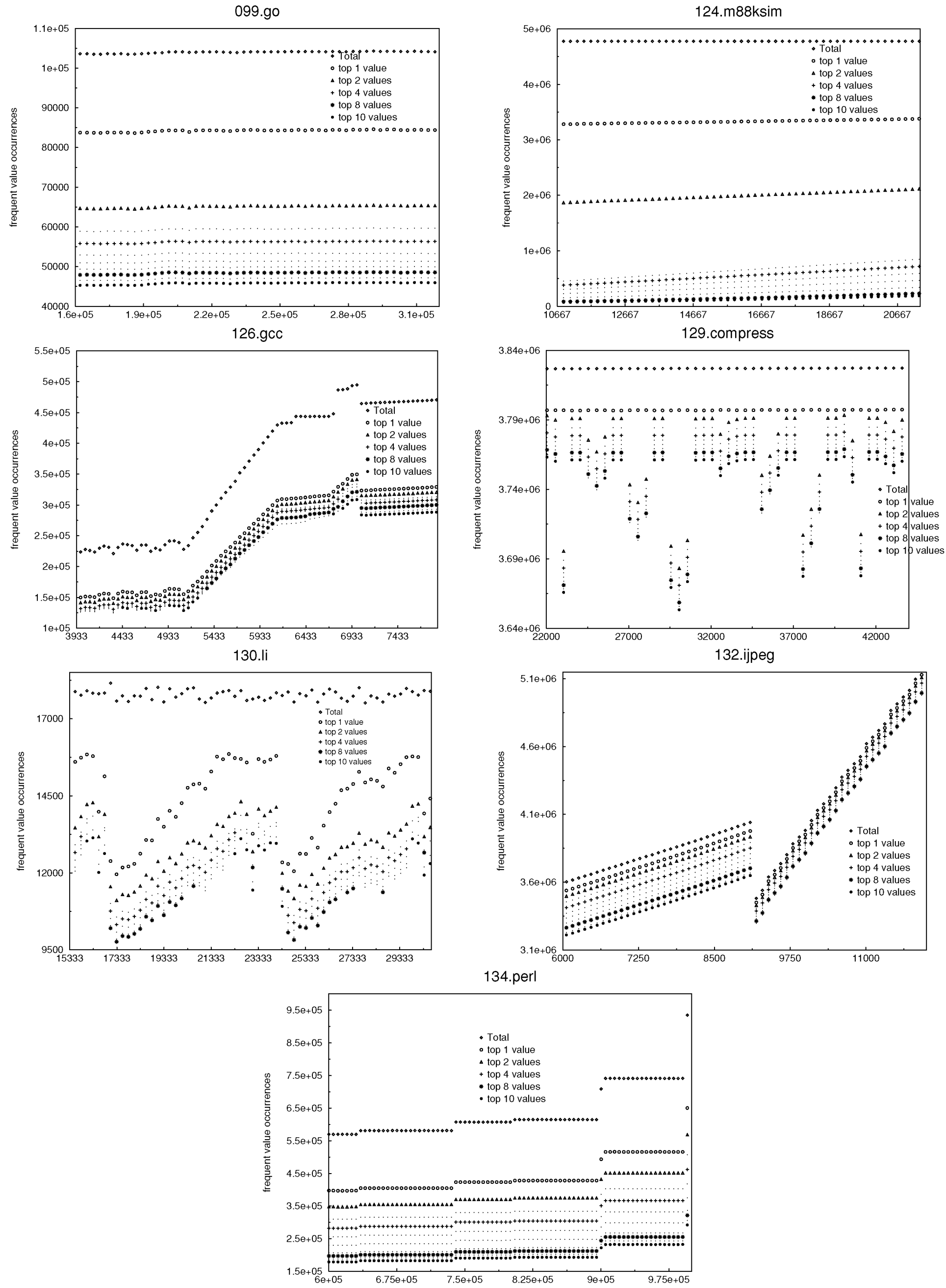

Fig. 4. Stability of frequent values over program execution.

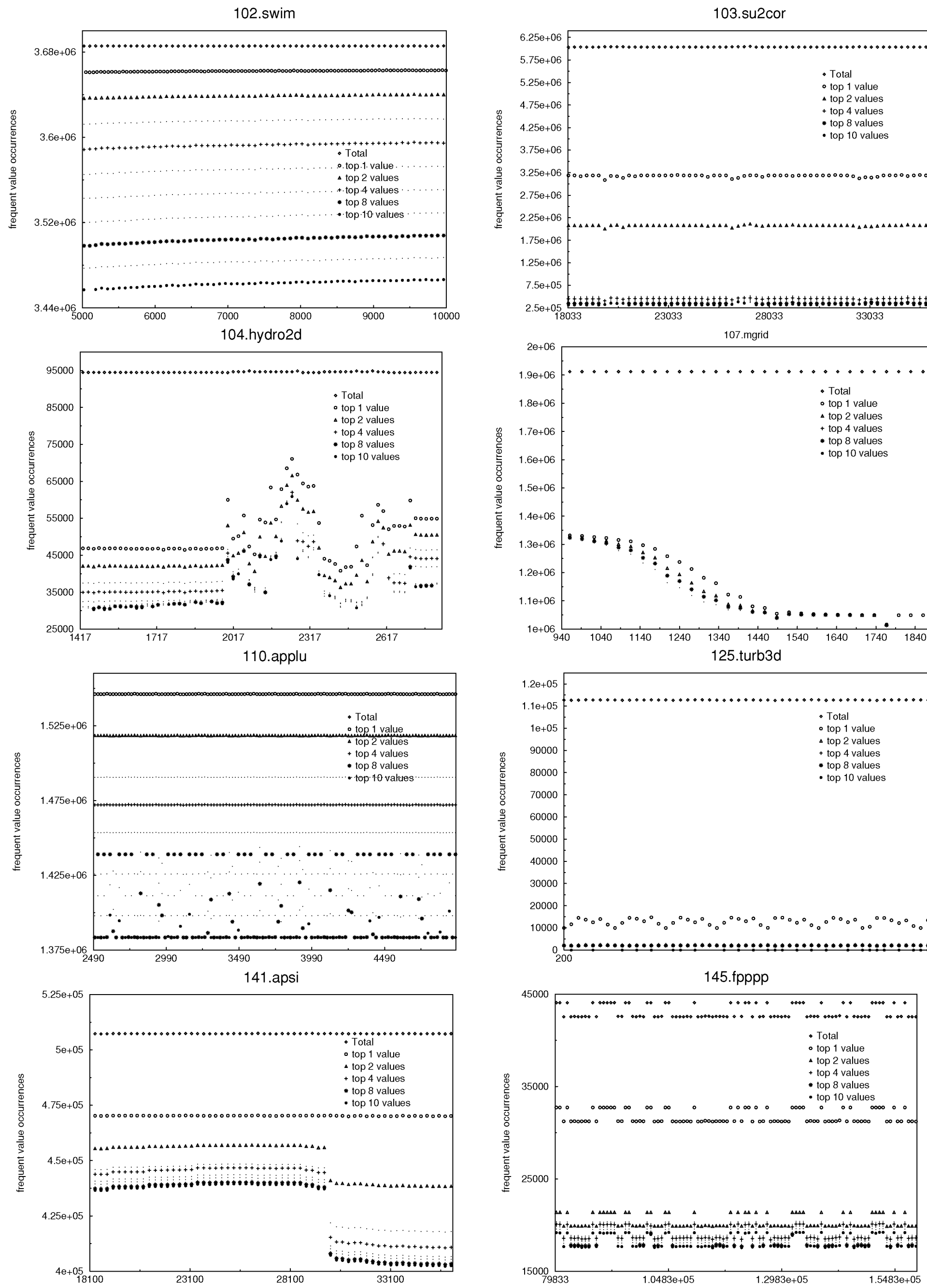

Fig. 5. Stability of frequent values over program execution.

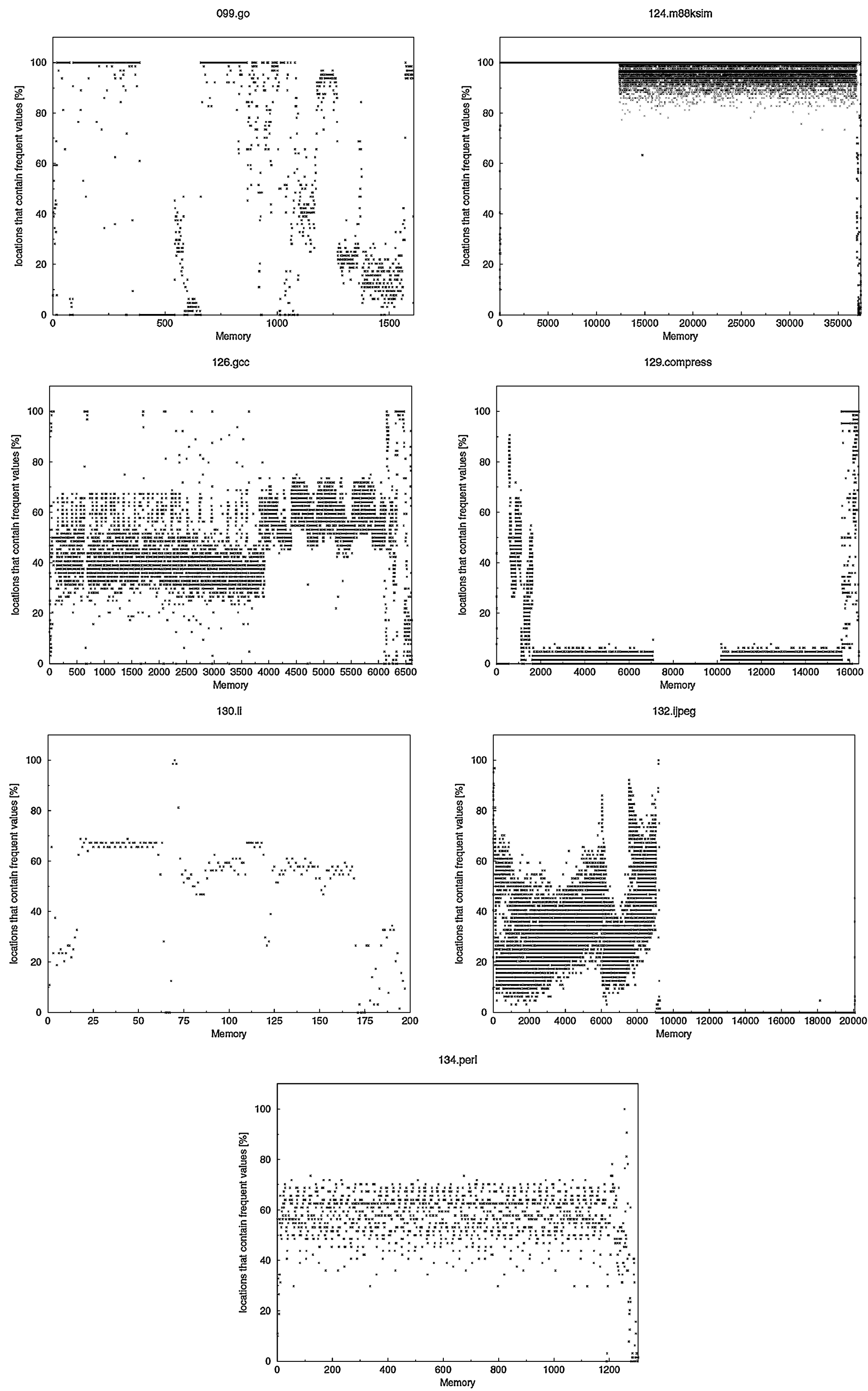

Fig. 6. Distribution of frequent values across memory (heap and stack).

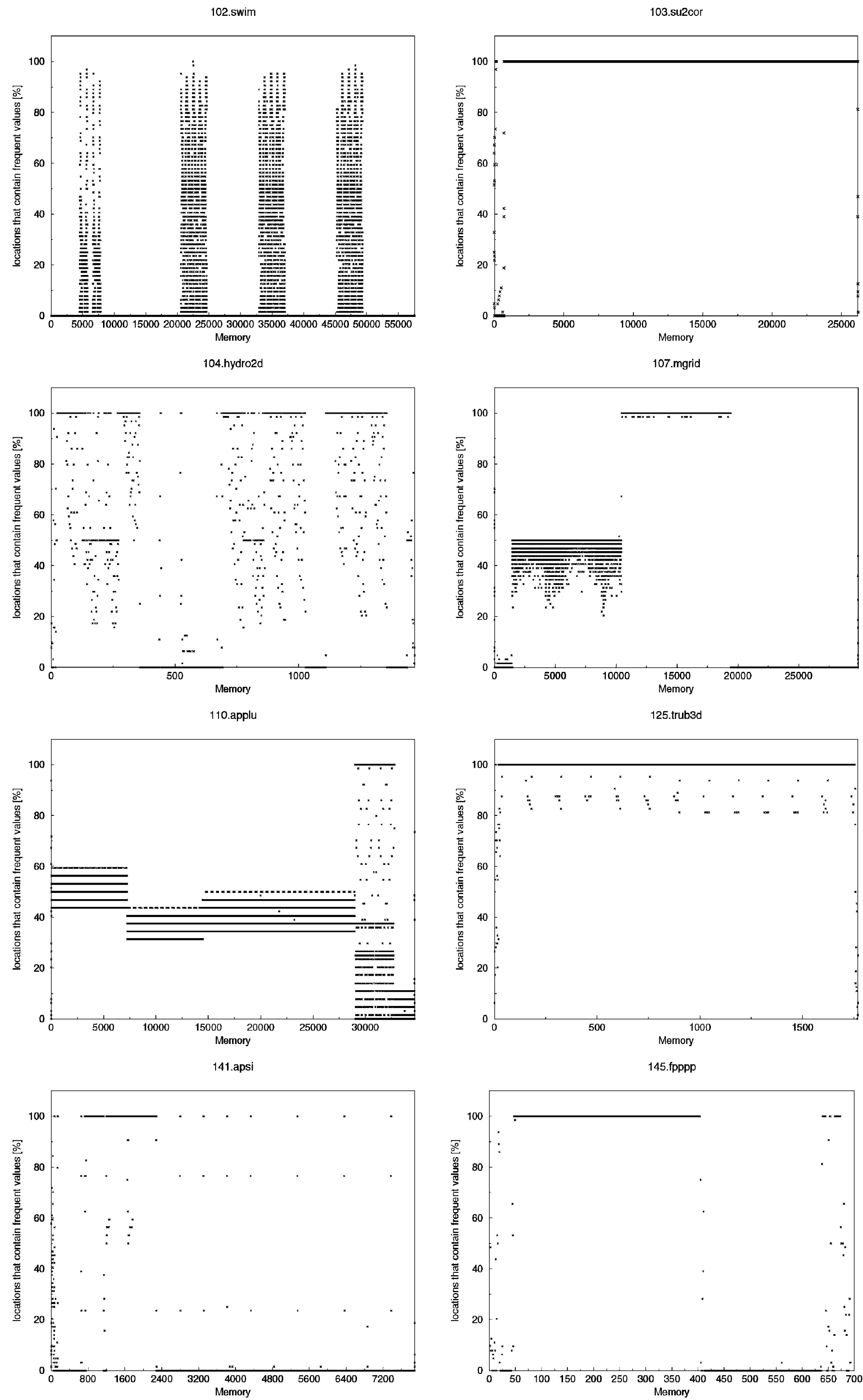

Fig. 7. Distribution of frequent values across memory (heap and stack).

identifying these values. Before we describe the different methods for finding frequent values, it is useful to understand the nature of applications that will make use of these methods.

While we have discussed frequent value locality in context of the memory contents of a program over its entire execution, the observations have much broader implications. Given that frequent values are observed across the memory, it is also expected that these values would be frequently encountered at all points in the memory hierarchy. For example, in the on-chip data cache, on the data bus that brings data into the on-chip cache, and of course in the main memory itself. At different points in the memory hierarchy at which frequent value locality is being exploited, different types of frequent value finding methods may be appropriate.

In this section we discuss a number of different approaches, ranging from software profiling techniques to hardware profiling techniques that can be used to find frequent values. Different methods are suitable for different applications, depending on the constraints under which the application must operate. We describe the following three scenarios for finding frequent values in this section and evaluate their effectiveness.

- *Find Once for a Given Program.* This method finds a *fixed frequent value set* through a *profiling run* which is then used by the application in all later *execution runs*. This is a purely software-based approach. Thus, once the values are known, they must be communicated to any hardware-based application, either through compiler-generated code or operating-system support. Moreover, if the frequent value set is sensitive to the program input, this approach will cause loss in performance.
- *Find Once Per Run of the Program.* This method finds a *fixed frequent value set* during each execution run of the program. The set of values is found through *limited online profiling* during the initial execution of the program, after which the values are fixed and profiling ceases. These values are then used by the application during the remainder of the execution. In other words, the *fixed frequent value set* is found during each execution, and therefore the frequent value set being sensitive to program input is not a problem for this method. This approach uses *specialized hardware* for finding the values. Therefore, no compiler or operating system support is required to communicate the values to the hardware.
- *Continuously Changing During Program Run.* This method maintains a *changing frequent value set* by carrying out *continuous profiling* of the program during each execution run. Moreover, profiling is carried out by specialized hardware. In this method, an application can benefit from adaptation of the frequent value set during a given run.

The two low-power applications that we consider later in this article, and the manner in which they fit into the above scenarios, are briefly described later to further motivate the need for algorithms that fit the above scenarios. As we can see, these two applications operate at different points along the memory hierarchy.

- *Low-Power Frequent Value Cache*. We present the design of a low-power data cache which stores frequent values in encoded form to reduce the dynamic activity in the data cache. This application must use a *fixed set of frequent values* because the encoding must remain fixed for the duration of the program. This is because a change in encoding would require at a minimum that we flush the cache. Moreover, here we are interested in finding frequently occurring values in the data stream between the CPU and the data cache, which we will refer to as the *frequently accessed values*.
- *Frequent Value Encoding for Low-Power Data Bus*. We also describe the design of a bus encoding technique which is aimed at reducing the switching activity on the external data bus of the CPU. This application can take advantage of a *continuously changing set of frequent values* since the encoding is localized to the data bus, that is, no other part of the system has to be aware that encoding is being carried out prior to sending a value across the data bus, and decoding is performed immediately after receiving the value at the other end of the data bus. Moreover, here we are interested in finding frequently occurring values in the data stream between the on-chip cache and off-chip memory, which we will refer to as *frequently transferred values*.

Note that since the data streams relevant to the above applications flow across different points in the memory hierarchy (between the CPU and the on-chip data cache and between the on-chip cache and the off-chip memory), they correspond to values stored in a program's allocated memory. Therefore, we expect these data streams to exhibit frequent value locality. In other words, we expect to find the presence of *frequently accessed values* and *frequently transferred values*.

Given the above applications, it is clear that the first two scenarios for finding frequent values are relevant for finding frequently accessed values, whereas the third scenario can be used for finding frequently transferred values. Therefore, in the remainder of this section, after describing our algorithms for finding frequent values under the three scenarios, we evaluate them in the appropriate context of frequently accessed values or frequently transferred values. All evaluations in this paper are based upon 15 programs from the `Spec95` benchmark suite which were run on the *reference inputs*, unless stated otherwise.

3.1 Find Once for a Given Program

The method for finding frequent values that we use under this scenario is simple but time consuming. Since this process is performed only once for a given program, we can justify spending a significant amount of time on finding frequent values. We instrument the program to intercept all data values involved in load and store instructions, as these are the values that constitute the data stream between the CPU and the data cache. We maintain a hash table in which we store all encountered values along with the frequencies with which they are encountered. The hash table size is not allowed to grow beyond an upper limit, which was 300 MB in our implementation. For some programs, this was large enough to hold all values encountered during the execution, but for others this was not the case. When the hash table reached its limit, we

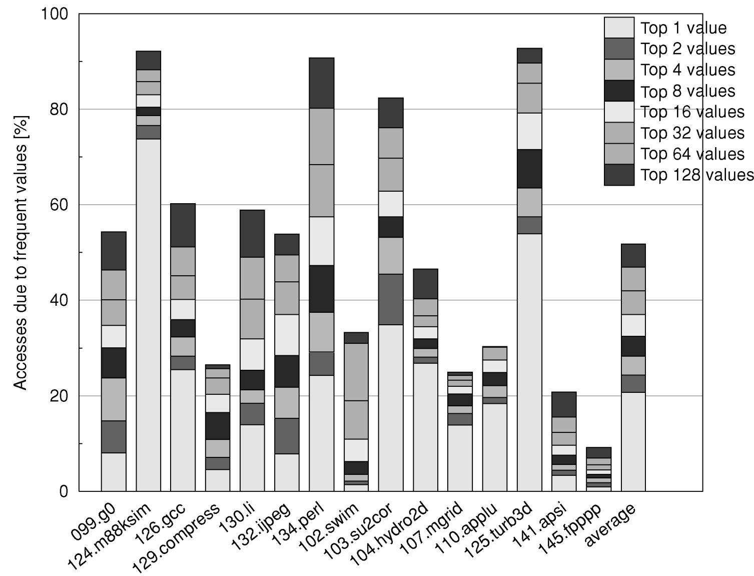

Fig. 8. Access percentage attributed to top 128 frequent values.

removed two-thirds of the least frequently occurring values from it and then continued processing future accessed values. The values discarded have a maximum occurrence count of 200, which is less than $10^{-4}\%$ of total accesses at the time. Therefore, it is highly unlikely that any frequent values would be discarded.

The results of implementing this method and applying it to `Spec95` programs are described next. Consider the data in Figure 8, which shows what percentage of total accesses involve *frequently accessed values*—we consider a maximum set size of 128 values during program runs based upon reference inputs. As we can see, the data stream of accessed values, contains frequently occurring values, as on an average 128 values account for over 50% of all accesses.

The data presented above is ideal data, since in collecting the above data both the profiling runs and the execution runs were carried out using the same inputs (reference inputs). Since the frequently accessed values will be found by running the program once on some input and used later during program runs on other inputs, we wanted to see how much is lost due to the sensitivity of frequent values to program inputs. Therefore, we next carried out an experiment in which the profiling run on *train inputs* was used to identify frequently accessed values. Then, accesses to these values were measured during program run on reference inputs. The results are shown in Figures 9 and 10. For a varying number of frequent values, the accesses to frequent values as a percentage of total accesses during a program run on reference inputs is plotted. One curve is based on the use of frequent values found from the profiling run on train inputs and for comparison, the other ideal curve used the reference inputs during the profiling run. As we can see, for most programs this approach does quite well, as the real curve is close to the ideal curve.

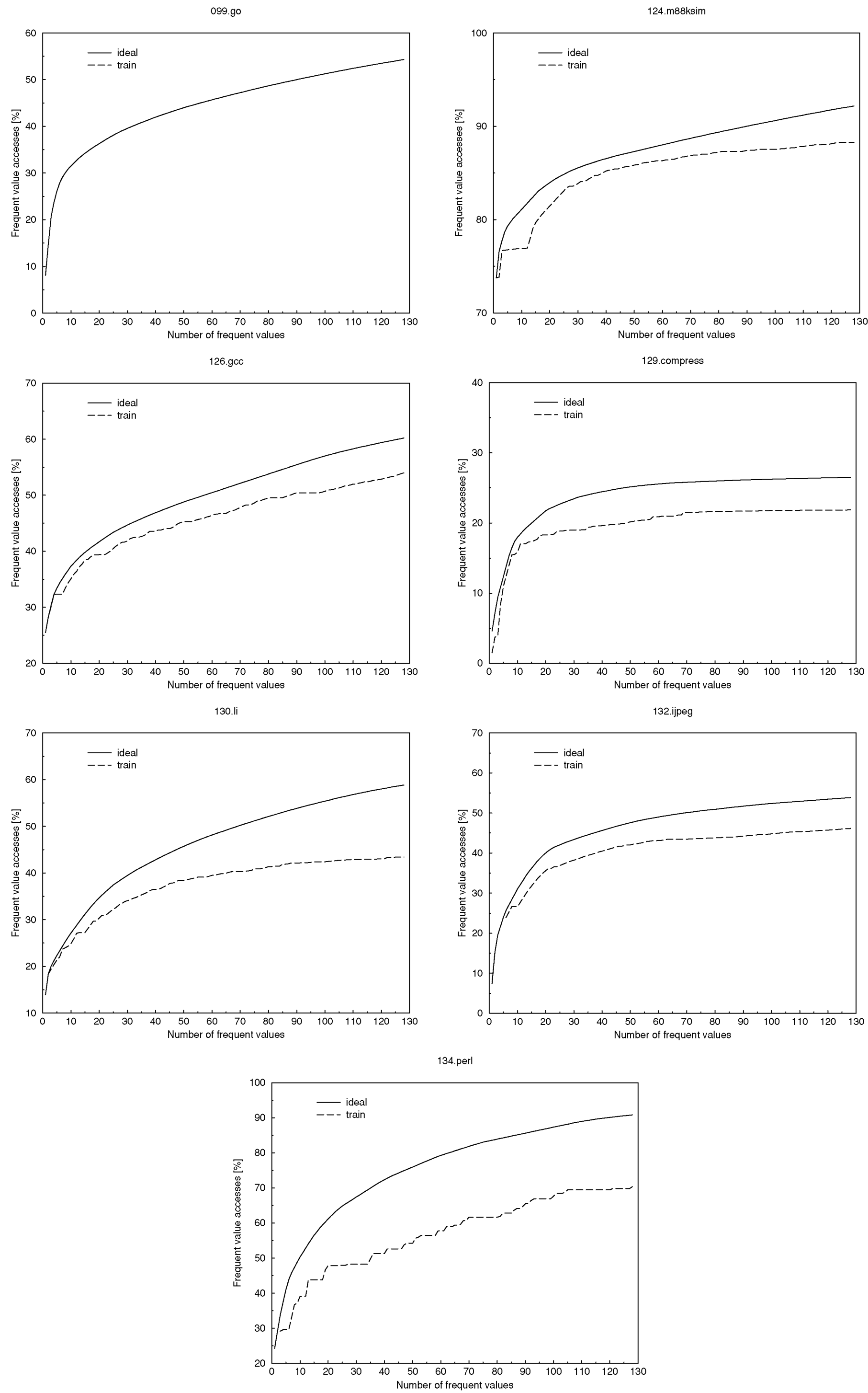

Fig. 9. Finding in profiling run for use in later execution runs.

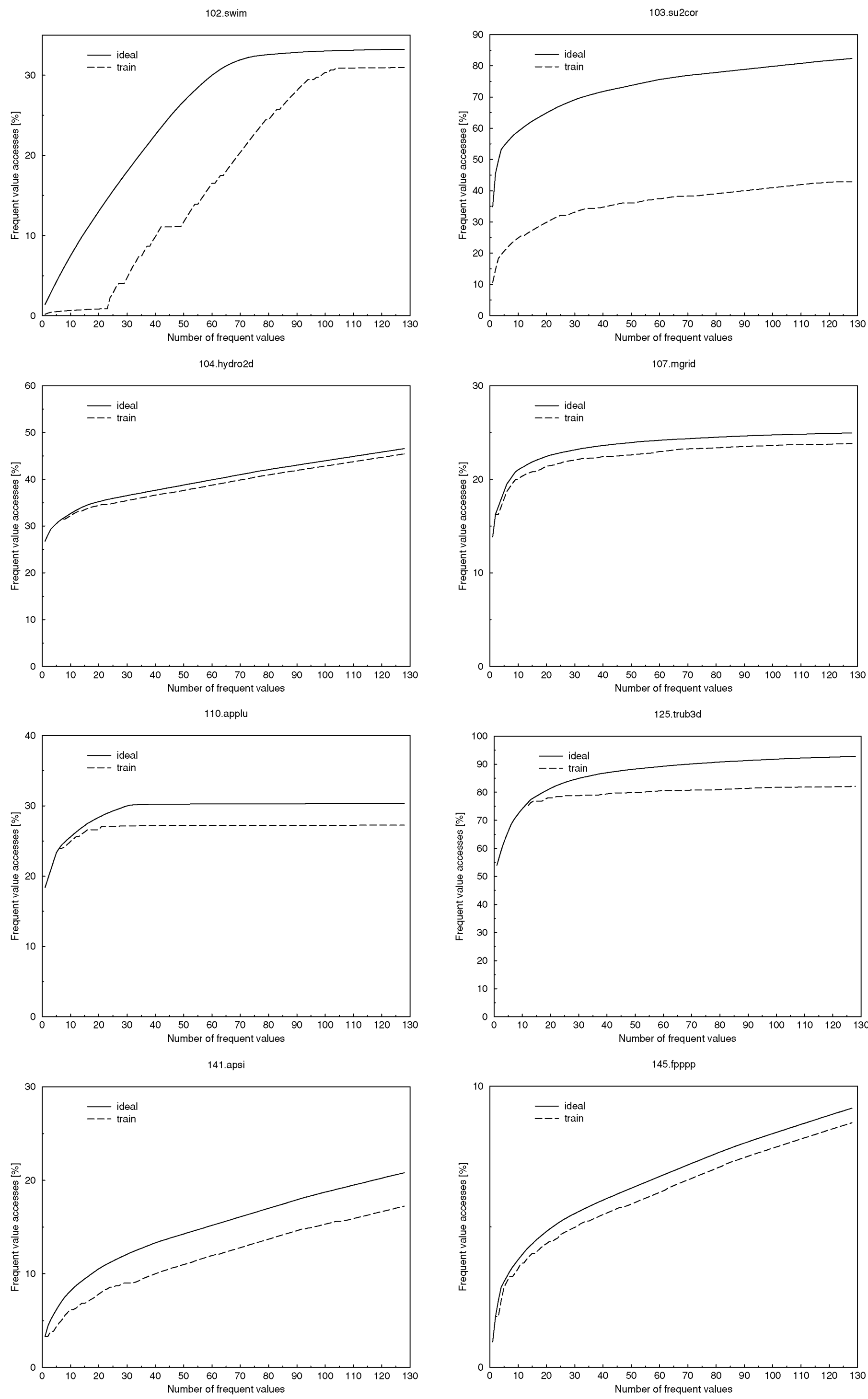

Fig. 10. Finding in profiling run for use in later execution runs.

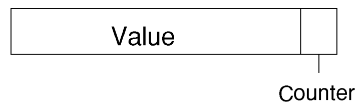

Fig. 11. Value table entry.

3.2 Find Once Per Run of the Program

As mentioned earlier, the algorithm for finding frequently accessed values during each program run is meant for implementation in hardware. Therefore we use a small table of frequent values in this method. To find the top n values, we use a table of $2n$ entries each having a *value* field and a *counter* field, as shown in Figure 11. The *value* field stores the data value encountered during monitoring and the *counter* field contains a c bit saturating counter.

Each time a data value is involved in an access by the CPU, we update the table of values as follows. If the value is already present in entry i, then the counter at entry i is incremented by one. When the counter saturates, the entry i is *swapped* with entry $i-1$. The purpose of this activity is to let frequent values gradually percolate to the top half of the table. When a new value is encountered, and there is no free entry in the table, a victim entry in the table needs to be selected to free up space. An entry is freed from the bottom half of the table with the smallest counter value because the bottom half is expected to contain values seen less often in comparison to values in the top half of the table.

Our method is inspired by the conventional software *value profiling* technique in Calder et al. [1997]. However, the algorithm in Calder et al. [1997] 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. When half of the table is cleared, 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. When replacing a value from the bottom half, the counter value is used to free up an entry corresponding to a less frequently seen value from among the values in the bottom half of the table.

Our approach to approximating sorting is very effective in practice, as our experiments comparing *conventional value profiling* with our *hardware value profiling* show. We compare the two algorithms by comparing the quality of their frequently accessed value sets, which is measured in terms of accesses that can be attributed to the values in the set. In the experiments, we varied the swapping interval by varying the counter width c from 1 bit to 3 bits. A longer interval means frequent values climb up in the table at a slower pace, and a shorter interval leads to faster convergence but may cause excessive swapping between two entries which already contain frequent values. From Figure 12, it is interesting to see that counter lengths of 1 and 2 bits give nearly the same results; however, a 3-bit counter degrades a little. This is because the interval

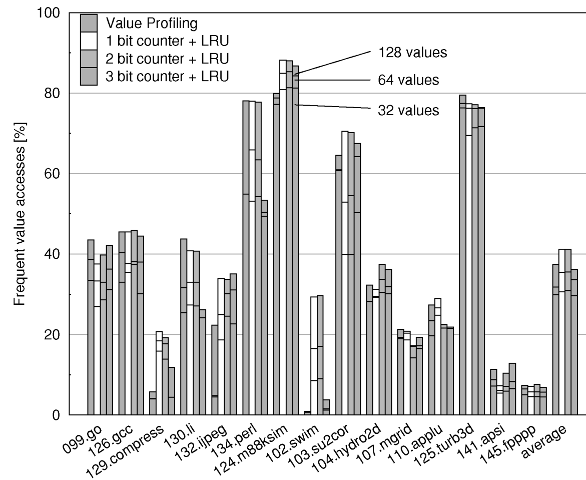

Fig. 12. Comparison of value profiling technique and our hardware method for capturing 32, 64 and 128 frequent values.

between swaps is longer, causing a slower pace for frequent values to move up. Therefore, in the rest of the experiments, we chose a 2-bit counter which achieves the same results as a 1-bit counter without introducing unecessary swaps. When compared with the value profiling technique [Calder et al. 1997], our algorithm produces nearly the same results as the conventional algorithm, and in many cases performs even better (e.g., for `129.compress`, `132.ijpeg`, `124.m88ksim`, `102.swim`, and `103.su2cor`).

Next we show how effective this method is in finding frequently accessed values. The effectiveness of this algorithm depends upon the degree of profiling. One can expect that a greater amount of profiling will usually be more effective. However, the more profiling we do, the less time the program has left for exploiting the frequently accessed values. Therefore, in our experiments we varied the amount of profiling from 1 million to 800 million instructions for most programs of moderate size. The results are presented in Figures 13 and 14. For each benchmark we present a set of curves corresponding to different profiling levels, which are specified in terms of number of instructions, and in parenthesis we indicate what fraction of total program execution was spent on profiling. Even though in our experiments we varied the profiling levels between 1 and 800 million instructions, the profiling periods displayed in these plots were selected to show interesting areas of the graph. In some cases, many short profiling intervals are shown while in others longer profiling intervals are shown.

The results in Figures. 13 and 14 show that the programs can be divided into two categories. For many of the programs, the degree of profiling makes only a small difference (e.g., `124.m88ksim`). In other words, the frequently accessed values can be identified using a small amount of profiling and greater amounts of profiling are not necessary. The reason for this behavior is that, usually, a very small subset of frequently accessed values account for most of the frequent

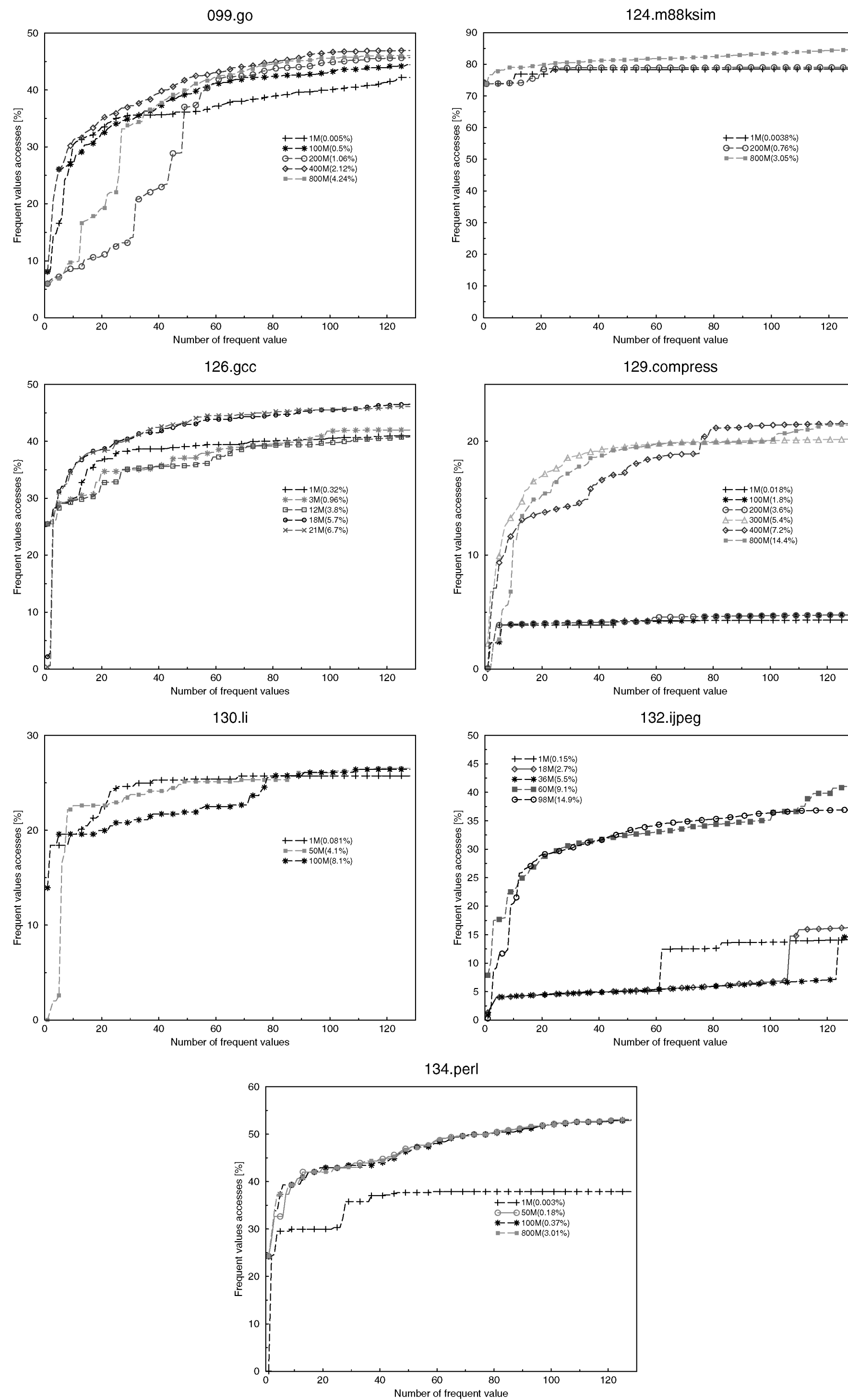

Fig. 13. Finding and using in each execution run.

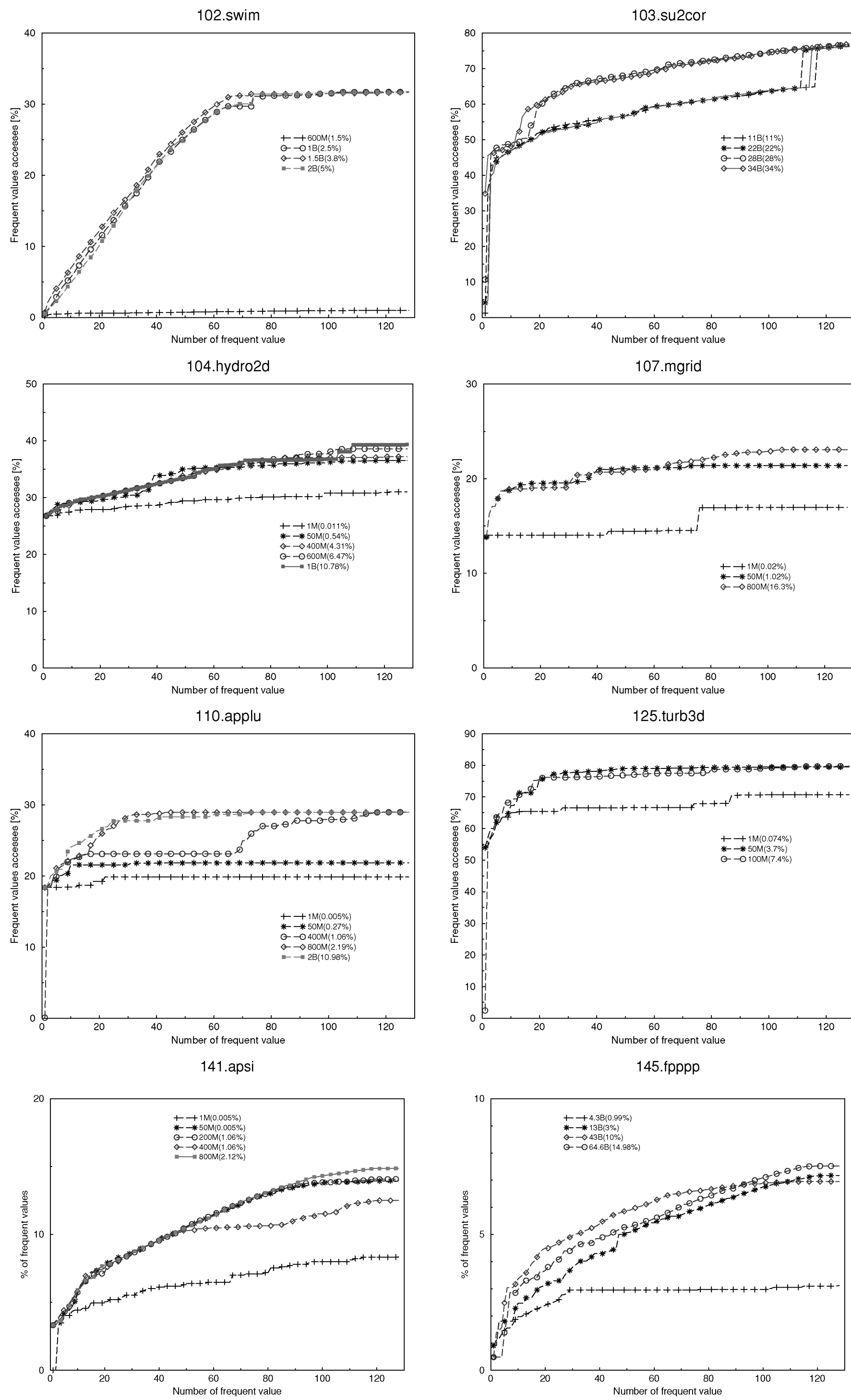

Fig. 14. Finding and using in each execution run.

value accesses, and these values are so frequent that they are seen immediately as execution begins. For example, in Figure 8, we can see that in the case of `124.m88ksim`, the top 128 values account for 92% of all accesses; however, the topmost value alone accounts for 74% of all accesses. For other programs, increasing the profiling interval beyond a certain threshold makes a significant difference (e.g., `129.compress`). This is because, for these programs, typically a larger number of frequent values need to be identified accurately because they all account for a significant number of accesses. The larger the number of important frequent values, the longer it may take to find them, as some of these values may show up a bit later in the execution. For example, Figure 8 shows that in case of `129.compress`, whereas the top 128 values account for nearly 27% of all accesses, the topmost value accounts for only 5% of accesses. In fact, to get close to 27% of accesses, it is important to accurately identify the top 32 frequently accessed values for `129.compress`.

3.3 Continuously Changing During Program Run

Let us now consider the hardware algorithm for maintaining a continuously changing set of frequent values. A table with as many entries as the number of frequent values that are to be identified is maintained. We use the LRU replacement policy for filling and updating the frequent value table. To gain time ordering information, we use a *reference bit* and an n-bit timestamp for each value recorded in the coder. The reference bit is set when the value appears at the input. At regular intervals, the reference bit is shifted right into the high-order bit position of the n-bit timestamp, causing all bits in the timestamp also to be shifted right and the lowest-order bit in the timestamp being discarded. This operation is performed for all entries in the two tables and at the same time that all the reference bits are reset. Thus, the timestamp keeps the history of value occurrences for the last n time periods. For example, the timestamp of 000 means this value did not appear during the last three time intervals, timestamp 100 means it was seen only in the last interval, and the timestamp 000 with reference bit set means it is encountered in the current time slot. When an entry is required and a value is to be evicted, the entry that is selected is the one with the smallest timestamp and clear reference bit. The new value is put in with a fresh reference bit and timestamp (all 0's) in this selected entry.

Since we use the above approach for the bus encoding application, we evaluated it in context of data stream between on-chip cache and off-chip memory. We measured the percentage of data traffic that could be attributed to the changing set of 32 frequently transferred values found using the above algorithm. Figure 15 shows the results of this experiment. On an average, 32% of the traffic was attributed to the frequently transferred values. It is interesting to note that in the case of the `129.compress` benchmark, when a fixed set of frequently accessed values was used, it did not account for a substantial number of accesses; but when a changing set of frequently transferred values was used, it accounted for nearly 68% of the total traffic. Therefore, while for some benchmarks a fixed set of frequent values may be adequate, for others, a changing set may provide better results.

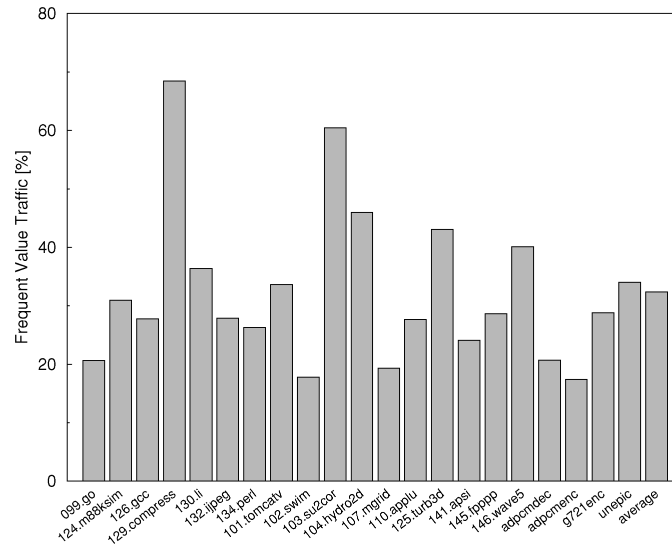

Fig. 15. Data bus traffic due to 32 frequent values.

4. APPLICATIONS

4.1 Frequent Value Cache

In this section, we demonstrate how the *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, 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. Therefore, FVC design represents a trade-off between lower energy consumption for *frequent value* accesses and higher access times for *nonfrequent value* accesses.

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 range 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 stored in $log_2 n$ number of bits, which range 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 so that one array contains $log_2 n$ bits corresponding to each word and the other contains the remaining $32 - log_2 n$ bits (see Figure 16). Frequent values are stored in encoded form in the low-bit

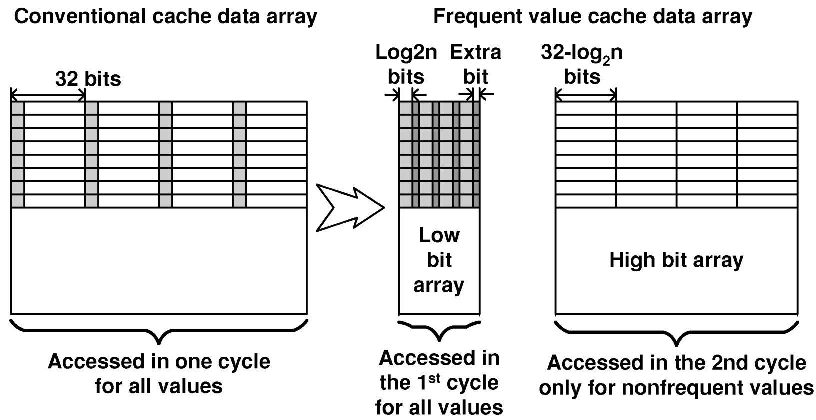

Fig. 16. Partitioning the data array.

array while nonfrequent values use the space in both data arrays. An additional bit corresponding to each word in a cache line is needed to indicate whether the word contains an encoded frequent value or an unencoded nonfrequent value. Therefore, an N word cache line needs N additional bits. These bits are stored along with every word in the low-bit array so that its word width becomes $log_2n + 1$.

When reading a word from the cache, initially we simply read from the low-bit array. Since the bits read out contain a flag bit, we examine it to determine what comes next. If the bit is set, which means the value was stored in encoded form, we need not read any additional bits and must proceed to decode the value. In this case we have greatly reduced the activity in the data cache. On the other hand, if the value is stored in unencoded form, we proceed to access the remainder of the word from the second data array.

Since the retrieval of log_2n bits from the low-bit array and that of $32 - log_2n$ 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, a frequent value is read in one cycle while a nonfrequent value is read in two cycles. In other words, in the first cycle, the log_2n 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 array are accessed.

We developed a detailed model for the FVC design. The details of the design are shown in Figure 17. As we can see, instead of one data array in a conventional cache, there are two data arrays—a low-bit array of $log_2n + 1$ bits per word and the high-bit array of $32 - log_2n$ bits per word. In a realistic cache implementation, the entire line is first read out from the data array, then the appropriate subset of bits corresponding to the word being accessed are 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.

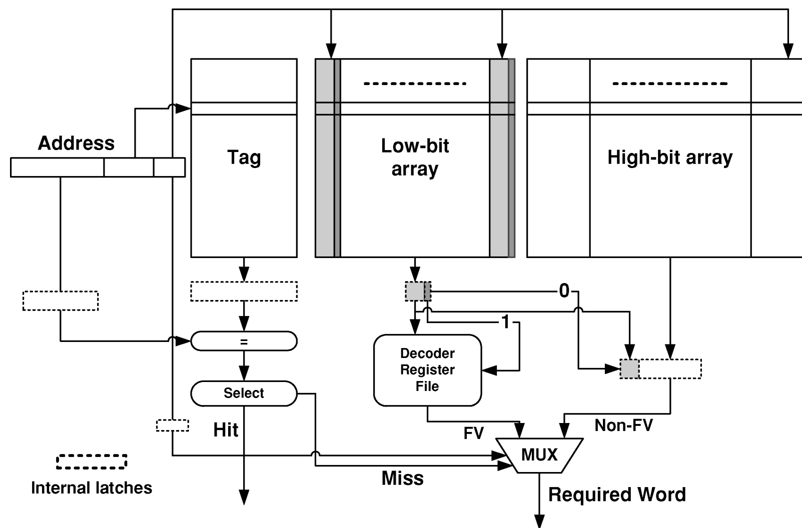

Fig. 17. The FVC design.

Therefore, decoding will increase the cache access time, which is not desirable. Hence, we adopt the subbanking scheme proposed by Villa et al. [2000] and Ghose and Kamble [1999], in which the subbank containing the target word can be read independently. So we drive only those bitlines of the local wordlines that are required. In that way, they perform early selection of proper words from the line that are outputs from the low-bit array. Once a code is selected, it is passed to a *frequent value register file* that contains the frequent value. 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-bit flows 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.

The hardware for encoding during cache write operations is designed as a CAM that can match an incoming frequent value and output its CAM index in binary form. 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.

We have modified XCACTI 2.0 [Huang et al. 2001; Reinman and Jouppi 1999] to incorporate a model of the above FVC design. Accessing a frequent value differs from conventional access in that the word width is narrowed down to code size plus one. Accessing an infrequent value increases each word width by one. For the *frequent value register file*, we adopted the register file model in Brooks et al. [2000], using the same technology. We used those models to compare the access time and energy behavior of the FVC for a range of configurations for a conventional cache and FVC—including varying associativity, cache size, line size, and the number of frequent values. Next, we present some details of this

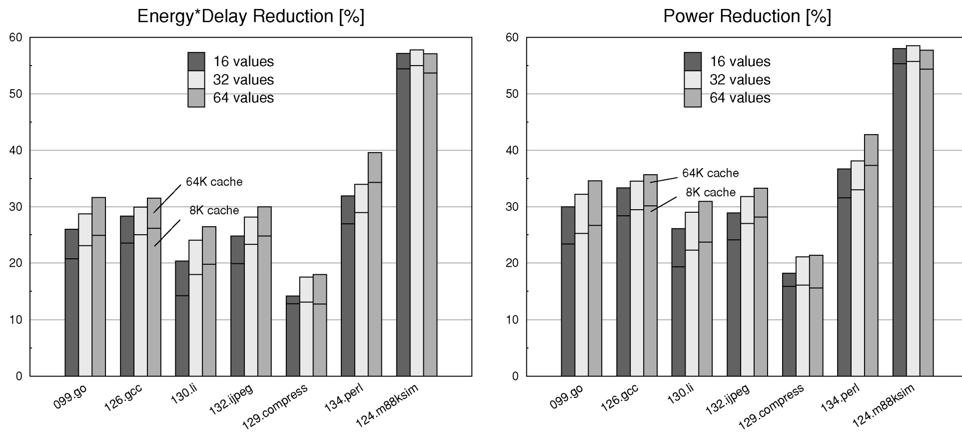

Fig. 18. Use frequent values found from train input.

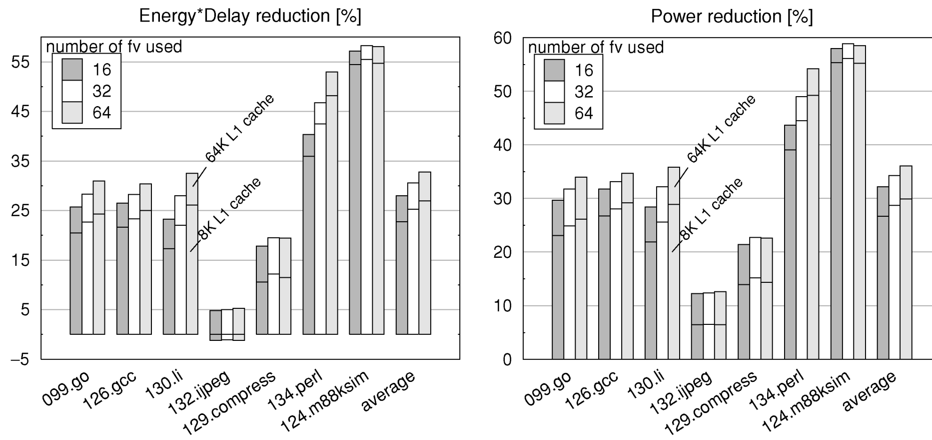

Fig. 19. Use frequent values found in first 5% program run.

study to demonstrate the feasibility and effectiveness of our FVC design. All the data presented in this paper is for 0.18 μm technology. In all experiments performed for this paper we used our FAST simulation system [Onder and Gupta 1998].

In our experiments, we considered two cache configurations: 8 Kbyte–16 bytes per line–4-way set associative cache and 64 Kbyte–32 bytes per line–8-way set associative cache. We verified that the access times of the conventional cache and the FVC were the same. Further, we considered configurations for FVC where $log_2 n$ was varied from 4 bits to 6 bits (i.e., number of frequent values was varied from 16 to 64). The results in the form of energy-delay product and power reduction are shown in Figures 18 and 19. Two sets of results are presented according to the two different methods for finding frequent values. In the first case, we found the frequent values using train inputs and used them during the reference input run. In the second case, we found the frequent

values by monitoring at runtime the values accessed during the first 5% of the program's execution and then using them for the remainder of the run. For the 64 Kbyte cache, which uses 64 frequent values, the average reductions in energy-delay product and power are nearly 33%. These reductions fall slightly if fewer frequent values are used or the size of the cache is smaller (i.e., 8 Kbyte).

4.2 Frequent Value Encoding

Because the I/O pins of a CPU are a significant source of energy consumption, work has been done on developing encoding schemes for reducing switching activity on external data buses [Stan and Burleson 1995; Benini et al. 1999]. However, existing techniques provide only modest amounts of reduction in switching activity. This is because existing techniques are general purpose in nature and do not exploit the characteristics of data transmitted over the data bus. This is because until now no special characteristic of data streams transmitted over the CPU's external data bus has been identified.

The phenomenon of frequently occurring values is a characteristic that is highly suitable for exploitation by a bus encoding technique for achieving reductions in switching activity. In fact, for the benchmarks used in this paper, we observed that frequent values account for over 32% of transmissions over the external data bus. Therefore we designed a simple new encoding scheme, called *FV encoding*, that is significantly more effective than prior techniques.

Our overall approach is as follows. The frequent values are transmitted over the bus in encoded form while the nonfrequent values are transmitted in their original unencoded form. The set of frequent values is kept in a table implemented as a CAM by both the encoder and the decoder. This table is searched, and if the value to be transmitted is found in it, then the value is regarded as a frequent value which is then transmitted in encoded form. In order to ensure that the decoder can determine whether the transmitted value is in encoded form or not, additional *control* signals must be sent from the encoder to the decoder in some situations. By using the same method for maintaining frequent values at the two ends of the bus, we ensure that the contents of the frequent value tables at both the encoder and the decoder are always identical.

Our method for encoding frequent values has the flavor of one-hot encoding, with one important difference. Our encoding scheme overcomes the major drawback of one-hot encoding in that it does not require 2^n wires, where n is the number of bits representing the value, to transfer the data. Instead, it achieves low switching activity by using the same number of wires as the data bus width. In this work, we assume that this number is 32. We are able to achieve the above goal as follows. The "hot" wire generated from the encoder is not used to represent the true value being transferred but, rather, it indicates in which entry of the frequent value table in the encoder or decoder the frequent value can be found. In other words, if the ith entry in the frequent value table is found to contain the same value as the one being transmitted, then the ith output wire is set to 1 and all the remaining wires are set to 0. This is how a *one-hot code* is formed and sent over the data bus,

completing the coding process. When the decoder receives the code from the bus, it reads out the value from the ith entry indicated by the code. Under the above scheme, if frequent values are transmitted back-to-back, then at most two bits switch while all other bits remain zero. This is how switching activity is reduced.

The nonfrequent values are transmitted in unencoded form. If a value to be transmitted is a nonfrequent value, it cannot be found in the encoder CAM. Thus, the encoder does not attempt to generate a code. Instead, it simply passes the original value onto the data bus. When the decoder receives the value and finds more than one hot wire in it, it concludes that the transmitted value is not encoded.

It is possible that a nonfrequent value being transmitted in unencoded form contains a single high bit and all of its remaining bits are zeros. We ensure that the decoder does not erroneously decode this value by sending a single bit control signal from the encoder, telling the decoder to skip decoding. Our experimental results also include the switching overhead resulting from sending the control signal.

The base encoding scheme reduces switching to at most 2 bits if a frequent value being transmitted is also preceded by a frequent value. While our base encoding scheme gives good performance when frequent values are encountered back to back, a pattern of intervening frequent and nonfrequent values is not favorable to our base scheme. We measured the percentage of traffic due to frequent values that are also preceded by frequent value transmissions. On an average, this number is 16%. We also know that, on an average, the frequent values account for 32% of the overall traffic. Therefore, on an average, 16% of transmitted values are frequent values that are preceded by nonfrequent values.

We further reduce switching between nonfrequent and frequent value transmissions by using a decorrelator described in Benini et al. [1999]. If we take the *XOR* of the current value to be transmitted ($Code_n$) and the previously transmitted value ($Send_{n-1}$), then this has the effect of flipping only those wires of the bus that are high in $Code_n$. Therefore, if $Code_n$ corresponds to a frequent value, it contains only 1 high bit, and therefore no matter whether it is preceded by a frequent value or a nonfrequent value (i.e., $Send_{n-1}$ is frequent or nonfrequent), the switching activity is only 1 bit. In other words, transmission of a frequent value always results in switching off one bit.

Experiments show that FV encoding of a changing set of 32 frequent values yields an average reduction of 30% in data bus switching activity for our benchmarks (see Figure 20). The figure also shows that if a fixed set of frequent values is used, the reductions would be around 20%. Furthermore, Figure 21 shows that the reduction in switching achieved by FV encoding is 2 to 4 times the reduction achieved by the *bus-invert* [Stan and Burleson 1995] coding scheme, and 1.5 to 3 times the reduction achieved by the *adaptive method* in Benini et al. [1999]. In all of the above experiments, the CPU we used also had on-chip instruction and data caches. If these caches are removed, the reductions in switching activities increase even further.

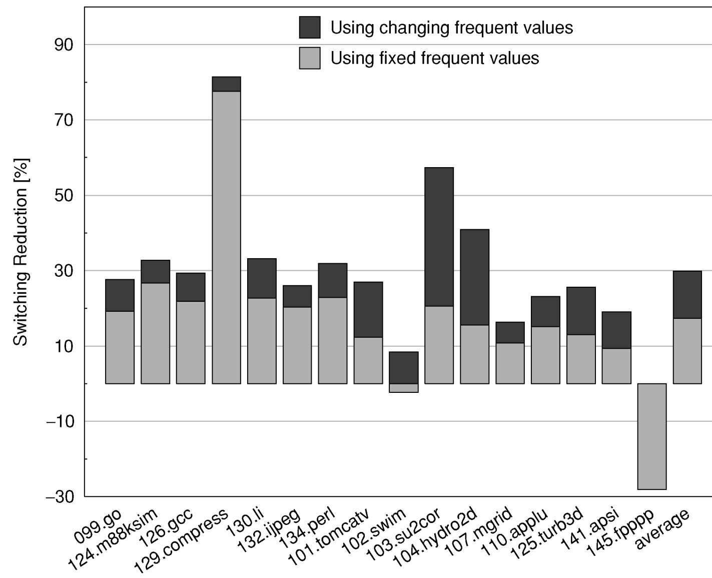

Fig. 20. External data bus switching reduction using FV encoding.

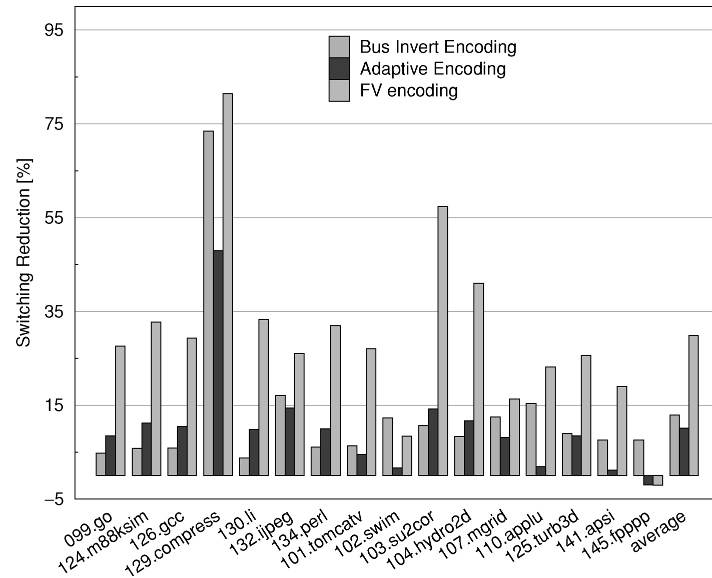

Fig. 21. Comparison with bus-invert and adaptive encoding.

5. CONCLUSIONS

In this article we presented experimental evidence that shows that most programs exhibit frequent value locality, according to which small number of distinct values are encountered very frequently in memory. As a consequence, frequent values are also observed in data streams at various points in the

memory hierarchy. We discussed two low power applications of frequent value phenomenon that are important for the domain of embedded systems. The frequent value data cache provides significant reductions in dynamic energy consumed by the cache and the frequent value encoding results in significant reductions in switching activity on an external data bus of a CPU. We presented three algorithms for finding frequent values under different settings and demonstrated that they are highly effective in practice.

REFERENCES

BENINI, L., MACII, A., MACII, E., PONCINO, M., AND SCARSI, R. 1999. Synthesis of low-overhead interfaces for power-efficient communication over wide buses. In *Proceedings of the Design Automation Conference*. ACM, New York, 128–133.

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*. ACM/IEEE, New York, 83–94.

CALDER, B., FELLER, P., AND EUSTACE, A. 1997. Value profiling. In *Proceedings of the 30th International Symposium on Microarchitecture*. IEEE/ACM, New York, 259–269.

GABBAY, F. AND MENDELSON, A. 1997. Can program profiling support value prediction? In *Proceedings of the 30th International Symposium on Microarchitecture*. IEEE/ACM, New York, 270–280.

GHOSE, K. AND KAMBLE, M. 1999. Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers, and Bit Line Segmentation. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*. IEEE/ACM, New York, 70–75.

HUANG, M., RENAU, J., YOO, S. M. AND TORRELLAS, J. 2001. L1 data cache decomposition for energy efficiency. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*. IEEE/ACM, New York, 10–15.

LARIN, S. Y. 2000. Exploiting program redundancy to improve performance. *Ph.D. Thesis*. ECE Dept., North Carolina State University, Raleigh, NC.

LIPASTI, M. H., WILKERSON, C. B., AND SHEN, J. P. 1996. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 138–147.

ONDER, S. AND GUPTA, R. 1998. Automatic generation of microarchitecture simulators. In *Proceedings of the International Conference on Computer Languages*. IEEE Press, New York, 80–89.

REINMAN, G. AND JOUPPI, N. 1999. An integrated cache timing and power model. *Tech. Rep.*. Western Research Lab.

STAN, M. R. AND BURLESON, W. P. 1995. Bus-invert coding for low power I/O. *IEEE Transactions on VLSI Systems 3*, 1 (Mar.), 49–58.

VILLA, L., ZHANG, M., AND ASANOVIC, K. 2000. Dynamic zero compression for cache energy reduction. In *Proceedings of the 33d Annual International Symposium on Microarchitecture*. IEEE/ACM, 214–210.

ZHANG, Y., YANG, J., AND GUPTA, R. 2000. Frequent value locality and value-centric data cache design. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 150–159.

Received January 2002; revised June 2002; accepted July 2002