ABSTRACT

We propose a low-leakage cache architecture based on the observation of the spatio-temporal properties of data caches. In particular, we exploit the fact that during the program lifetime a few data values tend to exhibit both spatial and temporal locality in cache, i.e., values that are simultaneously stored by several lines at the same time. Leakage energy can be reduced by turning off those lines and storing these values in a smaller, separate memory. In this work we introduce an architecture that implements such a scheme, as well as an algorithm to detect these special values. We show that by using as few as four values we can achieve 18.45% leakage energy savings, with an additional 13.85% reduction of dynamic energy as a consequence of a reduced average cache access cost.

Categories and Subject Descriptors:
C.3 SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS
B.3.2 MEMORY STRUCTURE, DESIGN STYLE, CACHE MEMORIES

General Terms: Design, Experimentation

Keywords: Leakage power, Caches, Architecture.

1. INTRODUCTION

The percentage of silicon real estate dedicated to embedded memories is steadily increasing: as of today, embedded memory occupies, on average, more than 50% of the active area in large SoCs [1]. At the same time, leakage (static) current is a growing concern in nanometer technologies, especially for densely packed structures, like memories. These two compound effects have triggered a flurry of research activity on circuit, logic, architectural and system-level techniques to reduce memory static current and power.

Power consumption (both static and dynamic) is a serious issue especially for cache memories. Several solutions targeting dynamic and static power reduction for cache memories have been proposed in the last few years ([2]-[14]). Dynamic power reduction is mainly based on two main ideas [3]: (i) reducing the cost of the average memory access by careful design of the memory hierarchy, (ii) reducing the number of memory accesses by careful design of the access patterns or of the density of the information transferred to and from the memory.

Static power has been tackled mainly by using power management techniques, that rely on special circuit arrangements in memory arrays that allow low-leakage modes of operations, which imply either loss of memory state [5, 6] and/or increased access time [7]. Quiescent cache blocks can be transitioned to a low-leakage mode for static power savings, at the price of some marginal increase of miss rate (for non state-preserving modes), or of an increased memory access time (when exiting state-preserving modes).

In this paper, we present a technique for static (and dynamic) reduction based on the observation that the distribution of long-lived values in a cache is highly non uniform. In other words, there is a small set of bit patterns which tend to occupy many cache locations for a long time. Intuitively, this can be seen as an implication of well known principles, namely spatial, temporal and value locality, which have been analyzed and exploited for various cache optimizations [9, 13, 12]. However, we exploit spatio-temporal value locality (STV-locality hereafter), which is stronger than spatial, temporal or value locality alone. Our explorative analysis shows that there is a significant amount of STV-locality, and that it can be efficiently captured by address and data profiling on a suitably modified functional cache simulator.

We propose a technique that aggressively “compresses” bit patterns of a given size with high STV locality (pre-stored in a small index memory), called ST-values, and turns off (in a non-state-preserving state) portions of the cache that contain them. The idea is that if there exist multiple copies of the same value in the cache, all of which leak during the same interval, leakage can be reduced by shutting off those lines, making only one copy of the value and store it elsewhere. Notice that our technique does not increase the miss rate since there is no need to re-activate values that have been turned off.

Two features are worth being emphasized about the proposed techniques. First, in its base implementation, it relies on the profiling of a given application, and is targeted for application-specific embedded systems. However, as we will show, in many cases it is possible to relax the dependency on a specific application. Second, our approach can be applied in conjunction with other leakage reduction techniques (e.g., the drowsy cache [7]), for additional power savings. Our experimental results show that leakage savings of 36% on average can be achieved, on a set of standard embedded benchmarks. These savings come with a savings of dynamic energy of 21%.
2. PREVIOUS WORK

Since leakage power has emerged as a critical design issue, researchers have focused on the reduction of memory leakage, because of two main reasons. First, memories usually take a large fraction of the chip area; second, the structure of storage cells is intrinsically high-leakage because of the lack of stacking effect.

Several architectural solutions ([8]–[12]) have thus been proposed for the reduction of leakage power in memories, and more specifically in caches. These schemes are all based on a similar paradigm, that is, to shutdown “suitable portions” of a cache, and differ by (i) how portions are defined, (ii) the definition of “suitable”, and (iii) the way the shutdown is implemented. Concerning the latter, the circuit-level solutions for controlling leakage can be broadly classified as non-state-preserving ([6, 5]) or state preserving ([7]), depending on whether they destroy the cell contents or not.

One approach consists of dynamically resizing a cache by monitoring the miss rate, and shutting off (in a non-preserving state) part of the sets according to the dynamic behavior of the application [8].

The cache decay scheme applies shutdown at the line granularity, turning off lines that have not been accessed for a given period [9], that can be evaluated through different algorithms. In [11], a method is proposed to dynamically determine the time interval for deactivating cache lines.

The drowsy cache [7] is the first solution that exploits state-preserving circuit techniques to reduce leakage. Here, the problem is transformed into a dynamic voltage scaling one, since the decision is whether to lower cell voltage rather than shutting off the line. While non-state-preserving techniques necessarily imply a performance penalty (due to extra misses for accesses to lines that are turned off), state-preserving solutions allow to tradeoff reduced leakage savings for keeping performance unchanged.

The approach of [12] relies on a “vertical” definition of the portion to be shut off. Based on the observation that few frequently occurring values often occupy a large portion of the data cache ([13]), it is possible to choose a small set of these frequent values and encode it using a reduced number \(n\) of bits. The data array is thus partitioned into two arrays, the first containing the \(n\) LSBs of the data, the other containing the rest of the bits. When accessing a frequent value, only the reduced size array can be used, while the other one is shut off (in a non-state-preserving mode).

3. LEAKAGE REDUCTION USING STV LOCALITY

The discussed leakage reduction technique stems from the joint observation of two properties of caches. First, as observed in [9], the lifetime of a cache line can be used to selectively shut it down so as to reduce its leakage. Second, as analyzed in [14], a small number of distinct values that frequently occur in a cache are also widely distributed (i.e., exist in multiple copies) in the cache itself. These properties can be seen as two different perspectives of a more general problem: exploiting lifetimes of the lines uses the temporal behavior of values, whereas exploiting replicated values uses the spatial behavior of the cache. The objective of this work is to combine the two dimensions into a unified approach.

Our technique is based on the detection of a small set of values, which are widely distributed in the cache, and whose lifetimes are as much overlapped as possible. These values, called spatio-temporal values (STVs), once identified, are stored into a separate array (the STV buffer), thus allowing to completely turn off (i.e., in a zero-leakage mode) the portions of the cache data array that contain STVs.

The detection of STVs is based on a profiling of the sequence of memory accesses of the processor, and attempts to maximize the probability of having a value replicated in the cache for the longest possible amount of time. Further details on how STVs are chosen are discussed in Section 4.3.

Clearly, since the structure used to store the STVs will also leak during idle cycles, there is a tradeoff between how many STVs can be stored and their distribution in the cache. While the number of STVs directly impacts the complexity of the STV buffer, the distribution of STVs in the cache is affected by their granularity, i.e., its size. Therefore, a preliminary analysis of the various options is required in order to come up with a leakage-efficient architecture.

Two important issues are worth being emphasized. First, freezing STVs in the cache does not cause a performance penalty. Unlike solutions based on shutdown, in which the re-activation of a line requires a few cycles of latency, our scheme simply turns off STVs which are never read from the cache, but always from the STV buffer. Therefore, there is no increase in the miss rate since we do not need to reactivate STVs in the cache.

Second, our optimization is not an alternative to schemes like the cache decay or drowsy cache, which can rather be used together; in fact, such power management solutions can in fact be applied to lines that do not contain any STV, or to non-STV values (if these techniques are applied using a smaller granularity).

3.1 Design Space Exploration

In order to evaluate the optimal tradeoff between the energy cost of the overhead and the distribution of STVs in the cache, we have carried out a simple exploration of the design space of the STV cache architecture. To do this, we need to approximately evaluate the potential leakage energy savings that can be achieved by our scheme. A technology-neutral measure of leakage can be given in terms of bits per cycle.

Consider then a cache with \(L\) lines of \(p\) data bits each. Let \(N\) be the number of STVs, each having size \(m\) bits (\(m \leq p\)), and \(t\) the number of tag bits (which depends on the cache parameters and its associativity). \(N\) and \(m\) are the two parameters of the space.

The quantification of the overhead is done as follows. We assume that the identification of the STVs is done through a set of index bits (STV index), that are contained into a separate array (the overhead array) with an entry for each cache line. The number of the STV index bits is thus \(\log_2 N\cdot(p/m)\), which increases for smaller values of \(m\). Besides the STV index bits, the overhead array stores an extra bit (flag bit) per each STV in a line, that is used to signal whether that STV is present in that line. There are thus \(p/m\) such flag bits, resulting into an overall overhead of \(v = (1 + \log_2 N)\cdot p/m\) bits per line. We denote by \(t' = t + v\) the sum of the tag and overhead bits. Furthermore, let \(q = N\cdot m\) the total equivalent number of bits in the STV buffer.

Finally, let \(C\) denote the total number of execution cycles of the application. Leakage energy for the STV cache can be approximately expressed as follows:

\[
E_{\text{leak,STV}} = C(L \cdot t' + (L \cdot p - k \cdot m) + q)
\]
The expression contains three terms. The first one (\(L_1\)) refers to the leakage in the tag plus overhead portion. The second one refers to the leakage of the data portion (\(L_p\)), from which we subtract the number of values which are shut off; this is accounted for by introducing the quantity \(k\), that represents the average number of STVs in the cache per cycle. The third term expresses the leakage in the STV buffer. Notice that this approximate formulation is based on the assumption of a perfect cache, i.e., zero miss rate.

In the case of a regular cache, leakage is simply given by

\[
E_{\text{leak,reg}} = C(L \cdot (t + p)).
\]

We have then derived the value of \(k\) from simulation of a set of typical embedded applications (the PowerStone [16] suite) for a 8KB cache with 16B lines, for different values of \(N\) and \(m\). The STVs have been chosen as described in Section 4.3. Table 1 shows the corresponding savings. We notice two main facts. First, a granularity of \(m = 32\) (i.e., one word) is consistently more effective than other values. This is due to a higher occurrence probability of a STV in the cache. Second, the leakage reduction of a word-based scheme is relatively insensitive to the value of \(N\), indicating that increasing the number of STVs also increases the distribution of STVs in the cache.

However, we should consider that dynamic energy also plays a role in choosing the value. While the average load of a word-line is decreased (because some portions of the data are shutdown), at each memory reference the STV buffer is accessed. Therefore, taking into account also dynamic energy, we ended up in choosing the following optimal design point: \(N = 4, m = 32\). These values will be used hereafter to discuss the details of the proposed architecture.

### 4. STV CACHE DESIGN

#### 4.1 Architecture

Figure 1 shows a conceptual architecture of the STV cache, for the optimal configuration described in the previous section. Without loss of generality, we will also assume that cache lines consist of 4 words (i.e., \(p = 128\) bits). Some details (such as tag match circuitry, output buffers and sense amps) are abstracted away for clarity purposes.

Line \(j\) is an example of a line containing STVs, in which two of the four words (shown with dashed pattern) are turned off (in a low-leakage mode); in a regular, non-STV line such as Line \(i\), the entire line will rather be active (and thus leak).

Figure 2 shows the structure of the overhead bits of a generic line. There are four blocks of 3 bits each; the first bit indicates whether the corresponding word is a STV, and the remaining two bits store its STV index. If one of the words is not a STV, the flag bit is set to 0 (and the meaning of the two STV index bits becomes irrelevant).

The block labeled STV buffer has two main functions. First, it stores the \(N\) STVs, that should be selected using the STV index bits. Second, it implements a lookup mechanism to check whether a fetched (on a miss) or a newly written value (for writes) is an STV or not. The most intuitive implementation of the STV buffer is thus by means of content-addressable memory (CAM), which allows both regular access (using an address) and an associative one. Figure 3 shows the details of the implementation of the STV buffer.

Besides the CAM, the STV buffer also includes two MUXes. The first, driven by the address offset bits, selects the proper STV_index and flag bit out of the four ones read out from the overhead array (read or write hits are in fact referred to a specific word). The second is driven by the match output of the CAM, and detects whether to send (during a read miss) to the cache output the fetched word (data_in) or the value (STV_i).

The presented architecture corresponds to the cache blocks which are direct-mapped. In order to extend the architecture to associative caches, we need to duplicate the same structure with overhead and STV buffer for each way of the cache.
4.2 Cache Operations

A STV cache deviates from the behavior of a regular cache when one of its lines contains one or more STVs (for granularity smaller than line size, as in our case). This piece of information is provided by the bits stored in the overhead array.

On a cache access, two operations are carried out in parallel, for any access: first, a line of the overhead array (flag and STV_index bits) is read out; second, the tag bits are read and matched against the address tags. The subsequent operations depend on the outcome of these two matchings and on the type of access (read or write). We identify five different cases corresponding to slightly different behaviors. In the following, for the sake of simplicity, we will assume a direct-mapped cache, using write-through with write no-allocate policy.

1. Hits
   
   (a) Read, non-STV: In this case the cache behaves as regular cache when a read hit occurs.
   
   (b) Read, STV: In this case, using the offset of the memory address, we identify what STV we are reading, and read out the corresponding STV from the CAM to the cache output.
   
   (c) Writes: This case is similar to a normal write hit, with one difference. If the value we are writing in the line is a STV, we must update the STV index bits and set the flag bit to reflect the change. Here, we must check if the newly written word is a STV. This is done by accessing the CAM in the lookup mode. If there is a hit, we read out the index and write it in the corresponding location in the overhead array (signal update_STV). Otherwise, the word is not a STV and we simply write data_in into the cache, as in a normal write, and we reset the corresponding flag bit.

2. Misses

   In the case of misses the differences occur during the refill of the missed line. Since our architecture is based on a 32-bit bus, the cache refill will take four cycles, each one corresponding to the writing of a single word in the replaced line, similarly to the case of a write, yet repeated four times.

   (a) Read: This case is similar to a regular write hit; that is, all the four words should be checked whether they are STVs or not. There is one difference though. One of the four words will also need to be sent to the cache output (data_out), depending on the address offset. If it is a STV, we need to read out the value (STV(val)) from the CAM. Otherwise, just update the flag and STV_index bits and proceed as a normal read.

   (b) Write: This case does not involve the STV buffer, since the cache uses write no-allocate policy for misses. Therefore, this is identical to a miss in a regular cache.

4.3 Choice of STVs

One key issue in the implementation of the STV scheme is the identification of the STVs. To this purpose, one important issue to be emphasized is that it is the distribution of values which is important, regardless of their frequency. This is somehow in contrast to what reported in [12], where occurrence frequency (i.e., a measure of "typical" behavior) is used to increase the shutdown potential of a cache line.

It is easy to show that occurrence frequency is not directly related to potential opportunities for shutdown; for example, one line containing a non-frequent value (say, occurring only once) which stays in the cache without being ever replaced would be the ideal candidate for shutdown.

As already mentioned in Section 3, STVs are selected according to criteria driven by (i) the spatial distribution of a value in the cache (how many copies of a value are in the cache at a given cycle), and (ii) the temporal distribution of a value (for how long a value will reside in the cache). It is the joint maximization of these two measures that is used to determine STVs.

The idea is therefore that, if there exist multiple copies of the same value, all of which leak at the same time, by making only one copy of the value in another place and shutting down all those lines, leakage energy can be reduced.

In order to exploit this fact, we need to find out the time intervals during which multiple cache lines are holding the same value, and hence are the potential candidates for being copied somewhere else, and shutting down the lines which are holding these values. It is intuitive that more widely distributed a values is, (i.e., the more the number of lines in which a particular value exists at the same time), the better the candidate. This suggests that we could assign a weight to a given value based on the overlapping of its lifetimes over different cache lines.

**Example 1.** Consider a given value (word), and let us track the lifetime of that value for all the cache lines. We represent lifetimes in terms of waveforms, where "high" ("low") indicates that the value exists (does not exist) in that line at a given time. Consider then the fragment shown in Figure 4.

*Figure 4: Value Lifetime on Different Lines.*

*During the time interval $T_2$ the same value exists in three distinct lines. Hence copying it to another place (i.e., the STV buffer) will save $2T_2$ processor cycles of leakage energy (assuming that the place where the value is copied also leaks the same amount). Similarly, during the time intervals $T_3$, the value exists across only two different lines, hence the advantage of copying will be $T_3$ processor cycles of leakage energy. The same holds for $T_4$. In the interval $T_5$, on the contrary, the value exists only in one line so there is no advantage in copying it elsewhere. Summing all the contributions we determine a weight for the value of $2T_2 + T_3 + T_4$.*

*Notice that the weight has a "physical" meaning, since it is actually a measure of cycles; if we multiply this value by the leakage energy per word we obtain actual energy figures.*

Given a set of memory references, STVs are thus profiled and assigned a weight as described above. The $N$ STVs with largest weight are those that are then selected and stored into the STV buffer.
5. EXPERIMENTAL RESULTS

5.1 Experimental Setup
In order to do application profiling we used MPARM [15], a multi-processor platform based on ARM7 cores. We have configured MPARM as a single processor platform in order to run our experiments. We used the PowerStone benchmarks [16] that consist of various widely used embedded kernels. All the experiments are done with a 8KB cache with 16-byte line size, with associativity of 1 (direct mapped), 2 and 4 for three different configurations, using write-through policy with write-no-allocate.

5.2 Energy Models

5.2.1 Leakage Energy
Leakage is usually specified in terms of energy per bit per cycle [7, 8]. This metric is suitable for our purposes, since the implementation of the STV cache mostly relies on the addition of extra memory cells (in the overhead array and the CAM).

The estimation of leakage is thus done by means of a formulation similar to that of Section 3.1, yet using the profiling information obtained from simulation. This implies to consider misses and their relative costs, and the actual number of cycles in which the cache is idle. Therefore, our measure of leakage will be \( \text{bits per cycle} \).

One important observation concerns the CAM. Although it stores bits, the CAM cell is slightly more complex than the standard SRAM one. Figure 5 shows the structure of the CAM we are considering. This cell contains three extra transistors that form the comparator (M1, M2, and M3); the first two are connected to Search Line(SL) and complement of SL, while the third drives out the match line.

![Figure 5: Structure of the CAM Cell.](image)

Since none of the transistors is directly connected to power supply, during an idle state there will be almost no leakage; therefore, we can safely assume that the leakage in the CAM cell will be almost equivalent to the leakage in SRAM cell. This statement is supported by data provided by industrial memory vendors, as well as by published data [17].

The leakage overhead in the CAM is in any case marginal, since it contains only \( 4 \times 32 = 128 \) bits overall. Most of the leakage overhead is due to the 12 overhead bits that are added to each cache line.

5.2.2 Dynamic Energy
Concerning dynamic energy, we leverage a traditional model which computes energy per access. Clearly, such a model needs to be parameterized with respect to the cache size and the type of operation (read/write).

We thus assume a generic model \( E(W, H, \text{op}) \) for a memory array of width \( L \) bits, height \( H \) words (of size 32 bits), and where \( \text{op} = \{\text{read,write}\} \). The model is derived from interpolation of data (through least-mean square regression) extracted from a memory generator by STMicroelectronics in a 0.13 \( \mu \text{m} \) technology. We opted for an empirical model because it allows us to account for the design rules of a memory array, which tends to keep the aspect ratio as square as possible. For example, a 1-bit array will actually be synthesized by the generator into an array with number of rows and columns as similar as possible.

For the CAM we have used a similar model, with the difference that the “match” operation must also be characterized. Based on data obtained with the same memory generator we quantified the cost of a match access for a CAM array to be about three times the cost of a SRAM. This is due to the fact that matching requires the activation of all the 4 wordlines. Again, although significantly higher than that of a regular access, we incur this cost only during writes or misses (Section 4.2).

5.3 Energy Results
Figure 6 shows the leakage savings for the PowerStone benchmarks, for the above mentioned cache configuration, and for 4 STVs. We notice significant variations of the savings, with peaks over 33%. The average saving is 18.45%.

![Figure 6: Leakage Energy Savings.](image)

Figure 7 shows the dynamic energy results for the same set of benchmarks. The average saving is 13.85%, but in one case leakage is slightly increased.

One interesting observation comes from the nature of the savings in dynamic energy. In the STV cache, dynamic energy overhead is dominated by the cost of the bits in the overhead array, since they are read out on each read operation. Therefore, the more we access the STV buffer, the more dynamic energy we save. The increase of dynamic energy of the \( \text{engine} \) benchmark is due exactly to this reason. Notice that in case of \( \text{jpeg} \) the dynamic energy drop downs to 0. This is due to the fact one of the STVs resides in cache for long time but with very less access frequency. Hence the overhead gets larger. Increasing the frequency of access to
the STVs is the objective of the FV cache [13]; this demonstrates that value locality alone is meaningful for dynamic energy, but not for leakage.

5.4 Impact on Performance

Concerning the performance overhead, there are two issues to consider. First, we have a larger tag array; second, for each access we also need to access STV buffer. For timing related figures we refer to [18], which has detailed simulated figures for both delay and energy for caches. We treat these two issues separately.

Since we add 12 bits for every tag row we increase the delay of the wordline driver. The work of [18] shows that the delay of wordline driver with 16 bits and 32 bits is 100 and 120 pS respectively (based on HSPICE simulations). In our case we increase tag size, per row, from 19 bits to 31 bits. And hence, conservatively, we can assume that there is an increase of 20 pS in tag access time.

After the tag readout, we need to perform STV buffer access. Since tag readout is followed by tag comparison, the available slack time is equivalent to the delay of the comparison operation. Once again we refer to [18], where the reported comparison time for 19 bits comparator is 40 pS. The cost of wordline driver delay is around 200 pS. This is the time slack available for performing STV buffer access. STV buffer behaves same as a normal SRAM when read access is performed hence we model the STV buffer as an SRAM (for reads). In [18], authors have reported delay for some small memory arrays and we use that to scale delay to our 32x1 STV buffer. With our configuration of STV buffer of 32 columns the wordline driver delay is around 120 pS. Data path delay including bitlines and sense amplifiers for 4 rows can be estimated at 80 pS (in [18] reported data path delay for a memory of 128 columns and 16 rows is measured to be around 250 pS).

For the row decoder delay we have a very simple 2-to-4 row decoder, which has delay of the order of 100 pS. Hence the total delay associated with STV buffer read is around 300 pS, about is 40 pS more than that of the comparator.

Summing the two effects, the overall read access delay increases by 60 pS. Assuming a clock cycle time of 2 ns (typical of most embedded processors), and a 1-cycle access time for the cache, this overhead corresponds to about a 60/20000 = 1.2% penalty.

6. CONCLUSIONS

Joint exploitation of spatial, temporal and value locality in data caches offers opportunities for leakage and dynamic power reductions that cannot be achieved by considering them individually. We introduced architectures and algorithms to exploit this spatio-temporal-value locality for implementing it into what we called the STV-cache. The proposed architecture provides 18.45% leakage energy savings, with a 13.85% concurrent reduction of dynamic energy, with no increase of miss rate.

7. REFERENCES


