Efficient System-on-Chip Energy Management with a Segmented Bloom Filter

Mrinmoy Ghosh[1], Emre Özer[1], Stuart Biles[1], and Hsien-Hsin S. Lee[2]

[1] ARM Ltd.
{mrinmoy.ghosh, emre.ozer, stuart.biles}@arm.com
[2] School of Electrical and Computer Engineering, Georgia Institute of Technology
leehs@ece.gatech.edu

Abstract. As applications tend to grow more complex and use more memory, the demand for cache space increases. Thus embedded processors are inclined to use larger caches. Predicting a miss in a long-latency cache becomes crucial in an embedded system-on-chip(SOC) platform to perform microarchitecture-level energy management. Counting Bloom filters are simple and fast structures that can eliminate associative lookup in a huge lookup space. This paper presents an innovative segmented design of the counting Bloom filter which can save SOC energy by detecting misses aiming at a cache level before the memory. The filter presented is successful in filtering out 89% of L2 cache misses and thus helps in reducing L2 accesses by upto 30%. This reduction in L2 Cache accesses and early triggering of energy management processes lead to an overall SOC energy savings by up to 9%.

1 Introduction

The increasing complexity and shrinking feature size of present day microprocessors has led to energy becoming an important design constraint. Energy is more of an issue in embedded cores that are a part of System-on-chips (SoCs) for handheld devices, where the prime concern is battery life. However, also due to shrinking feature size designers have more transistors per die at their disposal. This has led to large caches, which are major consumer of both static and dynamic power in embedded SoCs. This paper presents an innovative design to help reduce energy consumption in caches and also the SoC platform comprising of the CPU and multi-level caches.

The memory hierarchy of most processors contains single or multi-level caches designed as SRAM memories followed by a large DRAM backstore. Since an access to DRAM memory may take 100s of cycles,therefore in in-order processors, and in some cases for out of order processors, severe stalls may occur on a cache miss in the cache-level before the DRAM. Hence, the cache miss event can be used as a trigger for several microarchitectural energy management processes in the SoC. The energy management processes may include but are not limited to putting all caches in a state preserving low power drowsy mode and for power gating all or part of the processor core.

Bloom filters are simple and fast structures that can eliminate associative lookup when the lookup address space is huge. They can replace associative tables with a simple bit vector that can precisely identify addresses that have not been observed before. This mechanism provides early detection of events without resorting to the associative lookup buffers. This has significant implications on the performance and power consumption considering the fact that Bloom filters are very efficient hardware structures in terms of area, power consumption and speed.

This paper presents an innovative segmented design of the counting Bloom filter that saves energy by detecting a miss in the cache level before the memory. The detection of the miss happens much earlier than the actual request reaches the particular cache. The early detection would allow the processor to make the energy management processes quite early in the memory hierarchy. Starting energy saving measures early provides more energy saving opportunities than in the case where the measures are taken after a miss in the lowest cache level is detected.

The rest of this paper is arranged as follows. Section 2 explains the basics of Bloom filters. Section 3 describes the novel segmented Bloom filter design and elucidates how it aids in saving energy. Then, Section 4 describes the simulation methodology and the energy savings obtained using the segmented Bloom filter and presents the experimental results. Section 5 discusses prior art. Finally, Section 6 concludes the paper.

2 Bloom Filters

The structure of the original Bloom filter idea as described by Bloom [1] is shown in Figure 1a. It consists of several hash functions and a bit vector. A given N-bit address is hashed into k hash values using k different random hash functions. The output of each hash function is an m-bit index value that addresses the Bloom filter bit vector of 2^m where m is much smaller than N.

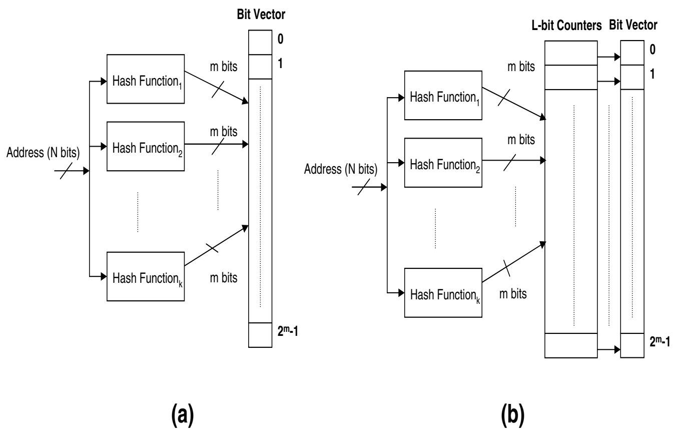


Fig. 1. **(a)** Original Bloom filter, **(b)** counting Bloom filter

Each element of the Bloom filter bit vector contains only 1 bit that can be set. Initially, the Bloom filter bit vector is zero. Whenever an N-bit address is observed, it is hashed to the bit vector and the bit value hashed by each m-bit index is set. When a query is to be made whether a given N-bit address has been observed before, the N-bit address is hashed using the same hash functions and the bit values are read from the locations indexed by the m-bit hash values. If at least one of the bit values is 0, this means that this address has definitely not been observed before. This is called a *true miss.* On the other hand, if all of the bit values are 1, then the address may have been observed but cannot guarantee it. If the address has not been observed but the bit vector indicates it does, this is called a *false hit.*

As the number of hash functions increases, the Bloom filter bit vector is polluted much faster. On the other hand, the probability of finding a zero during a query increases if more hash functions are used. The major drawback of the original Bloom filter is the high false hit rate because the filter can get polluted quickly and filled up with all 1s and it starts signalling false hits.

The original Bloom filter has to be quite large to reduce the false hit rate since once a bit is set in the filter there is no way we may reset it. So as more bits are set in the filter, the number of false hits increase. To improve performance of this kind of filter a mechanism for resetting entries set to one is needed. The counting Bloom filter as shown in Figure 1b is proposed by *Fan et al.* in [2], which aims at web cache sharing, provides the capability of resetting entries in the filter. For a counting Bloom Filter, an array of counters is added along with the bit vectors of the classical Bloom Filter. When a new address is observed for addition to the Bloom filter, each m-bit hash index addresses to a specific counter in an L-bit counter array. Then, the counter is incremented by one. Similarly, when a new address is observed for deletion from the Bloom filter, each m-bit hash index addresses to a counter, and the counter is decremented by one. If more than one hash index addresses to the same location for a given address, the counter is incremented or decremented only once. If the counter becomes non-zero, the bit in the Bloom filter bit vector addressed by the same m-bit index is set. If the counter becomes zero, then the bit is reset. Queries to a counting Bloom filter are similar to the original Bloom filter.

3 Segmented Bloom Filter Design

We propose an innovative segmented counting Bloom filter as shown in Figure 2 where the counter array of L bits per counter is decoupled from the bit vector and the hash function is duplicated on the bit vector side. The cache line allocation/de-allocation addresses are sent to the counter array using one hash function while the cache request address from the CPU is sent to the bit vector using the copy of the same hash function. The segmented Bloom filter design allows the counter array and bit vector to be in separate physical locations.

A single duplicated hash function is sufficient as our experiments show that the filtering rate of a Bloom filter with a single hash function is as good as the

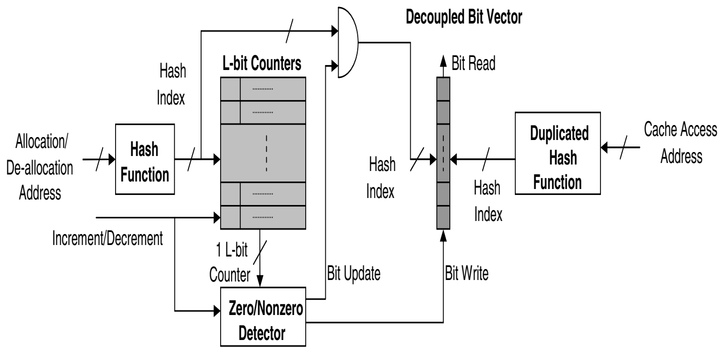


Fig. 2. Segmented Bloom filter

one with two or more hash functions. The implemented hash function chops the physical address into several chunks of hash index long and bitwise XOR them to obtain a single hash index. The number of bits needed per counter (L) depends on how the hash function distributes indeces across the Bloom filter. In the worst case, if all cache lines map to the same counter, the bit-width of the counter must be at most $log_2(Numofcachelines)$ to prevent overflows. In reality, the required number of bits per counter is much smaller than the worst-case.

The counter array is updated with cache line allocation and de-allocation operations. Whenever a new cache line is allocated, the address of the allocated line is hashed into the counter array and the associated counter is incremented. Similarly, when a cache line is evicted from the cache, its associated counter is decremented.

The counter array is responsible for keeping the bit vector up-to-date. The update from the counter array to the bit vector is done only for a single bit location if and only if the counter becomes zero from one during decrement operation or one from zero during increment operation. The following are the steps taken for updating the bit vector:

1) The L-bit counter value is read from the counter array prior to an increment or decrement operation.
2) The counter value is checked for a zero boundary condition by the zero/nonzero detector whether it will become non-zero from zero or zero from non-zero inferred by the increment/decrement line.
3) If a zero boundary condition is detected, the bit update line is asserted, which forwards the hash index to the bit vector.
4) Finally, the bit write line is made 1 to set the bit vector location if the counter will become non-zero. Otherwise, the bit write line is made 0 to reset the bit vector location.
5) If there is no zero boundary condition, then the bit update is not activated, which disables the hash index forwarding to the bit vector.

When the CPU issues a lookup in the cache, the cache address is also sent to the bit vector through the duplicated hash function. The hash function generates an index and reads the single bit value from the vector. If the value is 0, this is a safe indication that this address has never been observed before. If it is 1, it is an indefinite response, i.e. can be either miss or hit.

There are several reasons for designing a segmented Bloom filter: 1) We only need the bit vector, whose size is smaller than the counter, to know the outcome of a query to the Bloom filter. Decoupling the bit vector enables faster and low power accesses to the Bloom Filter. So, the result of a query issued from the core can be obtained by just looking at the bit vector. 2) The update to the counters is not time critical with respect to the core. So, the segmented design allows the counter array to run at a much lower frequency than the bit vector. The vector part being smaller provides a fast access time, whereas the larger counter part runs at a lower frequency to save energy. The only additional overhead of this segmented design is the duplication of the hash function hardware. Using a single hash function in the Bloom filter also simplifies the implementation and duplication of the hash function. 3) The decoupled bit vector can sit between the L1 and L2 caches or can also be integrated into the core. For systems in which the L1 and L2 caches are inclusive, the integrated bit vector can also filter out the L1 instruction and data caches if an L2 cache miss is detected.

3.1 SoC Energy Management

This section explains how the segmented Bloom Filter detects L2 Cache misses, and saves the overall system energy without losing performance in an in-order processor. In an in-order processor with two cache levels, severe stalls may occur due to an L2 Cache miss. This is because after a data access misses the L2 cache, it accesses the DRAM memory, which may take more than 100 cycles, depending on the processor frequency before the data returns.

By detecting an L2 cache read miss early with a segmented Bloom filter, we can save static energy of the system by turning off all or part of the core and by putting the L1 and L2 caches into drowsy or low-power state-preserving mode until the data returns. The overhead incurred by this technique is turning on and

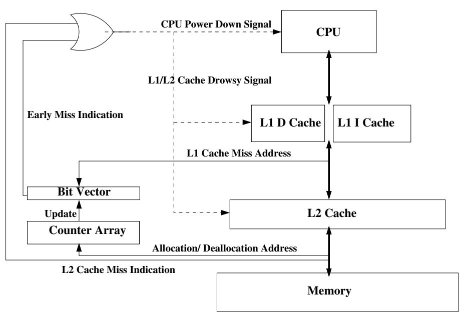


Fig. 3. SOC with caches not assumed to be inclusive and the bit vector below the L1 Cache

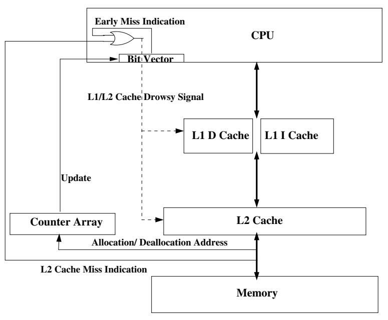


Fig. 4. SOC with inclusive caches and the bit-vector inside the CPU

turning off of the core and caches. This overhead is not much of a concern because the turn-off period overlaps with the memory access, which may take hundreds of cycles. Also, since it is known exactly when the data returns from memory, the turned-off units can be turned on in stages to save power. In addition to reducing static energy, dynamic energy of the system can also be reduced by preventing an L2 Cache access. Not only does this save the dynamic energy of the L2 but also reduces the bus energy consumption due to reduction in bus switching activity.

The segmented Bloom filter is shown in Figure 3 for a SOC in which the L1 and L2 caches which are not assumed to exhibit inclusive behaviour. In such a system, the bit vector is located just below the L1 caches. The CPU issues a cache address to the L1 data cache. On a miss, the bit vector snoops the address and signals in a cycle if the L2 cache does not have the cache line. On receiving the signal, the CPU is powered down and the L1 I and D and L2 caches can be put into the drowsy mode. The access to the L2 cache is also stopped.

Figure 4 shows a system where the L2 cache is inclusive with the L1 caches. Here, the bit vector is placed inside the core and can detect L2 cache misses before they are sent to the L1 caches. In a cache system using inclusion property, an L2 miss is also a miss in the L1 cache. Thus, a cache request address can be sent to directly to memory when a miss is detected by the bit vector inside the core.

For both systems, the bit vector may not be 100% consistent with the counter array as there is some delay occuring between the counter array and bit vector. This situation happens if incrementing the counter in the counter array is deferred till the time of a linefill. At that moment, the corresponding bit location in the bit vector might be 0. So, if the counter changes from 0 to 1, the counter array sends an update to the bit vector to set the bit location in the vector. Before this update reaches the bit vector, if the CPU accesses to the same bit location, then it reads 0 and assumes that this line is not in the cache and therefore forwards the request to memory. This drawback is eliminated if the counter is incremented at the time of the miss, rather than the linefill. By the time the actual linefill occurs, the bit vector will have been updated by the counter array. We see that segmenting the Bloom Filter allows the bit vector to be placed in a different physical location leading to more energy saving opportunities. This concept may be extended to cases where there are more than two levels of caches and the segmented bloom filter is used to filter out requests to the cache that is accessed just before DRAM memory. In such a case, though the counter array would be updated for the cache before memory, the bit vector may be kept at a place where it would be accessible with any of the previous cache levels, thus providing early miss indication.

4 Experimental Results

4.1 Experimental Framework and Benchmarks

We use SimpleScalar [3] to model the behavior of caches and segmented Bloom filter. The CPU is an in-order processor that stalls on a load operation, which is a typical behavior of many embedded processors. We compute the total energy consumption of the on-chip system including the CPU, caches and the Bloom filter. Our baseline model is the system with no Bloom filter. We use a total of eight applications, *bzip2, gcc, gzip, mcf, parser, vortex* and *vpr* from *SPECint2000*, *lame* MP3 player application from *MiBench* [4]. 2 billion instructions are simulated in the SPECint benchmarks while *lame* runs to completion. SPECInt benchmarks were chosen because they are known to stress the L2 cache. Only a few embedded applications such as *lame* could stress the L2 cache.

Table 1. Architectural assumptions

Drowsy-mode in/out time = 10 cycles
CPU turn-on/off time = 10 cycles
Shutdown Penalty = 20 cycles
Bit vector access time = 1 cycle
Memory access time = 100 cycles
CPU Energy = 2 x L1 Cache Energy
Cache Drowsy Energy = 1/6 x Cache Leakage Energy

Other pertinent architectural assumptions or fixed-parameters are listed in Table 1. The following assumptions are made to estimate the energy consumption of the baseline system (i.e. system without the Bloom filter) and a low-power system with the segmented Bloom filter. The time taken to put the caches in drowsy mode is 10 cycles, and it also takes another 10 cycles to put them into the normal mode. Similarly, the time taken to turn the CPU components off is also assumed to be 10 cycles. The total time for turning on and turning off, that is 20 cycles is called the *shutdown penalty.* The access time to the bit vector takes one cycle while the memory access time is 100 cycles. We also assume that the CPU energy consumption is twice the total L1 instruction and data cache energy consumption. This is a realistic assumption as embedded processors tend to have this trend as illustrated in [5]. The cache leakage energy in the drowsy mode is taken to be one sixth of the cache leakage energy as estimated by *Flautner et al* in [6].

We experiment two different cache architecture configurations as shown in Table 2. The first configuration has 2-way set-associative 8KB L1 instruction and data caches, 4-way 64KB L2 cache, a 8192-bit Bloom filter bit vector and a

Table 2. Architectural configurations

Description	**Configuration 1**	**Configuration 2**
L1 I and D cache	2-way 8KB	2-way 32KB
L2 cache	4-way 64 unified	4-way 256 unified
Bit vector size	8192 bits	32768 bits
Counter array size	8192 3-bit counters	32768 3-bit counters
L1 latency (cycles)	1	4
L2 latency (cycles)	10	30

Bloom filter counter array of 8192 entries with 3-bit[1] counter per entry. The line size is 32B for both L1 and L2 caches. The latencies of the L1 instruction and data caches and L2 cache are 1 and 10 cycles, respectively. This configuration represents low-end market such as industrial and automative applications in the embedded domain.

The second configuration includes 2-way set-associative 32KB L1 instruction and data caches, 4-way 256KB L2 cache, each has a 32B line size. The Bloom filter consists of a 32768-bit bit vector and a counter array of 32768 entries with 3-bit counter per entry.

The latencies of the L1 instruction and data caches and L2 cache are 4 and 30 cycles, respectively. This configuration represents the domain where slightly larger scale applications are targeted, e.g. consumer and wireless applications.

We have chosen the number of Bloom Filter entries to be around four times the number of cache lines. We experimented with different BF sizes and found this emperical ratio to provide best results. The area overhead for the Bloom Filters is about 6% of the L2 Cache area for both the configurations.

4.2 Energy Modeling

The L1 caches, L2 cache, bit vector and the counter array were designed using the *Artisan* 90nm SRAM library [7] in order to get an estimate on the dynamic and static energy consumption of the caches and the segmented Bloom filter. The Artisan SRAM generator is capable of generating synthesizable verilog code for SRAMs in 90nm technology. The generated datasheet gives an estimate of the read and write power of the generated SRAM. The datasheet also provides a standby current from which we can estimate the leakage power of the SRAM.

We have two system energy models. The first model is the baseline model in which the dynamic and static energy consumption of the CPU, L1 instruction and data caches and the L2 cache are calculated. The second system model is the low-power system model in which the dynamic and static energy consumption of the bit vector and counter array is also added to the rest of the system components. Table 3 shows the abbreviation of the variables used in the formulation to evaluate the system energy of the baseline and low-power system models.

Baseline System Energy Model.

$$\begin{aligned}
Cyc_{off} &= Num_{L2readmiss} * (Lat_{mem} - SP) \\
Cyc_{on} &= Cyc_{tot} - Cyc_{off} \\
E_{cpu}^{base} &= Cyc_{on} * CPU_{dyn} + Cyc_{off} * CPU_{leak} \\
E_{\$}^{base}(type) &= Num_{cacheaccess} * \$_{dyn} + Cyc_{on} * \$_{leak} + Cyc_{off} * \$_{dr} \\
E_{sys}^{base} &= E_{cpu}^{base} + E_{\$}^{base}(I) + E_{\$}^{base}(D) \\
&+ E_{\$}^{base}(L2) \qquad (1)
\end{aligned}$$

[1] Although the worst case number of bits required per counter is 12, we observe in our experiments that the value of each counter never exceeds 4. Thus we use 3 bits per counter to save energy and have a policy of disabling a particular counter if it saturates.

Table 3. Abbreviations and their descriptions

Abbreviation	**Description**
Cyc_{tot}	Total Number of Cycles
Cyc_{off}	Number of Idle Cycles
Cyc_{on}	Number of Active Cycles
$Num_{cacheaccess}$	Number of Cache Accesses
$Num_{L2readmiss}$	Number of L2 Read Misses
$Num_{L2access}$	Number of L2 Accesses without filtering
$Num_{L1access}$	Num of L1 Accesses
Num_{L2filt}	Number of Filtered L2 Misses
Lat_{mem}	Memory Latency
SP	Shutdown Penalty
Lat_{L2}	L2 latency
Lat_{vector}	Bit vector latency
CPU_{dyn}	CPU Dynamic Energy per Cycle
CPU_{leak}	CPU Leakage Energy per Cycle
$\$_{dyn}$	Cache Dynamic Energy per Cycle
$\$_{leak}$	Cache Leakage Energy per Cycle
$Cache_{dr}$	Cache Drowsy Energy per Cycle
BV_{dyn}	Bit Vector Dynamic Energy per Cycle
BV_{leak}	Bit Vector Leakage Energy per Cycle
BV_{dr}	Bit Vector Drowsy Energy per Cycle
$Counter_{dyn}$	Counter Array Dynamic Energy per Cycle
$Counter_{leak}$	Counter Array Leakage Energy per Cycle
$Counter_{dr}$	Counter Array Drowsy Energy per Cycle

Low-Power System Energy Model. We now estimate the energy consumption of the low-power system model having L1 and L2 caches which are assumed to exhibit inclusive behaviour with the segmented Bloom filter as follows:

$$\begin{aligned}
Cyc_{off} &= Num_{L2readmiss} * (Lat_{mem} - SP) + Num_{L2filt} * (Lat_{L2} - Lat_{vector}) \\
Cyc_{on} &= Cyc_{tot} - Cyc_{off} \\
E^{low}_{cpu} &= Cyc_{on} * CPU_{dyn} + Cyc_{off} * CPU_{leak} \\
E^{low}_{L2} &= (Num_{L2access} - Num_{L2filt}) * L2_{dyn} + Cyc_{on} * L2_{leak} + Cyc_{off} * L2_{dr} \\
E^{low}_{L1}(type) &= Num_{L1access} * L1_{dyn} + Cyc_{on} * L1_{leak} + Cyc_{off} * L1_{dr} \\
E^{low}_{vector} &= Num_{L2access} * BV_{dyn} + Cyc_{on} * BV_{leak} + Cyc_{off} * BV_{dr} \\
E^{low}_{counter} &= Num_{L2access} * Counter_{dyn} + Cyc_{on} * Counter_{leak} + Cyc_{off} * Counter_{dr} \\
E^{low}_{sys} &= E_{cpu} + E^{low}_{L1}(I) + E^{low}_{L1}(D) + E^{low}_{L2} + E^{low}_{vector} + E^{low}_{counter}
\end{aligned}$$

If the L1 and L2 caches are inclusive, then the energy consumption of the L1 cache is determined by the total number of L1 accesses less the number of filtered L2 misses. Also, the number of L2 accesses is replaced by the number of L1 accesses in the bit vector energy equation.

$$\begin{aligned}
E^{low}_{L1}(type) &= Num_{L1access} - Num_{L2filt} * L1_{dyn} + Cyc_{on} * L1_{leak} + Cyc_{off} * L1_{dr} \\
E^{low}_{vector} &= Num_{L1access} * BV_{dyn} + Cyc_{on} * BV_{leak} + Cyc_{off} * BV_{dr}
\end{aligned}$$

Finally, the percentage savings in the total system(Dynamic + Leakage) energy is defined by the following equation:

$$\% \text{ Savings} = \frac{E_{sys}^{base} - E_{sys}^{low}}{E_{sys}^{base}} \tag{2}$$

4.3 Cache and Bloom Filter Statistics

The cache miss rates for the L1 instruction and data caches and L2 cache and the miss filter rates of the Bloom filter for the two configurations are provided in Table 4 and Table 5. The miss filter rates in the last column of the tables are the percentage of the L2 misses that the Bloom filter can detect. For instance, 94% of the L2 misses can be detected by the Bloom filter in *gcc* for the first configuration. The remaining 6% of them cannot be detected, i.e. false hit rate. The average miss filter rates across all benchmarks are 86% and 88% for both configurations. These rates imply that a great majority of the L2 misses can be caught by the Bloom filter. An 88% filtering of L2 misses also implies that the Bloom Filter is able to reduce accesses to the L2 cache by more than 30%.

Table 4. Cache miss and miss filtering rates for configuration 1

Benchmark	**L1 I**	**L1 D**	**L2**	**Bloom Filter**
bzip2	4.82%	0.002%	45.55%	83.21%
gcc	10.52%	4.19%	48.56%	94%
gzip	5.66%	0.01%	45.99%	96.12%
mcf	26.21%	1.24%	58.07%	87.60%
parser	6.08%	0.68%	36.68%	82.76%
vortex	3.84%	13.24%	21.41%	84.49%
vpr	3.64%	2.14%	13.35%	81.47%
lame	2.76%	0.78%	27.61%	81%
MEAN	**7.84%**	**2.78%**	**37.15%**	**86.33%**

Table 5. Cache miss and miss filtering rates for configuration 2

Benchmark	**L1 I**	**L1 D**	**L2**	**Bloom Filter**
bzip2	3.54%	0.0002%	48.90%	88.36%
gcc	10.01%	1.55%	55.92%	99.07%
gzip	4.72%	0.001%	12.45%	95.25%
mcf	25.12%	0.0001%	63.74%	83.43%
parser	3.60%	0.05%	31.81%	86.38%
vortex	1.36%	4.84%	5.88%	77.96%
vpr	1.71%	0.21%	39.19%	83%
lame	0.99%	0.30%	12.36%	93.88%
MEAN	**6.38%**	**0.87%**	**33.78%**	**88.42%**

4.4 Energy Consumption Results

Table 6 shows the L2 dynamic energy savings for the two configurations with respect to the L2 cache in the baseline model. *gzip*, *parser*, *vortex* and *lame* suffer a drop in L2 dynamic energy savings in the second configuration because of improvements in L2 miss rates for using a much larger L2 cache. As the L2 miss rate improves, the number of misses of which the Bloom filter can take advantage to shutdown the CPU and caches diminishes. The L2 energy savings drop rates in *gzip*, *vortex* and *lame* are more dramatic because their miss filtering rates also drop in the second configuration except for *lame* where it actually improves. However, this increase in the miss filtering rate is not sufficient to boost the L2 energy savings for *lame*. The miss filtering rate for *parser* also improves in the second configuration. This explains why the drop in L2 energy savings in the second configuration for *parser* is not as significant as the others.

In summary, using the segmented Bloom filter provides an average of 33% and 30% savings in the L2 dynamic energy respectively for the two configurations.

Figure 5 plots the SoC static energy savings. The SoC static energy includes the leakage energy of the CPU, L1 and L2 caches in the baseline model, and

Table 6. L2 cache energy savings

Benchmark	**Configuration 1**	**Configuration 2**
bzip2	37.90%	43.21%
gcc	45.65%	55.40%
gzip	44.21%	11.85%
mcf	50.87%	53.18%
parser	30.35%	27.47%
vortex	18.09%	4.59%
vpr	10.88%	32.53%
lame	22.37%	11.60%
MEAN	**32.54%**	**29.98%**

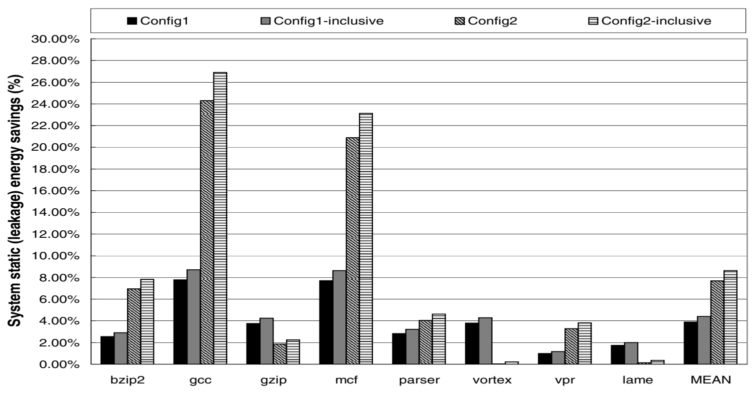
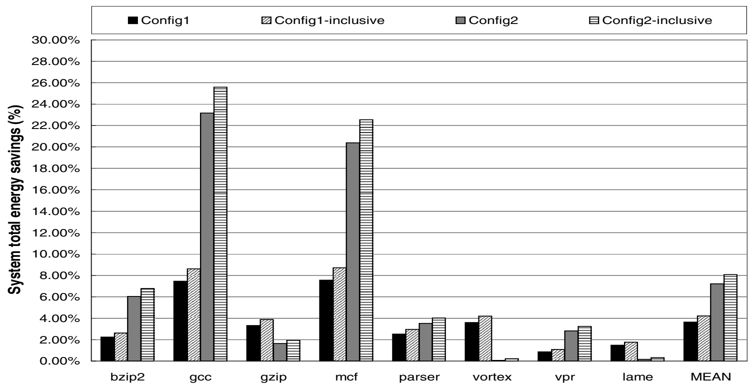


Fig. 5. Static SoC energy results



systemenergy.eps

Fig. 6. Total SoC energy results

the leakage energy of the bit vector and the counter array are accounted for the low-power SoC model. In addition to the two configurations, we also show the results of the inclusive versions for each configuration. In the inclusive version, the bit vector is embedded within the core and filters out the L1 instruction and data cache accesses as well.

The percentage increases in the system static energy savings are quite significant for *gcc* and *mcf* from a smaller configuration to larger one. In configuration 2, 24% and 21% of the static energy consumption can be saved by using the segmented Bloom filter for *gcc* and *mcf*, respectively. The percentage increase in *gcc* is higher than *mcf* because the L2 miss rate increases and the miss filtering rate improves in *gcc* . Similar to the L2 dynamic energy results, when switching from a smaller configuration to a larger one, *gzip*, *vortex* and *lame* benchmarks observe some percentage loss in the static energy savings due to lower L2 miss rates. However, the static energy savings of *parser* in configuration 2 is slightly higher than that of configuration 1 even though its L2 miss rate is lower. This is because the high miss filtering rate in configuration 2 is sufficient to boost the energy savings.

The inclusive versions for both configurations show slightly better savings than the cases where inclusion is not assumed, for all benchmarks because the inclusive configuration allows early turning off the system components, which reduces the system static energy consumption.

The average system static energy savings are 3.9%, 4.4%, 7.7% and 8.7% for configuration 1, its inclusive version, configuration 2 and its inclusive version, respectively.

Figure 6 plots the total SoC energy savings in percentage. The total SoC energy is defined as the total dynamic and static energy consumed by the CPU, L1 caches, L2 cache for the baseline model. This also includes the dynamic and static energy consumption of the bit vector and the counter array for the low-power SoC model. Here, we see a very similar trend to the system static energy savings graph above in terms of rise and falls in the system total energy savings when changing to a larger configuration from a smaller one.

Similar to the SoC static energy reduction, the inclusive versions for both configurations reduces the total energy more than the cases where inclusion property is not assumed, for all benchmarks because of reductions in the number of L1 cache accesses, which reduces the dynamic as well as the static energy consumption.

The average total SoC energy savings for the first configuration and its inclusive version are 3.6% and 4.2%, respectively. These rates go up to 7.2% and 8.1% for the second configuration and its inclusive version. The reason for the additional improvement is due to much higher the L2 latency in the second configuration. A large amount of static energy can be saved during the long-latency L2 accesses by turning off the CPU, caches and also the counter array. Since the bit vector access time is constant, the effective gain in the total energy with increasing L2 latencies (i.e. larger L2 caches) also rises.

5 Related Work

The initial purpose of Bloom Filters was to build memory efficient database applications. Bloom filters have found numerous applications in networking and database areas [8] [9] [10] [11] [12] [13]. Bloom filters are also used as microarchitectural blocks for tracking load/store addresses in load/store queues. For instance, *Akkary et al.* [14] uses one to detect the load-store conflicts in the store queue. *Sethumadhvan et al.* [15] improve the scalability for load store queues with a Bloom filter. More recently, *Roth* [16] uses a Bloom filter to reduce the number of load re-executions for load/store queue optimizations.

The earliest example of tracking cache misses with a counting Bloom filter is given by *Moshovos et al.* [17], which proposes a hardware structure called *Jetty* to filter out cache snoops in SMP systems. Each processing node has a *Jetty* that tracks its own L2 cache accesses, and snoop requests are first checked in the *Jetty* before searching the cache. This is reported to reduce snoop energy consumption in SMP systems. A *Jetty*-like filter is also used by *Peir et al.* [18] for detecting load misses early in the pipeline so as to initiate speculative execution. Similarly, *Mehta et al.* [19] also uses a *Jetty*-like filter to detect L2 misses early so that they can stall the instruction fetch to save processor energy. We, on the other hand, propose a decoupled Bloom filter structure where the small bit vector can potentially be kept within the processor core to perform system dynamic and static energy conservation of L1 and L2 caches and the core itself.

Memik et al. [20] proposes some early cache miss detection hardware techniques encapsulated as *Mostly No Machine(MNM)* to detect misses early in the multi-level caches below L1 (i.e. L2, L3 and etc). Their goal is to reduce dynamic cache energy and to improve the performance by bypassing the caches that will miss. The MNM is a multi-ported hardware structure that collects block replacement and allocation addresses from these caches and can be accessed after the L1 access or in parallel with it. In comparison to the MNM, the segmented

Bloom filter design allows the processor to access only the bit vector, which is smaller and much faster. Potentially, it can run at the processor frequency. Since the counter array is located at the L2 cache, it can run at the same clock frequency as the slower L2 cache. This is a more energy-efficient design than the MNM. Besides, the bit vector can also be located inside the processor so that the L1 instruction and data cache misses can also be filtered out in the case of an inclusion between the L1s and L2. This way, we can save L1 I and D cache dynamic energy by not accessing them at all, and static energy by putting them into a drowsy mode. The *MNM* did not discuss static energy consumption in the caches, CPU or filters.

6 Conclusion

This paper introduces a segmented counting Bloom filter to perform microarchitectural energy management in an embedded SoC environment and evaluates its energy saving capabilities. We have shown that the segmented Bloom filter technique can be an efficient microarchitectural mechnanism for reducing the total SoC energy consumption. A significant part of the total SoC energy including L2 dynamic cache energy, L1, L2 and CPU static static energy can be saved in a system where the cache hierarchy is not assumed to exhibit inclusive behaviour. However, the segmented design is shown to be particularly more energy-efficient if the cache hierarchy exhibits inclusive behaviour. This is because the segmented design provides the opportunity to make the bit vector accesible before the L1 Cache access and allows for detection of misses much earlier in the memory hierarchy. The segmented counting bloom filter has been shown to filter out more than 89% of L2 misses, causing a 30% reduction in accesses to the L2 Cache. This results in a saving of more than 33% of L2 Dynamic Energy. The results also demonstrated that the overall SoC energy can be reduced by up to 9% using the proposed segmented Bloom filter.

As future embedded applications demand more memory and shrinking feature sizes allow more transistors on a die, embedded processors would be inclined to have larger caches. Having these longer latency caches would provide more opportunities for the segmented design to facilitate microarchitectural energy management earlier in the memory hierarchy. Therefore cache miss detection in general and the segmented filter design presented in this paper would play a key role in energy management for future embedded processors.

References

1. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(4) (1970)
2. Fan, L., Cao, P., Almeida, J., Broder, A.: Summary cache: A scalable wide-area web cache sharing protocol. IEEE/ACM Transactions on Networking **8**(3) (2000) 281–293

3. Burger, D., Austin, T.M.: The simplescalar tool set, version 2.0. Technical Report 1342, Computer Science Department, University of Wisconsin-Madison and MicroComputer Research Labs, Intel Corporation (1997)
4. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: the IEEE 4th Annual Workshop on Workload Characterization, Austin, TX (2001)
5. Fan, D., Tang, Z., Huang, H., Gao, G.R.: An energy efficient tlb design methodology. In: Proceedings of the International Symposium on Low Power Electronics and Design. (2005)
6. Flautner, K., Kim, N.S., Martin, S., Blaauw, D., Mudge, T.: Drowsy caches: Simple techniques for reducing leakage power. In: Proceedings of the 29th Annual International Symposium on Computer Architecture. (2002)
7. Artisan: Sram libraries. `http://www.artisan.com` (2005)
8. Border, A., Mitzenmacher, M.: Network application of bloom filters: A Survey. In: 40th Annual Allerton Conference on Communication, Control, and Computing. (2002)
9. Rhea, S., Kubiatowicz, J.: Probabilistic location and routing. In: IEEE INFOCOM'02. (2002)
10. Dharmapurikar, S., Krishnamurthy, P., Sproull, T., Lockwood, J.: Deep packet inspection using parallel bloom filters. In: IEEE Hot Interconnects 12. (2003)
11. Kumar, A., Xu, J., Wang, J., Spatschek, O., Li, L.: Space-code bloom filter for efficient per-flow traffic measurement. In: Proc. IEEE INFOCOM. (2004)
12. Chang, F., Feng, W., Li, K.: Approximate caches for packet classification. In: Proc. IEEE INFOCOM. (2004)
13. Cohen, S., Matias, Y.: Spectral bloom filters. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. (2003)
14. Akkary, H., Rajwar, R., Srinivasan, S.T.: Checkpoint processing and recovery: Towards scalable large instruction window processors. In: Proceedings of the 36th International Symposium for Microarchitecture. (2003)
15. Sethumadhavan, S., Desikan, R., Burger, D., Moore, C.R., Keckler, S.W.: Scalable hardware memory disambiguation for high ilp processors. In: Proceedings of the 36th International Symposium for Microarchitecture. (2003)
16. Roth, A.: Store vulnerability window (svw): Re-execution filtering for enhanced load optimization. In: Proceedings of the 32th International Symposium on Computer Architecture (ISCA-05). (2005)
17. Moshovos, A., Memik, G., Falsafi, B., Choudhary, A.: Jetty: Snoop filtering for reduced power in smp servers. In: Proceedings of International Symposium on High Performance Computer Architecture (HPCA-7). (2001)
18. Peir, J.K., Lai, S.C., Lu, S.L., Stark, J., Lai, K.: Bloom filtering cache misses for accurate data speculation and prefetching. In: Proceedings of the 16th International Conference of Supercomputing. (2002) 189–198
19. Mehta, N., Singer, B., Bahar, R.I., Leuchtenburg, M., Weiss, R.: Fetch halting on critical load misses. In: Proceedings of the The 22nd International Conference on Computer Design. (2004)
20. Memik, G., Reinman, G., Mangione-Smith, W.H.: Just say no: Benefits of early cache miss determination. In: Proceedings of the Ninth International Symposium on High Performance Computer Architecture. (2003)