An Energy-Efficient Adaptive Hybrid Cache

Jason Cong, Karthik Gururaj, Hui Huang, Chunyue Liu, Glenn Reinman, Yi Zou
Computer Science Department, University of California, Los Angeles
Los Angeles, CA 90095, USA
{cong, karthikg, huihuang, liucy, reinman, zouyi}@cs.ucla.edu

***Abstract*—By reconfiguring part of the cache as software-managed scratchpad memory (SPM), hybrid caches manage to handle both unknown and predictable memory access patterns. However, existing hybrid caches provide a flexible partitioning of cache and SPM without considering adaptation to the run-time cache behavior. Previous cache set balancing techniques are either energy-inefficient or require serial tag and data array access. In this paper an adaptive hybrid cache is proposed to dynamically remap SPM blocks from high-demand cache sets to low-demand cache sets. This achieves 19%, 25%, 18% and 18% energy-runtime-production reductions over four previous representative techniques on a wide range of benchmarks.**

Keywords—Energy Reduction; Hybrid Cache; Scratchpad Memory

I. Introduction

Caches are widely used in modern processors to effectively hide the data access latency, since the memory reference patterns in most applications have good spatial/temporal locality. For applications with predictable data access patterns, it is possible to let the software directly manage the on-chip storage. This alternative is called *scratchpad memory* (SPM). Because the SPM does not need to perform tag comparisons and drive associative ways, it is much more energy-efficient than caches [1]. Embedded architectures use SPM in conjunction with caches to reduce power consumption.

However, certain applications may prefer SPM (e.g., with predictable array access patterns) while other applications may prefer cache (e.g., with dynamic and random accesses). Even for applications that prefer SPM, the SPM size required by different applications may vary [1]. Under these circumstances, designing the cache and SPM separately at the physical level with a fixed size for each of them is likely to be suboptimal for particular applications.

As a result, reconfigurable caches have been proposed to provide good support for flexibly sizing the cache and SPM based on application requirements in a *hybrid cache* design. Column caching [2] and FlexCache [3] expose part of the cache as software-controlled memory. The reconfigurable cache [4] and virtual local store [5] enable the cache to be dynamically partitioned at a granularity from cache ways to cache blocks. Besides the reconfigurable caches, an adjustable-granularity cache-locking function—available on multiple embedded architectures such as Freescale e300 [6]—can also be utilized to achieve flexible partitioning of cache and SPM. Way stealing [7] uses special cache preload and locking instructions to provide local memory for instruction set extensions.

However, all of the above hybrid cache designs partition the cache and SPM without adaptation to the run time cache behavior; i.e., when allocating cache blocks into SPM, they will select blocks from cache sets uniformly. Since cache sets are not uniformly utilized [8], this uniform mapping of SPM blocks onto cache blocks may create hot cache sets at run time, which will increase the conflict miss rate and degrade the performance. Figure 1 shows the cache set utilization stats for a hybrid cache design (system configuration is shown in Section IV.A). Each column represents a set in the cache, and each row represents 1 million cycles of time. A darker point means a hotter cache set. As can be seen, the cache set utilization varies for different cache sets and different times. It becomes more serious for low-power processors with low cache associativity due to a tight power budget.

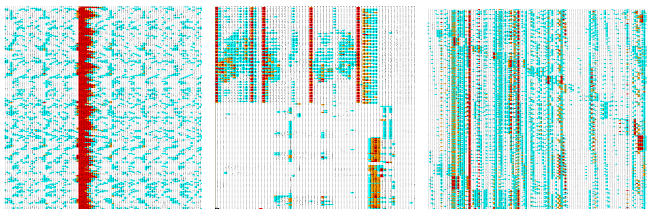

(a) *astar* (SPEC) (b) *jpeg* (MiBench) (c) *h264ref*(SPEC)

Figure 1. Non-uniformed cache sets utilization in hybrid cache.

Balancing cache set utilization has been intensively investigated. Most of these techniques, such as V-way cache [8], indirect index cache [9] and set balancing cache [10] require that the cache tag array and data array be serially accessed. However, since SPM is designed for fast local access, the hybrid cache is typically at the primary (L1) cache level, which generally requires a parallel access to tag/data array. Therefore, these pseudo-associative cache techniques are not suitable for hybrid caches. There are also techniques that do not require serial tag/data array access. Victim cache [11] uses a small fully associative cache to store the victim cache blocks to mitigate the conflict-miss. It increases the per-cache-access energy since victim cache is searched in parallel with the regular cache. Serializing the victim cache access can save energy but incurs additional cycles when hitting in the victim cache. Balanced cache [12] uses a content-addressable memory (CAM) inside the cache decoder and increases the decoder length to associate cache sets. Although the CAM access latency fits in the decoder slack and introduces only 10% per-cache-access energy overhead at 0.18um technology, as it comes to a nano-scale technology such as 32nm, the CAM access latency exceeds the decoder slack by 20%~40% and incurs a large per-cache-access energy overhead [13].

Therefore, it is important to find an energy-efficient approach to addressing the hot cache set problem in hybrid caches without requiring serial tag/data array access. Fortunately, the nature of hybrid cache provides another possibility for balancing the cache set utilization. Instead of pseudo-associating the cache sets (as done in the previous approaches) and maintaining a fixed SPM mapping, we can dynamically remap SPM blocks from high-demand cache sets to low-demand cache sets. Intuitively, it is similar to the previous cache-energy reducing techniques which dynamically activate and deactivate cache lines based on the cache set utilization [14]. However, switching on/off cache lines in one cache set will not influence the other cache sets, but migrating SPM blocks from a high-demand set to a low-demand set will increase the pressure of the destination cache set. Therefore, directly applying the previous approaches may result in a situation where several cache sets just keep passing SPM blocks among themselves repeatedly (referred to as *circular bouncing effect*).

Another challenge caused by dynamically remapping SPM blocks is to that of quickly locating the SPM block in the cache, since the SPM block locations may change. Obviously, using software to manage the remapping can be costly due to inefficiency and the impact on code portability. Therefore, hardware support is desired so that software can focus on the use of a logically continuous SPM.

This research is partially supported by the NSF Expeditions in Computing Award CCF-0926127, NSF grant CCF-0903541, and GSRC.

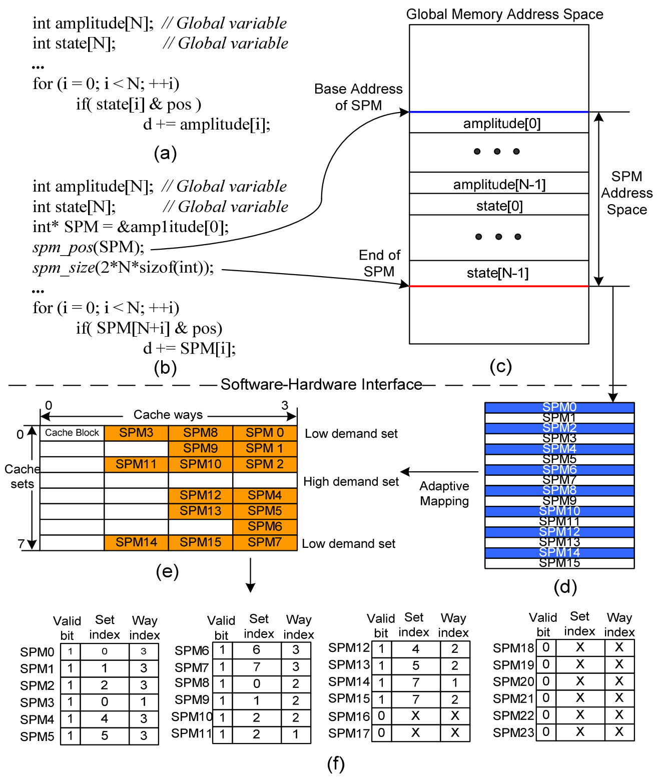
To the best of our knowledge, this is the first work that considers run-time adaptation in hybrid cache designs. The main contributions of the proposed *adaptive hybrid cache* (AH-Cache) are as follows:

- The look-up operation of the SPM location is hidden in the execution (EX) pipeline of the processor, and a clean software interface is provided as a non-adaptive hybrid cache.
- A victim tag buffer is used to assess the cache set utilization by sharing the tag array, resulting in no storage overhead.
- An adaptive mapping scheme is proposed for fast adaptation to the cache behavior without the circular bouncing effect using a floating-block-holders queue.

The remainder of this paper is organized as follows: Section II describes the software interface of AH-Cache. The AH-Cache architecture design and overhead is detailed in Section III. Section IV presents experimental results, and Section V concludes the paper.

II. Software Interface

First we will briefly talk about the software interface of AH-Cache, where we want to emphasize that the software only needs to be aware of a logically continuous SPM, but does not care where the SPM blocks are physically mapped. By providing such a clean software interface, 1) all of the previous compilation techniques that target SPM utilization optimization, such as dynamic data placement [1], stack and heap support in SPM [15], etc., can be directly used on AH-Cache since the compiler only views a logically continuous SPM; 2) in a multi-threaded architecture, the previous context switching schemes for SPM (e.g., [16]) can be directly used on AH-Cache, since the operating system only views a logically continuous SPM.



```
int amplitude[N];  // Global variable
int state[N];      // Global variable
...
for (i = 0; i < N; ++i)
     if( state[i] & pos )
          d += amplitude[i];
```

(a)

```
int amplitude[N];  // Global variable
int state[N];      // Global variable
int* SPM = &amplitude[0];
spm_pos(SPM);
spm_size(2*N*sizof(int));
...
for (i = 0; i < N; ++i)
     if( SPM[N+i] & pos)
          d += SPM[i];
```

(b)

	Valid bit	Set index	Way index
SPM0	1	0	3
SPM1	1	1	3
SPM2	1	2	3
SPM3	1	0	1
SPM4	1	4	3
SPM5	1	5	3
SPM6	1	6	3
SPM7	1	7	3
SPM8	1	0	2
SPM9	1	1	2
SPM10	1	2	2
SPM11	1	2	1
SPM12	1	4	2
SPM13	1	5	2
SPM14	1	7	1
SPM15	1	7	2
SPM16	0	X	X
SPM17	0	X	X
SPM18	0	X	X
SPM19	0	X	X
SPM20	0	X	X
SPM21	0	X	X
SPM22	0	X	X
SPM23	0	X	X

(f)

Figure 2. (a) Original code. (b) Transformed code for AH-Cache. (c) Memory space view of SPM in AH-Cache. (d) SPM blocks. (e) SPM mapping in AH-Cache. (f) SPM mapping look-up table (SMLT).

A simple example is shown in Figure 2. To manage the SPM in AH-Cache, the software is provided with two system APIs to specify the SPM base address and size. As shown in Figure 2(b), *spm_pos* sets the *SPM base address* register as the address of the first element of array *amplitude*, and *spm_size* sets the *SPM size* register as the size of the array *amplitude* and *state*. Note that these system APIs do not impact the ISA since they use regular instructions for register value assignment. The base address and size of the SPM can be set multiple times across the software. If the software sets the SPM size larger than the maximum SPM size (discussed in Section III.A), it can still run on AH-Cache, but AH-Cache will only provide its maximum SPM size. The SPM references beyond this size are treated as regular memory references and are supported by the cache. This scheme allows portability of the software on different AH-Cache sizes. We have developed a compilation pass [17] inside the LLVM [18] compilation infrastructure to automatically transform and optimize original application code for better SPM utilization on AH-Cache.

III. AH-Cache Architecture

A. *SPM Mapping Look-Up*

As shown in Figure 2(d)(e), the partition between cache and SPM in AH-Cache is at a cache-block-wise granularity. If the requested SPM size is not a multiple of a cache block, it will be increased to the next block-sized multiple. The mapping information of SPM blocks onto the cache blocks is stored into an *SPM mapping look-up table* (SMLT). The number of entries in SMLT is the maximum number of cache blocks that can be configured as SPM. Since AH-Cache must hold at least one cache block for each cache set to maintain the cache functionality, the maximum SPM size on a M-way N-set set-associative cache is $(M\text{-}1)*N$ blocks. In each SMLT entry, there are 1) a valid bit indicating whether this SPM block falls into the real SPM space, since the requested SPM size may be smaller than the maximum SPM size; and 2) a set index and a way index which locate the cache block upon which the SPM block is mapped.

In a most recent non-adaptive hybrid cache design [5], the high-order bits of the virtual address of a memory reference are checked in the early pipeline (after the ALU computes the virtual address) to determine whether it is targeting the SPM or regular cache. The checking is done by comparing these high-order bits with the SPM base address. This enables fast checking, but requires that the SPM base address be aligned with all of its low-order bits as 0.

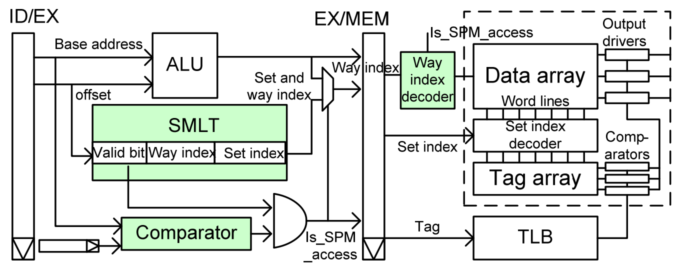


Figure 3. SPM mapping look-up and access in AH-Cache.

AH-Cache needs an additional step to use the low-order bits of the virtual address to look-up the SMLT. This further increases the pipeline critical path. To solve this problem, inspired by the zero-cycle load idea [19], we perform the address checking and SMLT look-up in parallel with the virtual address calculation of the memory operation in a pipelined architecture, as shown in Figure 3. Assuming a base + displacement address calculation mode, memory reference instructions will compute their virtual addresses from a base address and an offset address; these are obtained either from the register file or the immediate value of the instruction in the Instruction Decode (ID) stage. In the Execution (EX) stage, the ALU calculates the virtual address from these values. Simultaneously, the base address is compared to the SPM base address, and the offset is sent to the SMLT to obtain the mapping information (here the cache-set-index part of the offset bits will be used to index SMLT). When the output of the comparator is true and the valid information of the indexed SMLT entry is true, this memory reference is considered an SPM access instruction. In this way, both the address-checking and SMLT look-up are done in the EX stage, and SPM access time in AH-Cache will not be increased.

This architecture imposes a constraint on the compiler. The compiler should generate the memory reference instructions to SPM in such a way that the base address of this instruction must be the SPM base address and the offset must be the offset related to the SPM base address. This constraint does not impact the optimization ability of the compiler since this transformation can be performed in the last stage of the code optimizations. However, extra care needs to be taken when a pointer of some element of the SPM is passed as a parameter to a function, and all memory references inside the callee function are based on the input pointer parameter. The compiler should first divide the callee's input pointer parameter into two parts, a base pointer *base* and an *offset* of the original input pointer to *base*. Then, inside the callee, all memory references related to the original input pointer are generated with *base* as the new base pointer. For the caller, the SPM base address is passed to the callee's *base*, and the offset of original input pointer to SPM base address to the callee's *offset*.

One concern is whether the virtual address calculation at the ALU can hide the look-up time of the SMLT (obviously the comparator is not in the critical path since it is much simpler than the ALU). For an M-way N-set set-associative cache, the size of SMLT is $(M\text{-}1)*N$ entries with each entry containing $(1+\log M+\log N)$ bits. TABLE I shows the access latencies for various L1 cache configurations using Cacti [20] at 32nm technology (cache block size is 64B). As shown in the table, all the SMLT accesses can be finished in 0.2 ns which fits into the cycle time of a 4GHz core. Given the fact that previous non-adaptive cache [5] could add a comparator after the ALU, the small delay of the 1-level MUX added after the ALU will be much smaller in timing. It should be noted that the way index encoder is also used in the non-adaptive hybrid cache [5] to avoid TLB look-ups and tag comparisons at SPM accesses. It is not an overhead of AH-Cache.

TABLE I. SMLT LATENCY OF VARIOUS CACHE CONFIGURATIONS

Cache Size	**8KB**		**16 KB**		**32 KB**		**64 KB**	
Cache associativity	2	4	2	4	2	4	2	4
SMLT entries	64	96	128	192	256	384	512	768
SMLT width (bits)	8	8	9	9	10	10	11	11
Access latency(ns)	0.14	0.15	0.16	0.17	0.17	0.18	0.18	0.19

B. *Cache Set Demand Assessment*

As in [8], we refer to cache sets that highly utilize most or all cache blocks as *high-demand sets*, and cache sets that underutilize their available blocks as *low-demand sets*. We want the low-demand sets to accommodate proportionally more SPM blocks than the high-demand sets, as shown in Figure 2(e). Miss rate can not be used to recognize a high-demand cache set, since for streaming applications with little locality or applications hopelessly thrashing the cache, even if the miss rate is high, there is little benefit in increasing the cache blocks. Therefore, we use a *victim tag buffer* (VTB) to capture the demand of each set; this is similar to the miss tag introduced in [14], but with no memory overhead (as explained below).

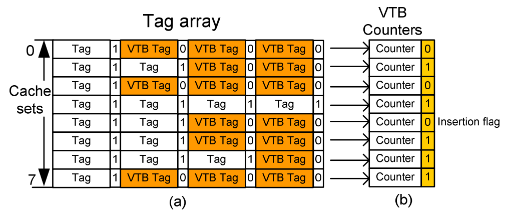


Figure 4. (a) A VTB in the tag array. (b) VTB counters and insertion flags.

Logically, the VTB consists of the same number of sets as the tag array and one less way in each set (since at least one cache block in each set is retained). When a cache block is configured as an SPM block, its tag is disabled, while its corresponding VTB tag is enabled. Once it is recovered from an SPM block and becomes a regular cache block, its tag is enabled, and its corresponding VTB tag is disabled. This way, we can naturally combine the original tag array and the VTB. For each tag entry in the original tag array, one bit is added to indicate whether this tag is a regular tag or a victim tag. Figure 4(a) shows the VTB inside the tag array for the mapping in Figure 2(e).

When a replacement happens in the cache part of AH-Cache, the tag of the victim block is written into the corresponding set of the VTB with pseudo LRU policy. There is a VTB counter for each set (not for each cache block, as shown in Figure 4(b)). The VTB is only accessed at a miss in the cache part of AH-Cache. If there is a hit in VTB, the set's VTB counter will be increased by 1, since this situation indicates that if this block had been enabled in the cache part of AH-Cache, it would have been a hit. Cache misses due to streaming or thrashing will not lead to a VTB hit, as there is no reuse.

C. *Adaptive Mapping*

If the application only requires P SPM blocks while AH-Cache can provide Q SPM blocks at most, then there will be S=Q-P cache blocks (referred to as *floating blocks*) used to adaptively satisfy the high-demand cache sets. When we say that cache set A gets a floating block from cache set B, it means that A sends one of its SPM blocks to B and enables the vacant cache way as a regular cache block, while B needs to evict one of its cache blocks to accommodate the SPM block from A. We can not simply make a cache set with a high VTB counter get a floating block from a cache set with a low VTB counter, since a low VTB counter only means that this cache set does not need more floating blocks; it does not mean that it can afford to lose one. Therefore, it is possible that several cache sets just keep passing SPM blocks among themselves repeatedly.

To solve this problem, we propose a mapping scheme based on a *floating block holder* (FBH) queue. The queue records the cache sets which are currently holding the floating blocks. Each queue node consists of the index of a floating block holder set and a *re-insertion bit*. A re-insertion bit indicates whether this cache set is re-inserted to the queue in the current adaptation interval (a fixed number of cycles). Each cache set holds a 1-bit *insertion flag* to indicate whether it has been inserted in the queue in the current interval, as shown in Figure 4(b). At the beginning of each adaptation interval, all the re-insertion bits in the queue and the insertion flags in the cache sets are reset to 0.

When a cache set A's VTB counter achieves a threshold T, the FBH queue will be searched, starting from the head, until a node with a re-insertion bit of 0 is found. Assume the set index in this node is B. Then set B will accommodate one SPM block from set A. This node is removed from the queue and a new node with set index A is inserted to the tail of the queue with its re-insertion bit as the current insertion flag of set A. Then set A's insertion flag is updated to 1. With the re-insertion bit as 1, a high-demand set will not give up its floating blocks once it is re-inserted to the queue in the current interval. Once all the re-insertion bits in the queue are 1, the remapping of this interval is stopped. This will effectively remove the potential circular bouncing effect. With a small number of SPM block migrations, the proposed mapping scheme can form an SPM mapping which adapts to the cache set demands in the current interval, as shown in TABLE IV. The threshold T determines the size of the VTB counter. The selection of T and the interval length I should be co-considered to make a lazy or aggressive adaptation. In this work we set T to 16 and I to 1 million cycles. Then the length of the VTB counter for each cache set is 4 bits.

Since a node removal is always accompanied with a node insertion, the FBH queue can be simply implemented with an SRAM controlled by a pointer. As shown in Figure 5(a), the number of active entries of this SRAM equals the number of floating blocks, and the total number of entries equals the maximum number of SPM blocks. When a VTB counter reaches the threshold and requests an SPM migration, the pointer will move from its current place until it finds one entry with a re-insertion bit of 0. Then the new node will overwrite this entry, and the pointer moves to the next entry. It turns back to the head of the SRAM when it reaches the end of the active region.

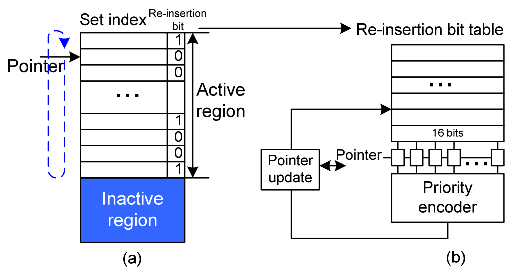


Figure 5. (a) FBH queue. (b) Parallel FBH search using RIBT.

Serially searching the FBH queue may incur a worst-case delay of *S* cycles, where *S* is the maximum SPM size. To reduce the searching time, a parallel search scheme is developed as follows. We store the re-insertion bits in another SRAM called *re-insertion bit table* (RIBT), as shown in Figure 5(b). Each RIBT entry contains 16 re-insertion bits. Then an *S*-entry FBH queue will have an *S*/16-entry and 16-bit wide RIBT. Every 16 re-insertion bits can be searched in parallel using a priority encoder which outputs the index of the first zero-bit of its input vector. Then the longest search time is decreased to *S*/16 cycles.

The FBH queue is searched when a cache set has a miss and its VTB counter achieves the threshold for requesting an SPM migration; thus, if the search can be finished before the missed cache block is fetched from the L2 cache, it will not affect the time of acquiring new data from the L2 cache. In our evaluation architecture, the L2 cache access latency is 20 cycles, while the maximum FBH queue length is 256 and the worst-case search time is 16 cycles, which is smaller than the L2 cache access latency. The FBH search latency can be further reduced by increasing the number of parallel searched re-insertion bits, at a cost of increasing the width of the priority encoder. We use the priority encoder designed in [21]. According to the Synopsys Design Compiler, the searching logic circuit has around 500 gates, which is less than 1% of the cache design.

D. *Storage and Energy Overhead*

To quantify the storage overhead of AH-Cache, we use a 16KB 2-way set-associative, 128 sets, 64B data block size, 4B tag entry size (including the tag, coherence state bits, dirty bits etc.) hybrid cache as an example. It can provide at most 128 64B SPM blocks. Then the SMLT contains 128 9-bit entries (1 valid bit + 6-bit set index + 2-bit way index). The VTB physically shares the tag array, thus it only incurs one additional bit for each tag entry. Each cache set also has one additional insertion flag and 4-bit VTB counter. The FBH queue contains 128 7-bit entries. The RIBT contains 8 16-bit entries. The migration buffer contains one 64B cache block. Therefore, the total storage overhead introduced by AH-Cache is around 0.4KB, which is 3% of the baseline hybrid cache size.

For the energy overhead, AH-Cache needs to access the SMLT at each cache access, access the VTB at each cache miss, and trigger the adaptive mapping unit (including the FBH queue, RIBT, and the 16-bit priority encoder) each time that a VTB counter achieves the threshold. According to Cacti [20], at 32nm technology, the access energy to the SMLT is up to 0.8pJ. The access to the VTB, which physically shares the tag array, adds an additional 0.8pJ for a 16KB cache and 0.9pJ for a 32KB cache per cache miss. The worst-case energy for the adaptive mapping unit when all the RIBT is searched is 2pJ; this is obtained from the Synopsys Design Compiler and Cacti [20]. When an SPM block migration happens, the block will be first read out from the cache and written into the migration buffer, and then read out from the migration buffer and written back into the cache in the next cycle; thus the energy overhead is 66pJ for a 16KB cache and 75pJ for a 32KB cache. As can be seen in TABLE IV, there are only 4.4 SPM migrations for average every 1 million cycles. Therefore, the total per-cache-access energy overhead of AH-Cache is only less than 6% of a non-adaptive hybrid cache with a per-cache-access energy of 16.6pJ for 16KB and 18.9pJ for 32KB. But it can save energy by reducing the low-level (L2) cache energy (reducing the miss rate) and leakage energy (reducing run time) as shown in the next section.

IV. Experiment Results

A. *Evaluation Methodology*

To cover a diverse set of applications, our benchmarks are chosen from multiple benchmark suites. We select the benchmarks which have intensive memory accesses that SPM can help to improve; i.e., we choose the benchmarks which can benefit from a hybrid cache design (since our goal is to improve the hybrid cache designs). These benchmarks include: five benchmarks from the MiBench benchmark suite [24]: *jpeg*, *gsm*, *dijkstra*, *patricia* and *susan*; five memory reference intensive applications from the SPEC2006 benchmark suite [25]: *h264ref*, *hmmer*, *astar*, *soplex* and *gobmk*; and also four medical imaging benchmarks [26]: 1) *biHarmonic* performs 2D image registration with bi-harmonic regularization term, 2) *mutualInfo* computes the mutual information of two 2D images, 3) *ricianDenoise* performs iterative local denoising based on the rician noise model, 4) *regionGrowing* evaluates whether a region is part of an object in image segmentation. The number of memory references of these benchmarks is shown in TABLE II in order to indicate their scale.

TABLE II. #Memory references of the evaluated benchmarks

jpeg	***gsm***	***susan***	***hmmer***	***soplex***	***h264ref***	***dijkstra***	***patricia***
19.8M	65.1M	76.1M	75.7M	22.1M	196.6M	47.7M	16.8M
astar	***gobmk***	***biHarmonic***	***mutualInfo***	***ricianDenoise***	***regionGrowing***		
93M	256.7M	48.6M	98.2M	7.9M	128.3M		

To demonstrate the advantage of the adaptation in AH-Cache, we implemented the following designs for comparison.

Non-adaptive hybrid cache (***N***): This is the baseline design which uses a 2-way set-associative hybrid cache as the L1 data cache. The SPM mapping onto cache blocks is fixed. We evaluated two cache sizes—16KB and 32KB, which are typical L1 data cache sizes in low-power processors. According to Cacti [20], the energy per access is 16.6pJ for 16KB and 18.9pJ for 32KB at 32nm technology.

Non-adaptive hybrid cache + balanced cache (***B***): This design enhances the baseline by using the balanced cache (B-Cache) [12]. It uses CAM and increases decoder length to increase the cache associativity. Due to the high energy overhead of CAM (90% more per-cache-access energy when BAS (B-Cache associativity) =8), to achieve a good performance and energy trade-off, we use BAS=4 and MF (mapping address mapping factor) =8 (1/8 of the memory address has a mapping to cache sets), which incurs additional per-cache-access energy of 6.4pJ for the 16KB cache and 7.8pJ for the 32KB cache. The energy data are obtained from [13] at 32nm technology, which extracts the technology parameters from Cacti [20]. It should be noted that the CAM access latency exceeds the original decoder slack, but we optimistically assume it does not increase the cache critical path.

Non-adaptive hybrid cache + victim cache (***Vp, Vs***): The design *Vp* enhances the baseline design by using a parallel accessed fully associative victim cache [11]. We use a 4-entry victim cache for the 16KB cache and an 8-entry victim cache for the 32KB cache; these have a per-cache-access energy overhead of 8.9pJ and 16.3pJ, respectively. Experiment results show that further increasing the victim cache size only marginally improves performance while using much more energy. We also implement a serially accessed victim cache *Vs*, where the victim cache is only accessed at a L1 cache miss to increase the energy efficiency, but it incurs additional cycles when blocks are in the victim cache. An additional pipeline is needed inside the hybrid cache to control the serial victim cache access.

Phase-reconfigurable hybrid cache (***R***): This design modifies the idea in [14] and applies it to the hybrid cache by reconfiguring the

SPM mapping at each fixed interval based on the VTB counter stats. At the reconfiguration time, cache sets with a VTB counter higher than a high-threshold can migrate their SPM blocks to cache sets with a VTB counter lower than a low-threshold. The length of the interval and the two thresholds are tuned to achieve the best performance. The architecture of this design is almost the same as our proposed design (hides SMLT access in EX pipeline and shares VTB in tag array), but without the adaptive mapping unit. It is used to evaluate the effectiveness of our adaptive mapping scheme.

Adaptive hybrid cache (***AH***): This is our proposed design. The energy overhead is discussed in Section III.D. The VTB counter threshold is 16 and adaptation interval length is 1 million cycles.

Static optimized hybrid cache (***S***): This design uses the offline analysis of the cache set demand stats to optimize the remapping at each interval. This design point is impractical, but it serves as a reference point to check the optimality of the AH-Cache.

Since all of the above designs can provide a clean software interface, from the software point-of-view they are the same. Thus the SPM configurations and utilizations for all the designs are the same. The benchmark binaries are generated by our compiler [17] to get the optimal SPM configurations. To accurately capture the system performance, we leverage the full system simulator SIMICS [22] and the GEMS toolset [23] as the timing model of the memory subsystem. All of the above designs are implemented in GEMS. The system configurations of SIMICS/GEMS are shown in TABLE III.

TABLE III. SIMICS/GEMS SIMULATOR CONFIGURATION

Core	Sun UltraSPARC-III Cu processor core
L1 Instruction Cache	16KB/32KB, 2-way set-associative, 64-byte block, 2-cycle access latency, pseudo-LRU
L1 Data Cache	16KB/32KB, 2-way set-associative, 64-byte block, 2-cycle access latency, pseudo-LRU
L2 Cache	512KB, 8-way set-associative, 64-byte block, 20-cycle access latency, pseudo-LRU
Main Memory	4GB, 320-cycle access latency

B. Performance Comparisons

Figure 6 shows the comparison results of misses for the L1 data cache (hybrid cache). The results are normalized to the baseline (design *N*). By functionally increasing the cache associativity with increased decoder length, design *B* reduces the cache misses by 44%. By accommodating victim cache blocks, designs *Vp* and *Vs* reduce the cache misses by 42%. By reconfiguring the SPM mapping at each interval, design *R* reduces cache miss by 34%. AH-Cache reduces the cache miss by 52% compared to baseline, and outperforms designs *B*, *Vp, Vs* and *R* by 19%, 22%, 22% and 33%, respectively.

The reason that AH-Cache outperforms design *B* is that the B-cache associates cache sets in a uniform way without considering the cache set demands; thus it is possible that the associated cache sets are all high-demand cache sets. Victim cache performance is constrained by its size (and additional victim cache access cycles for *Vs*). It can achieve larger miss rate reductions with a much higher energy overhead. Note that *Vp* and *Vs* perform very well for *ricianDenoise* which has only a few extremely high-demand cache sets. The fact that AH-Cache outperforms design *R* indicates that simply applying the previous phase-based reconfiguration approach to the hybrid cache can be affected by the circular bouncing effect. It can be seen that AH-Cache almost catches the optimality of design *S* in most cases (~1% difference), and even outperforms it at benchmarks *h264ref* and *susan* since design *S* is based on interval-level analysis, and it can not manage to adapt the dynamic variations inside an interval. This shows the positive effect of the run-time optimization of AH-Cache.

Figure 7 shows the performance comparison results in terms of run-time (cycles), which are normalized to the baseline (design *N*). The results of AH-Cache and design *R* include the remapping penalty (the core to L1 cache queue is suspended for two cycles for each SPM block migration). As shown in TABLE IV, the average number of SPM block migrations of AH-Cache at each interval is 4.4, which results in a run-time overhead of less than 0.1%. Some applications, such as *susan* and *gsm* which have dramatic cache misses reduction, do not see a corresponding run-time reduction because most of the memory references access SPM. However, the AH-Cache still reduces the run-time by 18% compared to baseline, and outperforms designs *B*, *Vp, Vs* and *R* by 3%, 4%, 8% and 12%, respectively.

TABLE IV. AVERAGE #SPM BLOCK MIGRATIONS IN EACH 1 MILLION CYCLE INTERVAL (UPPER: 16KB, LOWER: 32KB)

jpeg	***gsm***	***susan***	***hmmer***	***soplex***	***h264ref***	***dijkstra***	***patricia***
5.68	0.04	1.14	8.66	15.9	20.2	6.39	0.79
0.28	0.01	1.20	0.45	5.26	4.26	0.62	0.15

astar	***gobmk***	***biHarmonic***	***mutualInfo***	***ricianDenoise***	***regionGrowing***
10.5	4.87	0.03	1.95	0.03	0.04
10.3	2.65	0.03	0.03	0.02	0.01

C. Energy Comparisons

In addition to the L1 data cache energy discussed in Section IV.A, we also obtain the dynamic and leakage energy data of other memory subsystem components including the L1 instruction cache, L2 cache and the main memory through Cacti [20] and McPAT [27]. Given these energy data, we record the access times to the logics and storages in our simulations and back-annotate them to our energy estimation models to generate the energy results for each design.

The energy comparison results are shown in Figure 8 and are normalized to the baseline (design *N*). They are broken down into the dynamic energy of L1 cache (dominated by the L1 data cache), L2 cache and main memory, and the leakage energy. The designs *B* and *Vp* can reduce the L1 data cache miss rates, but with a higher per-cache-access energy. But they can still reduce the total energy in some cases by reducing the L2 cache energy (less access to L2 cache) and the leakage energy (less run time). Therefore, the average total energy overhead compared to baseline for designs *B* and *Vp* is 4% and 13%, respectively. By serializing the accesses to the regular L1 cache and victim cache, design *Vs* achieves an average total energy reduction of 3% compared to baseline. Design *R* achieves an average total energy reduction of 7% compared to baseline, mainly through moderately reducing the L1 miss rate and the run time.

With the additional energy of the SMLT, VTB, and adaptive mapping unit, AH-Cache can still achieve an energy reduction of 16%, 22%, 10% and 7% compared to designs *B*, *Vp*, *Vs* and *R*, respectively. It consumes less energy than designs *B*, *Vp* and *Vs* since its per-cache-access energy overhead is much less than the CAM in B-cache and victim cache. It consumes less energy than design *R* since its adaptive mapping more effectively reduces L1 miss and thereby consumes less L2 cache energy and leakage energy (less run time).

In summary, AH-Cache achieves energy-runtime-production reductions of 19%, 25%, 18% and 18% over the designs *B*, *Vp*, *Vs* and *R*, respectively. This verifies the energy efficiency of AH-Cache.

V. CONCLUSIONS

In this paper an adaptive hybrid cache called *AH-Cache* is proposed. By providing dynamic remapping of the SPM blocks onto cache blocks based on the run-time cache behavior in hardware, AH-Cache makes the software focus on the utilization of logically continuous SPM. Experimental results show that AH-Cache can achieve energy-runtime-production reductions of 19%, 25%, 18% and 18% over representative previous techniques. Thus AH-Cache can serve as an energy-efficient hybrid cache in low-power processors that require flexible SPM sizes to satisfy various application requirements, but have low cache associativity due to a tight power budget.

VI. ACKNOWLEDGMENTS

The authors thank Mishali Naik and Jiajun Zhang for helpful discussions and assistance in experiments, and the anonymous reviewers for their useful comments on this work.

REFERENCES

[1] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt, "DRDU: A data reuse analysis technique for efficient scratch-pad memory management," in *ACM Trans. Des. Autom. Electron. Syst.* vol. 12.2, p.15, 2007.

[2] D. Chiou, P. Jain, L. Rudolph, and S. Devadas, "Application-specific memory management for embedded systems using software-controlled caches," in *Proc. DAC*, 2000, pp. 416-419.

[3] C. Moritz, M. Frank, and S. Amarasinghe, "FlexCache: A framework for flexible compiler generated data caching," in *Lecture Notes in Computer Science*, vol. 2107, pp.135-146, 2001.

[4] P. Ranganathan, S. Adve, and N. Jouppi, "Reconfigurable caches and their application to media processing," in *Proc. ISCA*, 2000, pp. 214-224.

[5] H. Cook, K. Asanović, and D. Patterson, "Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments," Technical Report No. UCB/EECS-2009-131, 2009.

[6] J. Robertson and K. Gala, "Instruction and data cache locking on the e300 processor core," Freescale Application Note, 2006.

[7] T. Kluter, P. Brisk, P. Ienne, and E. Charbon, "Way stealing: Cache-assisted automatic instruction set extensions," in *Proc. DAC*, 2009, pp. 31-36.

[8] M. Qureshi, D. Thompson, and Y. Patt, "The V-way cache: Demand based associativity via global replacement," in *Proc. ISCA*, 2004, pp. 544-555.

[9] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *Proc. ISCA*, 2000, pp. 107-116.

[10] D. Rolán, B. Fraguela, and R. Doallo, "Adaptive line placement with the set balancing cache," in *Proc. MICRO*, 2009, pp. 529-540.

[11] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. ISCA*, 1990, pp. 364-373.

[12] C. Zhang, "Balanced cache: Reducing conflict misses of direct-mapped caches," in *Proc. ISCA*, 2006, pp. 155-166.

[13] B. Agrawal and T. Sherwood. "Modeling TCAM power for next generation network devices," in *Proc. ISPASS*, 2006, pp.120-129.

[14] M. Zhang and K. Asanovic, "Fine-grain CAM-tag cache resizing using miss tags," in *Proc. ISLPED*, 2002, pp.130-135.

[15] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap data allocation to scratch-pad memory in embedded systems," in *J. Embedded Comput.*, vol. 1.4, pp. 521-540, 2005.

[16] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel, "Scratchpad sharing strategies for multiprocess embedded systems: A first approach," in *Proc. ESTMEDIA*, 2005, pp. 115-120.

[17] J. Cong, H. Huang, C. Liu, and Y. Zou, "A reuse-aware prefetching algorithm for scratchpad memory," to appear in *Proc. DAC*, 2011.

[18] LLVM compiler infrastructure: http://llvm.org/

[19] T. Austin and G. Sohi, "Zero-cycle loads: microarchitecture support for reducing load latency," in *Proc. MICRO*, 1995, pp.82-92.

[20] HP Cacti, http://quid.hpl.hp.com:9081/cacti/

[21] C. Kun, S. Quan, and A. Mason, "A power-optimized 64-bit priority encoder utilizing parallel priority look-ahead," in *Proc. ISCAS*, 2004, pp. II 753-756.

[22] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," in *IEEE Computer*, vol. 35, pp. 50-58, 2002.

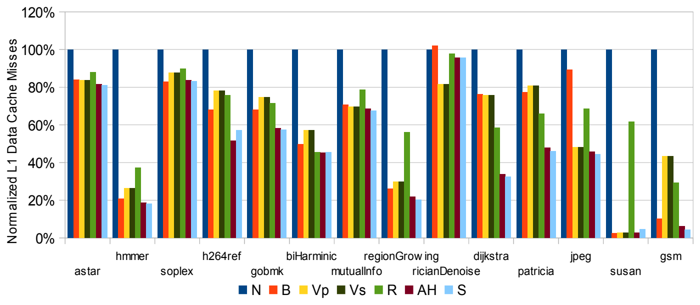
[23] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," in *Computer Architecture News*, pp. 92-99, 2005.

[24] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Workshop on Workload Characterization*, 2001, pp. 3-14.

[25] S. Bird, A. Phansalkar, L. K. John, A. Mericas, and R. Indukuru, "Performance characterization of SPEC CPU benchmarks on Intel's core microarchitecture based processor," in *SPEC Benchmark Workshop*, 2007.

[26] www.cdsc.ucla.edu

[27] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. MICRO*, 2009, pp. 469-480.



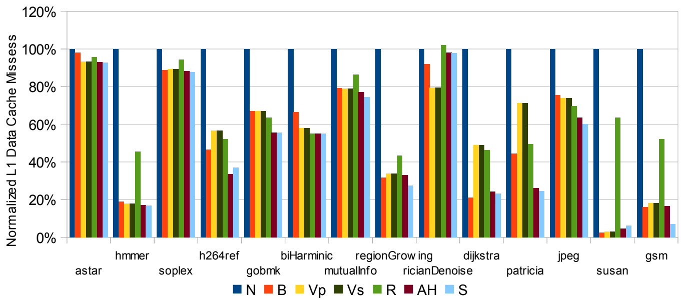
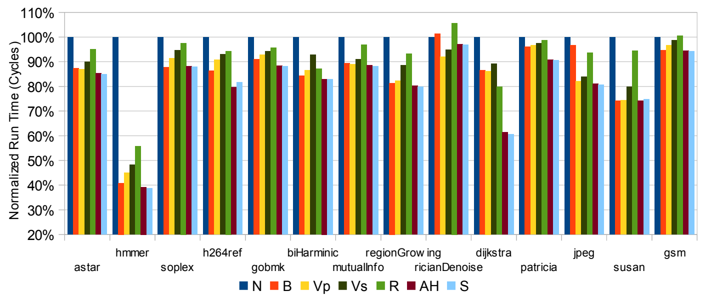


Figure 6. Comparison results of L1 data cache (hybrid cache) misses (left: 16KB, right: 32KB).



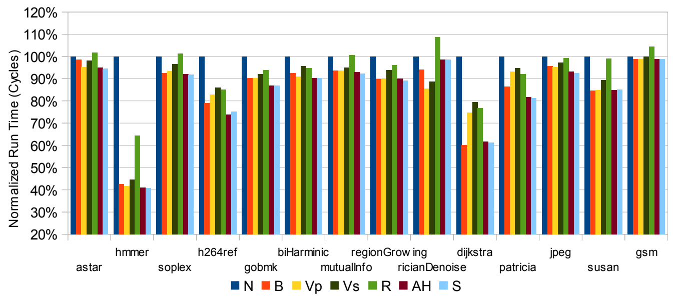
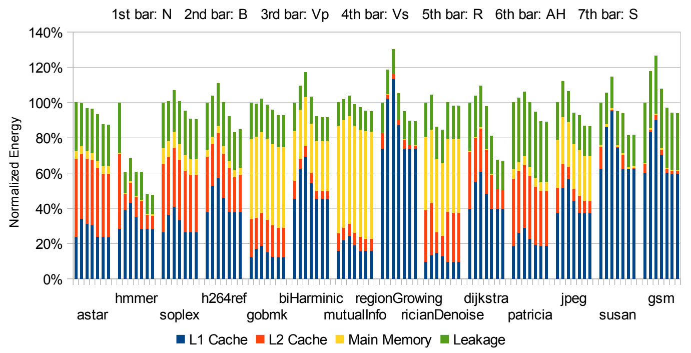


Figure 7. Comparison results of run time (left: 16KB, right: 32KB).



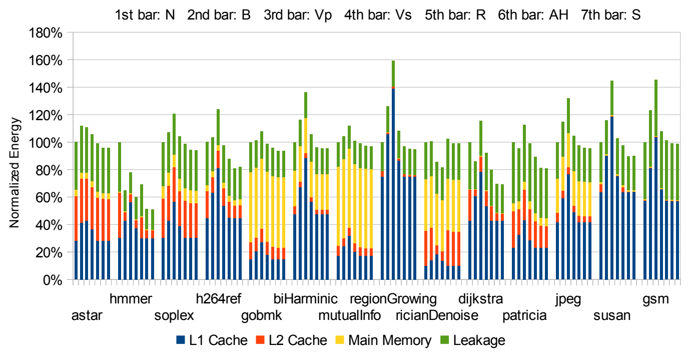


Figure 8. Comparison results of memory subsystem energy (left: 16KB, right: 32KB).