Increasing Cache Efficiency by Dead Block Prediction and Elimination

Joshua Linge
Department of Computer Science
University of Central Florida
Orlando, United States
jlinge@knights.ucf.edu

Basundhara Dey
Department of Computer Science
University of Central Florida
Orlando, United States
basundhara@knights.ucf.edu

Anjana Mishra
Department of Computer Science
University of Central Florida
Orlando, United States
anjana.mishra0@knights.ucf.edu

***Abstract*— This report elaborates on how exactly we improve cache efficiency by eliminating the dead blocks using Trace based dead block predictors. Also the counter based dead block predictor is discussed and all the results are finally compared on the basis of both the reference history and the burst history.**

***Index Terms*—Cache efficiency, Dead block, Trace based predictor, Counter based predictor.**

I. Introduction

No ideal state of 100% efficiency has ever been achieved by any existing machine that we know of, but the quest for improvement and betterment is always on the race. We know that cache efficiency plays a very important role in the overall efficiency of the system, thus improvising the cache efficiency will lead to the overall efficiency of the system. We know that there are some data present on the cache line which are not even accessed once before the whole block eviction, such data are called Dead Blocks and are a big reason of reduced cache efficiency. It is quite conspicuous that as the dead block reside in the cache line it takes up a lot of space which could have been used effectively by a Live Block (data which is accessed frequently before the block eviction) and the time for which a dead block reside in the cache line is much more than the time slot the live block occupies thus for less than half of the time only the cache is holding useful data. Thus we see that only due to the problem of existence of dead block we are having huge reduction in the cache efficiency. Thus in order to improve the situation we can do a control by predicting and eliminating the dead block. If we are able to predict a dead block on time and then we can eliminate the dead blocks and free up the time slot for the live block. Thus after doing this the majority of the data in the cache line will be live block having frequent accesses and this will highly increase the efficiency of the cache, serving our purpose to its best.

II. Dead Block and Cache efficiency

A cache line which will not be referenced before eviction is called a Dead Block and the cache line which will be accessed before the eviction is called as Live Block. Now consider the diagram given below.

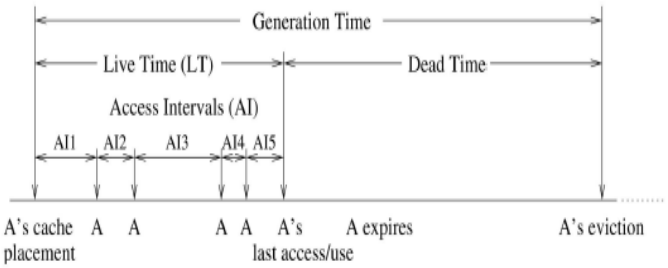

Figure 1. The life cycle of a cache line A.

Here we see that the time from when a block is placed on the cache line to the time when the block is evicted from the cache is known as generation time. The generation time is further divided into live time and dead time, where it is quite clear that live time is the time for which the cache has live block on the cache line where as dead time is the time slot for which the cache has dead block on the cache line. Generally we see that the dead time is of the order of the metric which is more than that of the live time. This is the exact problem we are solving. By removing the dead block we will reduce the dead time, and minimizing the dead time will eventually increase the live time. This finally increases the cache efficiency. The average fraction of the cache blocks that store live data is given by the following formula.

$$E = \frac{\sum_{i=0}^{A \times S - 1} U_i}{N \times A \times S}$$

Here we have A as associativity. S is the number of sets.
N is the total number of execution cycles. Ui is the number of cycles for which the block i contains live data. As we see that the value of E determines the average fraction of the cache block that stores the live data, we can conclude that this parameter is directly proportional to the cache efficiency.

III. Dead block prediction

When predicting dead blocks, one important point that is to be taken care of is that the dead block should be predicted only when the prediction can be fruitfully use, for this we have the following conclusions. Replacing dead lines promptly after their last use would increase cache performance by making the wasted capacity available for other cache lines that are not dead yet but for this the identification of a dead block should be done between the last access to the block and its eviction from the cache. If it is not done so and the prediction is made after a long time after the last access to the dead block then the prediction will be of no use and if we do the elimination at that time then probably we will save very less time for the live block to take place, which will not give us a good amount of efficiency and whole of the effort will be fruitless. Thus we now know that for dead block prediction to make sense and to be useful for dead block elimination, the prediction has to be made as soon as possible after the last access to the dead block.

IV. Reference History or Burst History

With every prediction the data is entered in the history table and there are two different ways in which a prediction is stored in the history.

- Reference History
- Burst History

Now to understand these two different types refer to the diagram given below.

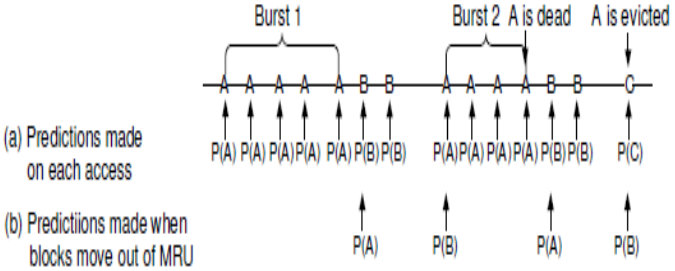

Figure 2. Predicting dead blocks at different times.

As we see in the diagram there are frequent predictions made which is on every access, when the history stores such prediction, it is known as reference history. Considering the other line of predictions shown below, we see that the prediction is made after every change (i.e. from A to B), or we can say when the block moves out of the MRU, thus this reduces the number of predictions to be stored in the history, and history stored in this manner is called as burst history. As we can easily see the number of predictions for the burst history will be much less than that of the reference history. This would also help in reducing the large power consumption we had with reference history.

V. Types of Dead Block Predictors

Dead block prediction can be performed both in software and hardware. Previous works proposed several dead block predictors and applied them in solving prefetching and block replacement. Three of them are most efficient: Trace-based, Counting-based and Time-based. On the basis of information collected through hardware profiling or compiler analysis, software solutions pass hints about dead block predictors. Though they are more accurate but mostly have very poor coverage.

Two kind of hardware solution classification can be concluded: data address based and PC based. PC based approach require lower storage overhead with compare to data based. To make prediction, the predictor maintains some state. On based of that, we classified the Dead block predictor into the above mentioned three categories.

A. *Trace Based Dead-Block Predictor*

Lai et al. proposed a concept of dead-block prediction for the L1 cache which records PC traces that result in a dead block. Lai et al. also proposed the dead-block correlating prefetcher. This prefetcher makes a prediction to determine if a block is dead and also what address should be prefetched into the dead block. The dead-block predictor and the dead-block correlation prefetcher rely on the repetitive behavior within programs to work effectively. Instead of using the memory access sequences, they instead use the instruction sequences to make predictions about memory behavior.

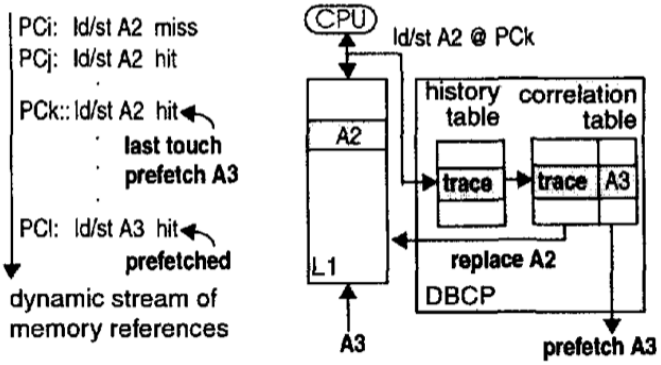

Figure 3. A dead-block correlating prefetcher.

In the example, we are given a sequence of memory accesses into a block frame. Address A2 was accessed from PCi and PCj and is now being accessed by PCk. After encoding the PC trace, A2 is predicted dead and the predictor suggests address A3 to be prefetched and stored into A2's block frame. Later when A3 is accessed, it will be a hit due to the prefetch.

1. Dead-Block Predictor

A Dead-Block Predictor consists of two major pieces: the history table and the dead-block table. The history table has an entry for each block in the cache and stores an encoded trace for that block. The dead-block table is a large table that stores

encoded traces that result in a dead block. Upon an access to a memory address in the L1 cache, the address's block access trace is checked from the history table. This trace is then encoded with the PC of the current load or store and stored back in the table. The dead block table is then accessed using this new encoded trace and the memory access address. If the table contains the encoded trace, then the address's block is predicted dead.

Lai et al. tested multiple methods for encoding block traces and concluded that a truncated addition of the PCs created suitable traces. To keep these traces compact, 12 bits were used. Another encoding method tested was xor, but due to multiple accesses from the same PC, xor was found to not properly encode traces.

The dead-block table is used to store previously seen traces that result in a dead block. This table should contain as many entries as possible so it will be a large and highly associative structure. To access this table, the load or store address will always access the same location independent of the trace encoding. The address and the trace encoding must both be considered when accessing the dead-block table. Lai et al. suggested xoring the address and the trace encoding was a hash function to spread the entries around.

After every memory access to the L1 cache causes a look up into the dead-block table to check if the current block will be dead. In order for the predictor to be useful, it will need to know when a block is dead and add its trace to the table. When the L1 cache replaces a block, the replaced block is dead since it is being evicted. For every replacement, the replaced block's encoding is added into the dead-block table. In order to reduce the number of mispredictions, each entry in the dead-block table has a saturation count. The count is incremented when adding a trace that is already stored in the dead-block table for a given address. If a block is used again after it has already been predicted dead, the count is decremented. Lai et al. suggested that a 2-bit counter is enough for accurate predictions. The counter helps eliminate traces that appear infrequently during program execution.

In figure 4, you can see the design of the Dead-Block Predictor. In the example used previously, the history table stores the encoded trace of PCi and PCj, which are previous PCs that accessed memory address A2. A2 is being accessed again and PCk. PCk is encoded with the previous trace encoding. This new encoding is then xored with A2 to access the dead-block table. One of the entries at this location stores same encoding and thus A2 is predicted dead.

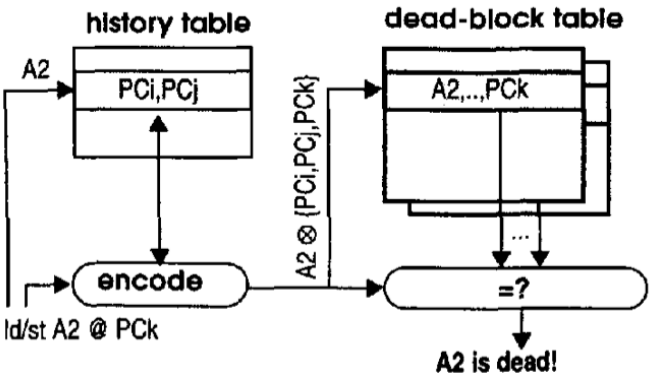

Figure 4. A dead-block predictor.

2. Dead-Block Correlating Prefetcher

The idea of the Dead-Block Correlating Prefetcher is to correlate dead block traces to a memory address. A Dead-Block Correlating Prefetcher is essentially a Dead-Block Predictor with an additional value in the dead-block table entry to store a predicted address to prefetch into the dead block. In order to increase address prediction accuracy and coverage, each entry in the history table and dead-block table contains the previous address mapped to a block that resulted in a dead block. This improvement of accuracy and coverage comes as a cost of a higher storage overhead. More than one address mappings can be stored but Lai et al. concluded that two previous mappings are enough for high accuracy and coverage.

In figure 5, you can see the design of the Dead-Block Correlating Prefetcher. In the example used previously, the history table stores the encoded trace of PCi and PCj, which are previous PCs that accessed memory address A2, as well as the previous address mapped to the block frame, A1. A2 is being accessed again and PCk. PCk is encoded with the previous trace encoding. This new encoding along with the address mapping is then xored with A2 to access the dead-block table. One of the entries at this location stores same encoding and address mapping and thus A2 is predicted dead. A3 is then prefetched and stored in the block frame.

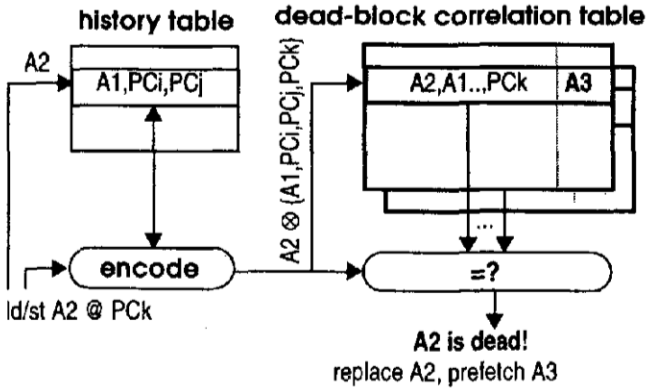

Figure 5. A dead-block correlating address prefetcher.

B. Counter Based Predictor

Kharbutli and Solihin introduced counting-based predictors for the L2 cache. It comes with the conception of The Livetime Predictor (LvP). LvP tracks both the times the block has been referenced and the PC of the instruction that first missed on the block. This value is stored in the predictor on eviction. A block is predicted dead if it has been accessed more often than the previous generation. Each predictor entry has a one bit confidence counter so that a block is predicted dead if it has been accessed the same number of times in the last two generations. In brief, if it reaches the threshold value. The predictor table is a matrix of access and confidence counters. The rows are indexed using the hashed PC that brought the block into the cache and the columns are indexed using the hashed block address. The counting based replacement policy chooses the predicted dead block closest to LRU as a victim, or the LRU block if there is no dead block. LvP and AIP (Access Interval Counter Predictor) are also used to bypass cache blocks, which are brought to the cache and never accessed again.

We choose the access interval and live time as our time intervals and call our predictors AIP and LvP, respectively. In terms of the type of events counted during the time interval for a cache line, possible choices are accesses to the cache, accesses to the set that has the line or accesses to the line itself. For AIP, a reasonable choice would be to count the number of accesses to the set that has the line. Counting the number of accesses to the cache (regardless of the sets) would be a poor choice due to requiring large counters to count possibly many accesses during a line's access interval, not to mention that accesses to other sets are a noise since they do not affect the line's live time or dead time. For LvP, a reasonable choice of event to count is the number of accesses to the line itself during a single generation. Counting the number of accesses to the cache or to the set that has the line would require much larger and frequent counting, which incurs a high storage overhead. With their respective event types to count, we found that both AIP and LvP only require four-bit counters per cache line to count their respective events that occur in their respective time intervals.

Through our project work, we mostly concentrate on AIP. For AIP to reap the benefit of early replacement of deadlines, its access interval threshold value must be considerably smaller than its dead time. A previous study on L1 caches has shown that, indeed, access intervals of a line are typically a lot smaller than its dead time [5]. In some previous experiments on L2 caches largely confirm the finding: The difference in a line's access interval duration and its dead time is often one or more orders of magnitudes (Fig. 6). This means that waiting for a typical access interval to elapse before concluding a line is dead will still allow the line to be replaced much sooner compared to when the line is replaced by LRU replacement.[2]

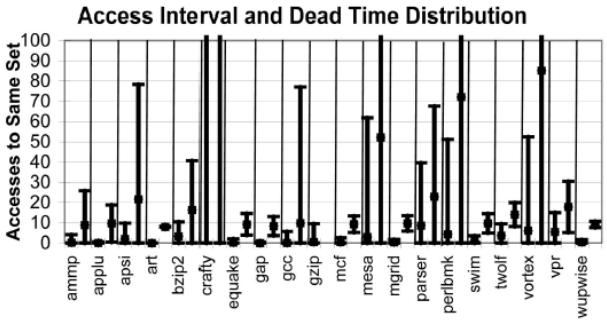

Figure 6. Access interval (left bar for each application) versus dead time (right bar for each application) distribution, showing the average plus/ minus its standard deviation. [2]

For some better understanding of AIP, we also look into the prediction structure of AIP. Fig 7 shows the prediction structures used by AIP. Several fields are added to each L2 cache line in order to keep a count of events and its threshold value, while a separate prediction table is added between the L2 cache and its lower memory hierarchy components to keep the history of threshold value for lines that are not cached.

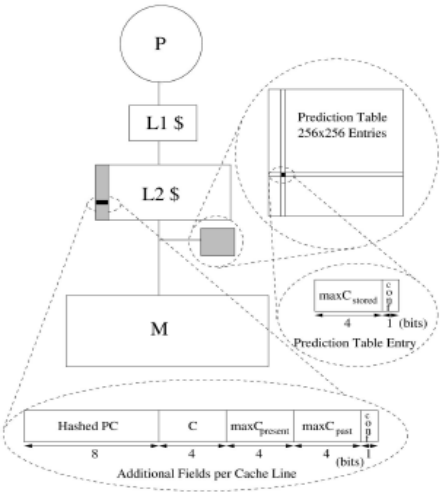

Figure 7: Prediction structure of AIP

So hashed PC, the first field is used to index a row in the prediction table when a line is replaced from the cache but has no role in counter bookkeeping. Then next is the event counter C, which is incremented each time the set that has the line is accessed, regardless of whether the access was a hit or miss. The third and fourth fields are the counter thresholds for the time intervals in the most recent generation and in the current generation. The fifth field is a single confidence bit that, if set, indicates that the line can expire and be considered for replacement. If it is not set, the line is not considered for replacement even if its threshold has been reached. Finally, to index a column in the prediction table, the line's block address is hashed by XOR-ing all 8-bit parts of it. For instruction lines, the block address is considered to be the PC.

C. Time Based Predictor

Hu et al. propose a time-based dead block predictor that learns the number of cycles a block is live and predicts it dead if it is not accessed for twice that number of cycles. This predictor is used to prefetch into the L1 cache and filter victim cache. Abella et al. propose a similar predictor based on number of references rather than cycles for reducing cache leakage.

VI. SIMULATION

SimpleScalar 3.0 was used for simulating an out-of-order processor. While the original dead-block predictor used the reorder buffer to check and create PC trace encodings in program order, we encoded the traces out of order during each cache access. The reason for this was that the history table contains a duplication of the L1 data cache tag array, but without major modifications, there was no easy way to keep the cache and the history table synchronized when the cache is access out of order and the history table is accessed in order. The standard LRU policy may cause different lines to be evicted with in order access and out of order access. In order to simplify things, each cache line also included a corresponding history table entry. The entry contained the PC trace encoding and the previous address that was stored in that block.

Processor	
Instruction issue/retire	4 instructions/cycle
Reorder buffer	128 entries
Load/store queue	16 entries
Prefetcher	
History table	512 entries
Dead-block table	2M entries 8-way associative
Cache and memory	
L1 data cache	16K, 32-byte block, 2-way associative, 1 cycle latency
L1 instruction cache	16K, 32-byte block, 2-way associative, 1 cycle latency
L2 cache	512K, 128-byte block, 8-way associative, 6 cycle latency
Main memory	18 cycle latency

Table 2. SimpleScalar simulation parameters.

The dead-block table was a similar structure to the cache. It had 2 million entries and was 8-way set associative. Each entry in the dead-block table contained a count, a valid bit, a trace encoding, and a prefetch address. Since each of these items corresponded to one entry, there were no offset bits used in the dead-block table access address. The address into the dead-block table was 32 bits long with 18 bits for the index and 14 for the tag. Lai et al. proposed that the address into the dead-block table could be a simple hash function such as xoring the encoding with the cache access address. After some testing, we came to the conclusion that this method of hashing caused a large amount of conflicts causing two different addresses and encodings to hash into the same address into the dead-block table. Instead of using a cache style set associativity where addresses with the same index but different tags would fit into the associative slots, if two addresses into the dead-block table were the same but each had different trace encodings, they would not be considered the same entry. The dead-block table used the LRU replacement policy.

After every cache access, the PC trace would be encoded and the dead-block table was accessed to see if the encoding was stored in the table. If the encoding was stored in the table, then the saturation count was checked. If the count were below 3, then the prediction would not be confident enough to declare the block as dead. If the count were equal to 3, the block would be declared dead. The prefetch address also stored in the dead-block table entry would then be prefetched and stored in the dead block's frame. Generally, a block's dead time is much larger than the block's live time. From this, we made the assumption that once a block is predicted dead, we can prefetch immediately. We do not add the access time and cycle time since the prefetch is executed in parallel with all other processor executions.

When a cache access resulted in a replacement, the previous encoding and address were saved and then used to add to the dead-block table. If an entry for this address and encoding was not already in the dead-block table, a new entry is allocated taking from the LRU entry. New entries set their saturation counts to 0 and the prefetch address to the address that replaced the block. If the entry was already contained in the dead-block table, the saturation count was incremented. One issue that was not discussed in the paper was when an entry in the dead-block table contained a different prefetch address than the address that replaced the block. Multiple methods were tested to circumvent the issue. One method involved simply replacing the previous prefetch address with the new address. Another method again replaced the prefetch address but also reset the saturation count to 0. The most effective method was to store multiple prefetch addresses and saturation counts for each address. The prefetch address with the highest saturation count was the address that was prefetched when a block was predicted dead. This method came with a higher storage cost for higher accuracy.

If a block was replaced and it did not have a valid encoding, then the block was never accessed by the program and was a bad prefetch. When this event occurred, the dead-block table is updated to remove the entry that caused the bad prefetch. The dead-block table was accessed using the previous address and encoding that was stored in the bad prefetched block.

The difference between a reference-based Dead-Block Correlating Prefetcher and a burst based Dead-Block Correlating Prefetcher is when the prediction is made. With reference-based predictor, a prediction is made after every cache access. With a burst-based predictor, the prediction is only made when a block moves out of the most recently used location. A MRU tag was added to each set in the L1 data cache. When making a prediction, check to see if the current address's

tag is equal to the MRU tag. If the tags are equal, do not make a prediction. If the tags are not equal, get the trace encoding for the previously MRU block and make a prediction.

VII. RESULTS

The simulator was then tested using the benchmarks GCC, BZIP2, and VPR. The simulation cycles can be seen in Table 3. There is a small speedup when using a dead-block predictor, but the speedup in this case does not outweigh the cost in creating the dead-block predictor. This may be result of small bugs in the implementation. Lai et al. concluded that using reference based dead-block correlating prefetcher could gain a speedup of 62% on average. If time permitted, the results could have been compared with cache that have an increased capacity equal to the space used by the predictors.

Simulation Cycles	GCC	BZIP2	VPR
No Predictor	1485011317	48875139362	83780415496
Ref Trace	1484027903	48702953522	83733972042
Burst Trace	1483994408	48700125498	83742842981

Table 3. Simulation cycles for the benchmarks.

Percent Execution	GCC	BZIP2	VPR
No Predictor	100.0%	100.0%	100.0%
Ref Trace	99.933%	99.648%	99.945%
Burst Trace	99.932%	99.642%	99.955%

Table 4. Percent of execution for the benchmarks.

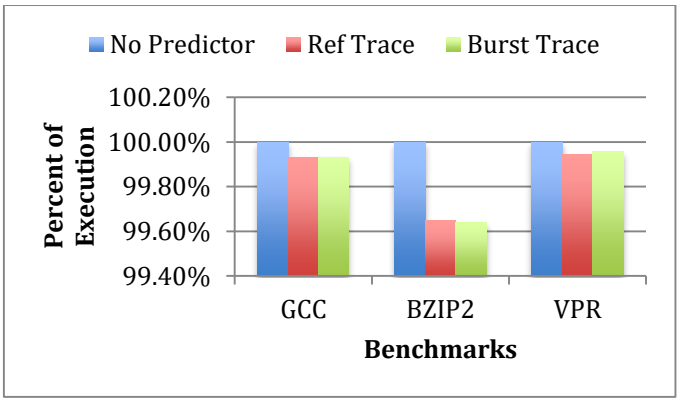

Figure 8. Graph of the percent of execution for the benchmarks.

VIII. CONCLUSION

Although we did not get a very large percentage of speed up but still with the results we got, we can say that the dead block elimination and has helped increase the cache efficiency, considering it to be directly proportional on the parameter E (average fraction of the cache blocks that store live data) we described above.

IX. References

[1] A-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 144–154, 2001.

[2] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4):433–447, 2008.

[3] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *Proceedings of the 29th International Symposium on Com- puter Architecture*, 2002

[4] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *Proc. of the 41st Int. Symp. on Microarchitecture*, pages 222–233, Lake Como, Italy, Nov. 2008

[5] G. Chen, V. Narayanan, M. Kandemir, M. Irwin, and M. Wolczko, “Tracking Object Life Cycle for Leakage Energy Optimization,”Proc. ISSS/CODES Joint Conf., 2003.