ROBTIC: An On-Chip Instruction Cache Design for Low Power Embedded Systems

Ji Gu, Hui Guo and Patrick Li
School of Computer Science and Engineering
The University of New South Wales, NSW 2052, Australia
{jigu, huig}@cse.unsw.edu.au

Abstract—**The on-chip instruction cache is a potential power hungry component in embedded systems due to its large chip area and high access-frequency. Aiming at reducing power consumption of the on-chip cache, we proposed a Reduced One-Bit Tag Instruction Cache (*ROBTIC*), where the cache size is judiciously reduced and the cache tag field only contains the least significant bit of the full tag. We developed a cache operational control scheme for *ROBTIC* so that with the one-bit cache tag, the program locality can still be efficiently exploited. For applications where most of the memory accesses are localized, our cache is able to achieve similar performance as a traditional full-tag cache; however, the power consumption of the cache can be significantly reduced due to the much smaller cache size, narrower tag array (just one bit), and tinier tag comparison circuit being used. The experiments on a set of benchmarks demonstrate that our approach can reduce up to 25.8% power consumption and 30.9% area of the traditional cache when the cache size is fixed at 32 instructions. With the cache size customization, a further 48% power saving can be achieved.**

I. Introduction

Cache has been widely used in modern computer architectures to bridge the performance gap between memory and processors. It also plays an important role in reducing power consumption of embedded systems, where low power is essential for a long battery life. However, in comparison with other on-chip components, the traditional cache takes a large portion of the chip area and still consumes a significant amount of power. This is especially evident in the instruction cache (I-Cache), where frequent instruction fetching by the microprocessor causes a large amount of switching power.

Take a conventional direct-mapped I-Cache of 1024 entries as the example. Each entry consists of a valid bit, a tag field, and a cache line to store memory data. The tag comparator is used for cache hit detection. Corresponding to a cache operation, the 32-bit PC address is split into 2 bits of byte offset for an instruction, 10 bits of index for a cache entry and 20 bits of tag for a cache map to the memory location. With this design, the tag takes up $20/(1+20+32) = 37.7\%$ of the whole cache area and entails a 20-bit tag-comparator. As can be seen, the larger the cache tag, the more area and power consumed.

To reduce the adverse effect of the cache tag, Scratchpad Memory (SPM) was introduced [1]. SPM uses part of the memory space and requires no mapping between SPM and the main memory, thus eliminating the tag and tag comparison. Since the SPM is not transparent to the software, a dedicated compiler is required to statically allocate code and data into SPM [2]. The static allocation greatly reduces the flexibility of using on-chip memory to store any part of memory data. Recent developments [3] [4] [5] on dynamically updating SPM with special instructions have improved the use of SPM; however, the associated hardware/software design effort and increased code size still impose limitations on SPM applications.

Since application code often comprises numerous *loop basic blocks* that exhibit high spatial and temporal locality (namely, once executed, the block of instructions will be repeatedly fetched), we can exploit such a locality for cache tag reduction.

In this paper, we propose a Reduced One-Bit Tag Instruction Cache (***ROBTIC***) that has **the low power feature of a SPM while offering the memory mapping flexibility of a traditional cache**. We develop a cache operational control scheme for *ROBTIC*, which can effectively explore the temporal and spatial locality of the application program so that cache power is reduced without sacrificing the cache performance.

The rest of the paper is organized as follows. Section II reviews some existing power optimization methods for I-Cache design. The structure and working principle of our *ROBTIC* cache, as well as the design approach to exploit the program locality for high cache performance, are given in Section III. Section IV presents the experimental setup, simulation results and related discussions. The paper is concluded in Section V.

II. Related Work

Many techniques have been proposed to improve the traditional instruction cache design for power efficiency. Kin et al. [6] introduced a filter cache to store recently accessed cache blocks. As the filter cache is small in size and has lower load capacitance, each access consumes less power than that of the normal cache. Block buffering [7] is another approach similar to the filter cache. Small buffers are associated with the first level cache to store the previously hit cache data block. In case the next requested data are in the buffer, there is no need to access the cache again, thus data access would be faster and consume less power.

Based on the filter cache idea, Bellas et al. [8] introduced an L0-cache that stores frequently executed instruction sections. In case there is a jump from one section to another in the execution flow, a prediction strategy is used to fetch the target instruction block into the L0-cache. This approach enables most of the instructions to be fetched from the smaller L0-cache, thus reducing the power dissipation. Similarly, the authors in [9] proposed a Loop cache for storing frequently executed instruction loops. This approach can achieve high power efficiency for applications in the digital signal processing (DSP) domain that features large number of loop executions.

A common characteristic of the approaches discussed above is that an extra smaller cache is placed in front of the traditional L1 cache. Rather than introducing an extra small cache, another class of approaches for cache power optimization focuses on the cache operation control and structure reduction. In [10], the author proposed to sequentially access cache tag and data array. The cache tag comparison is performed first and data will be fetched from the data array only after a cache hit is detected. A mismatch from the tag compare means there is no data fetched from the cache, thus power can be saved. Panwar et al. [11] proposed a technique to buffer the address of the previously hit cache line of the instruction cache. In case the previous one is a non-branch or un-taken branch instruction and address of the next cache access indicates the same line, a hit can be guaranteed and there is no need to access the tag array again and repeat the tag-compare.

In [12], based on the observation that the instruction cache conflicts hardly take place for a small application, the authors proposed to enable only one way in the set associative cache to make the cache work like a direct mapped cache. Thus, the amount of tag comparison can be reduced for power efficiency.

The approach proposed in [13] and the way-halting cache [14] design use a couple of the least significant bits of the tag for comparison. An extra array is employed to store the partial tag bits from each way and is always accessed first to compare with the corresponding bits of the memory location reference. Once the partial bits comparison suggests a cache miss for that way, no further tag comparison will be performed.

Our approach is similar to those in [13] [14] in a sense that all aim to cache tag reduction. However, a key difference exits: we do not keep the full tag in the cache tag field, nor do we use any extra tag memory structures for the partial tag bits as this incurs un-neglectable overhead of area and power consumption. On the contrary, our approach reduces the tag array in the cache to a vector of 1-bit. We develop a dynamical cache operational control scheme to effectively exploit the program locality so that the cache power consumption can be significantly reduced without introducing memory power consumption overhead.

III. ROBTIC Cache

In this section, we first present the high level structure of our Reduced One-Bit Instruction Cache and highlight its architectural differences. The design issues of the new architecture are then discussed, followed by detailed solutions.

A. Architecture

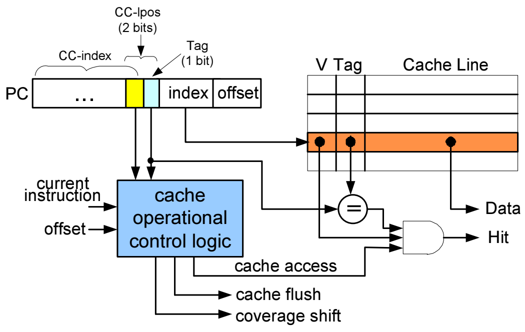

Fig. 1. ROBTIC Cache Architecture

Fig. 1 shows the general structure of *ROBTIC* cache. Like a traditional cache, it contains an array of cache entries with each entry indexed by a set of bits in the PC address (labeled as *index*). The width of the *offset* in PC is determined by the cache line size. The tag is used together with the valid bit for cache hit checking. With a cache hit, an instruction can be fetched from a valid cache entry. Unlike a normal design, our cache has a number of customizations and features:

- The tag for each cache entry is limited to one and only one bit;
- The cache size is extensively reduced;
- Due to the above extreme customizations, a cache operational control unit is introduced, which uses two bits of the *PC address* and the current instruction type to dynamically control the cache operation;
- With the proposed cache operational control unit, the high cache performance is retained;
- Because of the reduced cache and maintained performance, the cache power consumption is effectively reduced.

The idea behind ROBTIC cache is to exploit the distributed locality of applications. For an application program, usually there are a large number of loop basic blocks. Loop basic blocks present a high temporal and spatial locality. If those loop blocks can be cached during their execution, a high cache hit rate can be achieved. Surveys [15] [16] of a wide range of benchmark applications show that applications spend most of the time within certain blocks and more than 90% of loops contain no more than 30 instructions. Therefore, a *ROBTIC* cache size of 32 instructions (the closest power value of 2) can be expected to capture most of the temporal and spatial locality during the execution. We define this size as the *standard cache size* for *ROBIC*.

The cache hit rate is also closely related to the cache configuration and replacement policy. For simplicity, we assume that the instruction cache is directly mapped (therefore, the replacement policy issue is avoided); we also assume that a cache line holds multiple instructions, and all instructions are of equal size, as can be found in most RISC machines that are popular in embedded systems.

We will refer to the architecture in Fig. 1, when elaborating the design in following sections.

B. Cache Tag Size vs Cache Memory Coverage

The address tag field in a traditional cache is used to distinguish which memory location that a cache entry is currently mapped to. The full tag size is decided by the ratio of the cache size and memory size. With the full tag, cache can store data of any memory location; namely the cache has a full memory coverage. We define a *cache memory coverage* (or simply ***cache coverage***) as the cache mappable address space of the main memory.

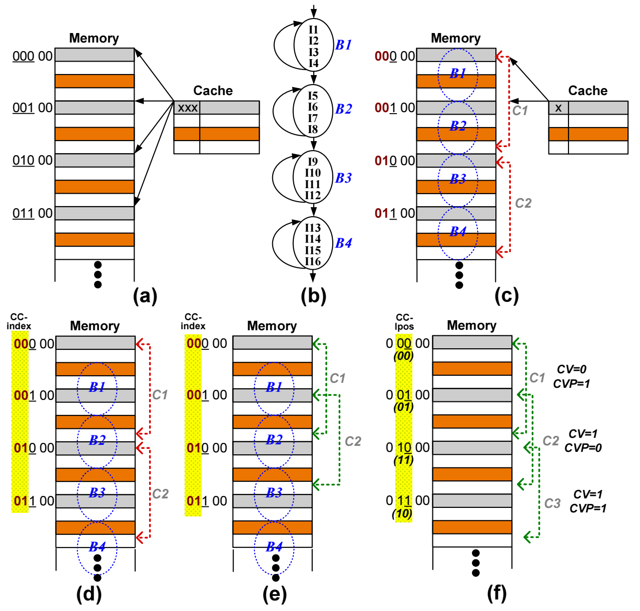

Fig. 2. (a) Full Cache Coverage (b) A program Example (c) Reduced Cache Coverage with 1-bit Tag (d) Ping-Pong Effect (e) Overlapped Cache Coverage (f) CV and CVP values for Adjacent Coverage Regions

Fig. 2 (a) illustrates a traditional cache design with a full cache tag. It contains 4 entries. Given a memory space of 32 entries, the full cache tag is 3 bits (as indicated by 3 x's in the cache tag field) in order to map any entries in the memory. The three most significant bits (underlined) in the memory address correspond to the tag value of a cache entry.

If the 1-bit tag is used, the cache can only be mapped to a segment of memory, as illustrated in Fig. 2(c), where the cache can, for instance, cover region $C1$ or $C2$, but not both at the same time. Each of the mappable regions can be identified by the most significant two bits of the memory addresses: for example, "00" for $C1$ and "01" for $C2$. We call this most significant bit set that can uniquely identify such a cache coverage, as the *cache coverage index* (***CC-index***), as has been used in Fig. 1.

If the program code is small enough that its memory location can be covered by the cache, the reduced 1-bit tag will probably not affect the cache performance. However, this case is very unlikely. Program code may span over a large memory section; any access to the code outside of the cache coverage will suffer a cache miss. To make any memory location of the program code mappable to the cache, we introduce a *dynamic cache coverage* scheme, where the cache coverage is dynamically changed with the PC address during program execution.

To explain, we again refer to the example in Fig. 2(c). Assume the section of memory stores a program of four loop blocks: $B1$-$B4$, as given in Fig. 2(b). Each block consists of four instructions. The cache coverage is initially $C1$ with *CC-index*="00"; when executed, the first block ($B1$) can be fully cached and the access to any instructions in this block can be performed on the cache. After the first block is finished, the second block ($B2$) overwrites the first block in the cache. When the third block ($B3$) starts, the cache coverage is changed to $C2$ (the *CC-index* is now "01") and all cache contents belonging to the previous coverage will be invalidated.

In an ideal case, such as the above example (Fig. 2(c)), where each loop block can be perfectly mapped to the cache, the locality of the loop blocks can be maximally explored with a small cache and 1-bit tag.

This ideal case is, however, very unlikely for real applications, where a loop block may reside randomly across different cache coverage regions, like $B2$ in Fig. 2(d). When block $B2$ is executed, the first half and second half of the block will evict each other from the cache for each execution iteration (a.k.a the ***Ping-Pong effect***) and the instructions can never be fetched from the cache in this case.

To overcome such a problem, we propose an overlapped dynamic cache coverage control scheme.

C. *Overlapped Dynamic Cache Coverage Control*

In order to eliminate the *Ping-Pong* effect and effectively exploit the temporal and spatial locality of loop blocks, we dynamically shift the cache coverage in an overlapping manner so that instructions at the previous mappable memory addresses can still be survival in the new coverage, as illustrated in Fig. 2(e), where two possible cache coverage regions, $C1$ and $C2$ are overlapped. During execution of block $B2$, when the PC points to the instructions outside the cache coverage $C1$, the new coverage $C2$ is used. Since $C2$ still partially covers the $C1$ region, all instructions for the loop block are retained in the cache for the following execution iteration. Therefore, the temporal and spatial locality of the loops is effectively exploited, achieving a high cache hit rate.

The control flow for the dynamic cache coverage operation is given in Fig. 3, which determines the cache in three different operation states: *cache coverage shift*, *cache flush*, and *normal cache access*.

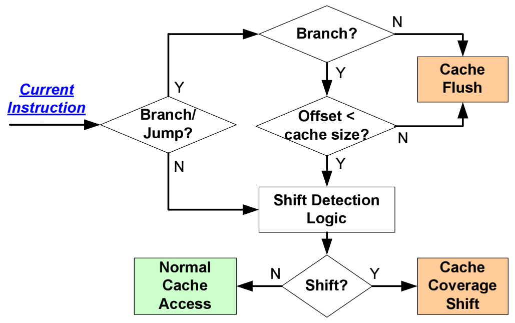

Fig. 3. Cache Operation Control Flow

Cache Coverage Shift

When the execution, or the current PC address, moves out of the current cache coverage, the coverage should be modified in order to make the new memory region mappable by the cache. We refer to such a change of cache coverage as *cache coverage shift* or simply ***CCShift***. CCShift invalidates (or flushes) cache entries that belong to the previous but not current coverage. Since a loop block (normally of the size smaller than the cache according to the cache size design discussed in the beginning of Section III) can be positioned in any location within a cache coverage, to ensure a block that traverses over two adjacent coverage regions is not flushed during its execution, we define that a *CCshift* always moves the cache coverage to its neighboring memory region by a cache size.

Cache Flush

In case the execution flow is diverted by a control instruction, such as *jump* or *branch*, the new cache coverage may be far away from the current coverage. In this case, all entries in the cache should be flushed (*cache flush*). To simplify the control logic, we assume that a *jump* instruction always causes a long distance hop, which often is the case. When a *jump* instruction, or a *branch instruction* with an offset larger than the cache size, is encountered, the cache is flushed. The *cache flush* invalidates all cache entries.

Normal Cache Access

When the execution advances, either in a sequential or branching manner, within the cache coverage, the cache is operated like a traditional cache access (henceforth called *normal cache access*). If the PC address is in the current cache coverage range and there is a cache hit for the current instruction, the instruction is fetched from the cache; otherwise, a memory access is incurred and the new instruction is cached.

The implementation of *normal cache access* and *cache flush* is the same as that used in a traditional design. Here, we only discuss the design for *cache coverage shift* and related shift condition detection.

D. *Cache Coverage Shift and Shift Detection*

With the overlapped dynamic cache coverage, we cannot use *CC-index* (see Fig. 1) to identify the cache coverage since it is not unique in a cache coverage, as can be seen in the example in Fig. 2(e), where the *CC-index* changes from "00" to "01" within the cache coverage region $C2$.

To identify such an overlapped cache coverage, a possible straightforward solution is to use the top and bottom locations to gauge the full-tag range of the coverage. To know if an instruction-fetch causes a coverage shift, one can check whether the related tag value in PC is within the tag range; if it is, then a normal cache operation is performed; otherwise, the coverage is shifted to a new coverage region, as outlined in Fig. 4(a). Here we assume *top* $<$ *bottom*. This design requires two registers for the top and bottom addresses of the current cache coverage, and two comparators for coverage range checking; they are all of the full-tag size. Hence, the overheads incurred from this design may cancel the savings from the 1-bit tag comparison.

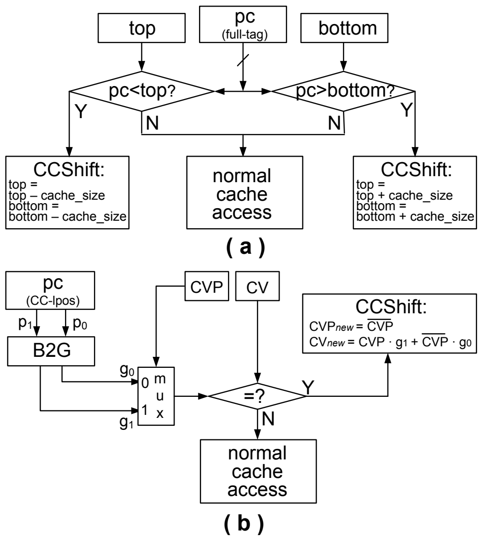

Fig. 4. Shift Detection (a) Design 1 (with Range Gauge) (b) Design 2 (with Gray-Encoding)

Since the cache coverage shift is only performed for sequential instructions or branch instructions within an offset smaller than the cache size, for a *CCShift*, only three consecutive coverage regions need to be locally differentiated, for which the least two significant bits of the full tag are sufficient. We call the two least significant bits the *local cache coverage position* (labeled as ***CC-lpos*** in Fig. 1); its value repeats every four consecutive regions when the cache coverage moves along in the memory. With *CC-lpos*, the pure relationship of that *bottom* $>$ *top* for a coverage will not hold. Therefore, use of range checking, as proposed in Fig. 4(a) is not applicable.

Here we present a different design, where the Gray-encoded *CC-lpos* is used to identify a coverage. To check whether the current execution is in the cache covered region, only one 1-bit comparison, rather than two multi-bit comparison, is performed. The design is detailed below.

We take the 2-bit *CC-lpos* (denoted as p_1p_0) from the PC and convert its binary value into Gray Code value, g_1g_0. With the Gray encoding, when the cache coverage shifts along the memory in an overlapped fashion, adjacent coverage regions can be differentiated by a bit in the *CC-lpos* Gray code that has a common value in a coverage region (hence we call it ***common value bit***).

Refer to Fig. 2(f) for an example, where the *CC-lpos* (i.e. p_1p_0) is bits 3&2 of the memory address (the two bit columns are highlighted in the figure), the related Gray code (g_1g_0) is given below each *CC-lpos* binary value. For coverage $C1$, g_1 is the bit of a common value (='0'). However, for coverage $C2$, g_0 is the common value bit (of value '1'). The common value (denoted as ***CV***) and its bit position (denoted as ***CVP***) can uniquely identify a coverage for any four consecutive regions. The design is given in Fig. 4(b). The CV and CVP are initially computed with the following logic:

$$\begin{cases} CVP & = \overline{g_1 \oplus g_0} = \overline{p_0} \\ CV & = g_0 = p_1 \oplus p_0. \end{cases}$$

For each new PC address, its *CC-lpos* bits, p_1p_0, are first converted into 2-bit Gray code, g_1g_0. The *CVP* of current cache coverage selects the common bit (from either g_1 or g_0) to compare it with the common value (CV) of the current cache coverage. If they are different, the cache coverage should be shifted; otherwise, a normal cache access is performed.

When the cache shifts to a new coverage, the common value bit for the new coverage is always different from the previous coverage, and the common value for the new coverage is the g_0 value if the previous coverage common bit is specified by the p_0 value, otherwise, the g_1 value if the common bit position is the inverse of p_0. Therefore, the CV and CVP for the new coverage are

$$\begin{cases} CVP_{new} & = \overline{CVP} \\ CV_{new} & = CVP \cdot g_1 + \overline{CVP} \cdot g_0. \end{cases}$$

In comparison with the design given in Fig. 4(a), the complexity of this proposed design (Fig. 4(b)) does not increase with the tag size, it only requires a 2-bit register, a 2-bit binary-to-Gray converter, a 2-to-1 multiplexor of 1 bit, and a 1-bit comparator. They are small, incurring little chip area and power overheads.

IV. Experimental Results

To examine the power efficiency of our *ROBTIC* cache, we applied the new cache design for a set of applications from Motorola's Powerstone [17] and MiBench [18] benchmark suites, as well as some other kernel-like applications. These are usually utilized in automotive control, image processing, DSP applications. The reference input data of each program are used in our experiments.

The commercial tool ASIPMeister [19] is used to generate the processor VHDL model as the platform for the applications. The MIPS32 ISA [20] is selected as the target processor instruction set architecture. The experiment starts with a given application written in C, compiled by the mips-gcc cross compiler and then simulated on the VHDL model. The functional correctness of the VHDL model is verified by comparing the result from simulation on the processor VHDL model with that from the MIPS32 software simulation of each application. The *ROBTIC* cache and the traditional cache have separately been implemented and integrated in the processor model for simulation. Each design is synthesized by Synopsys Design Compiler targeting Tower 0.18-micron standard cells, which provides the estimated area and delay for each cache design. The cache power consumption is obtained from the Synopsys PrimePower.

A. Implementation of Cache Coverage Shift

To investigate the cache coverage shift mechanism presented in Section III-D, we implemented the two design methods, one with the *range gauge* and another with the *Gray-encoding*. Table I gives the simulation results for the chip area, power consumption and delay of each design (last three columns). All designs are based on the 32-bit PC address and the cache of 16 entries, each 8 bytes long. The power consumption was measured when application *bcnt* was executed. For a comparison, the data for the full-tag compare design used in the traditional cache design is also given in the second row of the table. The power consumption and relative saving of the two designs as compared with the full-tag comparison design are presented in rows 3-6, respectively.

TABLE I
RESULTS OF CACHE COVERAGE SHIFT IMPLEMENTATION

		Area[μm^2]	**Power**[mW]	**Delay**[ns]
Tradt. Cache	Full-Tag Comp.	19076	3.17	1.93
ROBTIC	Range-Gauge	15654	2.35	1.72
	Impr.[%]	17.9	25.9	10.9
	Gray-Encoding	13186	2.29	1.89
	Impr.[%]	30.9	27.8	2.1

From Table I, we can see due to the reduction in tag size from 25 bits to 1 bit and the resulting decrease in overall logic complexity, both designs proposed for the new I-Cache architecture have less area and delay, and consume less power than the traditional design. The *Gray-encoding* approach brings a higher improvement than the *Range-Gauge* in terms of area and power consumption. The smaller delay improvement of the *Gray-encoding* is due to the extra level of Gray encoding logic.

B. General ROBTIC with Standard Cache Size

To see whether the design of *ROBTIC* is power efficient for any applications, we initially implemented the *ROBTIC* design with a standard cache size: 16 entries and each entry of 8 bytes, which can hold 32 instructions total. For comparison, the traditional I-Cache and the cache design with 5-bit Partial Tag Compare (PTC-5) that was proposed in [13] [14], were also implemented. All designs are of the same cache size.

Table II shows the results of the three designs tested on a set of applications (listed in Column 1). The cache performance in terms of hit rate for each design is given in Columns 2, 3 and 5 respectively; the performance improvement of the *PTC-5* and *ROBTIC* as compared with the traditional design is shown in Columns 4 and 6. The power consumption data are presented in Columns 7-11. It can be seen that, like the *PTC-5*, the *ROBTIC* (with only 1-bit tag) can achieve the same hit rate as the traditional cache with the full tag. The four exceptional cases are *jpeg*, *pocsag*, *qsort* and *stringsearch*, where the program locality cannot be effectively captured. This is due to a hefty number of function calls generated during execution, either from many infrequent blocks in a large application (such as *jpeg* and *pocsag*) or from the frequent loop blocks of a small application (such as *qsort* and *stringsearch*). For each function call, jumping to the entry of a function and returning to the calling program force the cache coverage to be flushed, greatly degrading the cache performance. However, for the other applications no matter large (e.g. *adpcm*) or small e.g. *des*, our proposed one-bit tag ROBTIC design can achieve the same performance as the traditional cache design.

From Table II, we can see the power reduction is considerable – from 23.4% of the *idct* to maximally 27.8% of the *bcnt* application, with an average of 25.8%. Note this average number does not include the four cases (underlined in Table II) where the cache power saving is at the cost of performance[1]. In comparison, the partial tag compare scheme achieves little power savings, with a maximum of 2.18% power reduction. This is because the partial tag scheme uses an extra tag array on top of the full-tag array in the cache tag field. An unmatch of the partial tag-compare prevents further access to the cache component and power is thus saved. However, if the partial tag-compare returns a match, the full tag comparison will still be performed. With the high cache hit rate, the full-tag cache field is frequently accessed and compared, therefore, little cache power can be saved. In comparison, our proposed *ROBTIC* does not introduce any extra partial tag array. On the contrary, we cut the tag field in the cache to 1 bit. This considerably reduces the total switching capacitance and hence power consumption.

[1]In these four cases, the reduced cache hit rate or increased cache misses incur extra accesses to the main memory, hence additional memory power consumption. Therefore, the comparison for power savings between different cache designs must be based on the same cache performance.

C. Application Specific ROBTIC with Customized Cache Size

To see whether the choice of the standard 32-instruction for the cache size of *ROBTIC* is most effective, we explored a small design space with the size ranging from 8 to 128 instructions to obtain a smallest cache that still maintains high cache performance for a given application. We observed that the 32-instruction cache designs generally provide a good trade-off between cache performance, and power consumption. But for some applications, the cache size can be further reduced for low power without sacrificing the cache performance.

Table III shows the further improvements of cache performance and power savings when *ROBTIC* cache size is customized. The cache performance and power consumption for the standard *ROBTIC* designs with fixed 32 instructions (labeled *standard* in the table), and for the custom designs (see label *customized*) with the cache size given in Column 2, are presented in Columns 3-8. As can be seen from the results, applications *adpcm*, *engine* and *stringsearch* have a customized cache size of still 32 instructions, so no further improvement is obtained. For applications *bcnt*, *crc*, *des* and *matrixmul* , where the customized size is larger than the standard (64/128 vs 32), a higher cache hit rate can be achieved. Though the power consumption of the cache itself is higher than that of the standard cache, the higher cache hit rate leads to large power savings on the main memory power consumption – a worthy deed for overall system power reduction. For the other six applications, however, the customized *ROBTIC* size is smaller but can achieve the same hit rate and bring a further power saving of about 48%.

V. CONCLUSIONS AND FUTURE WORK

This paper targets instruction cache power reduction for embedded systems and presented a reduced 1-bit tag I-Cache design (*ROBTIC*), where the cache size is reduced based on the basic loop block size of application programs, and only 1 bit tag is used.

We developed a dynamic cache coverage control scheme to effectively exploit the program locality so that the high cache performance is maintained and the memory power consumption overhead is mostly eliminated. We presented an innovative logic design for such a control scheme; the complexity of the control logic is independent from the cache and memory size, hence the design is highly efficient and scalable. For embedded applications where localization of most instructions execution can be expected, the proposed 1-bit tag I-Cache can achieve the similar performance as the normal cache but power consumption and area overhead can, on average, be reduced by 25.8% and 30.9%, respectively. With the cache size customization, a further 48% power saving can be achieved.

It should be noted that, the design approach proposed in this paper assumes that *jump* instructions always cause a long distance hop and force the whole cache to be flushed, which can be seen as a limitation of our approach. This restriction will be

TABLE II
RESULTS OF REDUCED ONE-BIT TAG INSTRUCTION CACHE: HIT RATE AND POWER CONSUMPTION

Application	Hit rate [%]					Power [mW]				
	Trad. I-Cache	PTC-5	Δ	ROBTIC	Δ	Trad. I-Cache	PTC-5	Δ	ROBTIC	Δ
adpcm	70.1	70.1	0	70.1	0	3.12	3.06	1.92%	2.27	27.2%
bcnt	49.5	49.5	0	49.5	0	3.17	3.16	0.32%	2.29	27.8%
blit	99.7	99.7	0	99.7	0	2.92	2.86	2.05%	2.21	24.3%
crc	87.3	87.3	0	87.3	0	3.03	3.00	0.99%	2.31	23.8%
des	49.5	49.5	0	49.5	0	3.18	3.17	0.31%	2.30	27.7%
engine	55.7	55.7	0	55.7	0	3.21	3.14	2.18%	2.35	26.8%
fir	90.1	90.1	0	90.1	0	2.99	2.97	0.67%	2.22	25.8%
gcd	77.8	77.8	0	77.8	0	3.08	3.05	0.97%	2.27	26.3%
idct	95.6	95.6	0	95.6	0	2.95	2.94	0.34%	2.26	23.4%
jpeg	63.1	63.1	0	45.8	-17.3%	3.17	3.17	0.00%	2.33	*26.5%*
lms	90.7	90.7	0	90.7	0	2.99	2.96	1.00%	2.24	25.1%
matrixmul	94.1	94.1	0	94.1	0	2.93	2.91	0.68%	2.19	25.3%
pocsag	85.3	85.3	0	71.8	-13.5%	3.07	3.01	1.95%	2.32	*24.4%*
qsort	82.9	82.9	0	54.5	-34.3%	3.05	3.03	0.66%	2.29	*24.9%*
stringsearch	95.1	95.1	0	51.4	-45.9%	2.92	2.91	0.34%	2.31	*20.9%*

TABLE III
COMPARISON OF STANDARD AND CUSTOM ROBTIC DESIGNS

Application	Cache Size (Numbr. of Instrs.)	Hit rate [%]			Power [mW]		
		standard	customized	impr. [%]	standard	customized	impr.[%]
adpcm	32	70.1	70.1	0	2.27	2.27	0
bcnt	128	49.5	96.3	46.8	2.29	7.69	*-235.8*
blit	16	99.7	99.7	0	2.21	1.15	47.9
crc	64	49.7	95.5	45.8	2.31	4.04	*-74.9*
des	128	49.5	71.2	21.7	2.31	7.98	*-245.5*
engine	32	55.7	55.7	0	2.35	2.35	0
fir	16	90.1	90.1	0	2.22	1.17	47.3
gcd	16	77.8	77.8	0	2.27	1.21	46.7
idct	16	95.6	95.6	0	2.26	1.14	49.6
jpeg	16	45.8	45.8	0	2.33	1.21	48.1
lms	16	90.7	90.7	0	2.24	1.17	47.8
matrixmul	64	94.1	99.8	5.7	2.19	3.94	*-79.9*
pocsag	16	54.5	54.5	0	2.29	1.19	48.0
qsort	16	54.5	54.5	0	2.29	1.19	48.0
stringsearch	32	51.4	51.4	0	2.31	2.31	0

relaxed in our future study. Also for the future work, we will investigate the multi-way associative structure in *ROBTIC*.

REFERENCES

[1] R. Banakar, S. Steinke, B. sik Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems," in *International Symposium on Hardware/Software Codesign*, 2002.

[2] F. Angiolini, L. Benini, and A. Caprara, "Polynomial-time algorithm for on-chip scratchpad memory partitioning," in *Proceedings of CASES'03*, 2003, pp. 318–326.

[3] B. Egger, C. Kim, C. Jang, Y. Nam, J. Lee, and S. L. Min, "A dynamic code placement technique for scratchpad memory using postpass optimization," in *Proceedings of CASES'06*, 2006, pp. 223–233.

[4] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 2, pp. 472–511, 2006.

[5] S. Steinke, L. Wehmeyer, B. sik Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proceedings of DATE'02*, 2002.

[6] J. Kin, M. Gupta, and W. H. Mangione-Smith, "The filter cache: An energy efficient memory structure," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, 1997, pp. 184–193.

[7] C.-L. Su and A. M. Despain, "Cache design trade-offs for power and performance optimization: A case study," in *Proceedings of ISLPED'95*, 1995, pp. 63–68.

[8] N. E. Bellas, I. N. Hajj, C. D. Polychronopoulos, and G. Stamoulis, "Using dynamic cache management techniques to reduce energy in general purpose processors," *IEEE Transactions on VLSI*, vol. 8, no. 6, pp. 693–708, December, 2000.

[9] ——, "Architectural and compiler techniques for energy reduction in high-performance microprocessors," *IEEE Transactions on VLSI*, vol. 8, no. 3, pp. 317–326, June, 2000.

[10] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas, "Sh3: High code density, low power," *IEEE Micro*, vol. 15, no. 6, pp. 11–19, December, 1995.

[11] R. Panwar and D. Rennels, "Reducing the frequency of tag compares for low power i-cache design," in *Proceedings of ISLPED'95*, 1995, pp. 57–62.

[12] A. Malik, B. Moyer, and D. Cermak, "A low power unified cache architecture providing power and performance flexibility," in *Proceedings of ISLPED'00*, 2000, pp. 241–243.

[13] R. Min, Z. Xu, Y. Hu, and W. ben Jone, "Partial tag comparison: A new technology for power-efficient set-associative cache designs," in *Proceedings of the 17th International Conference on VLSI Design*, 2004.

[14] C. Zhang, F. Vahid, J. Yang, and W. Najjar, "A way-halting cache for low-energy high-performance systems," *ACM Transactions on Architecture and Code Optimization*, vol. 2, no. 1, pp. 34–54, March, 2005.

[15] C.-L. Yang and C.-H. Lee, "Hotspot cache: Joint temporal and spatial locality exploitation for icache energy reduction," in *Proceedings of ISLPED'04*, 2004, pp. 114–119.

[16] K. Ali, M. Aboelaze, and S. Datta, "Reducing energy in instruction caches by using multiple line buffers with prediction," *Lecture Notes in Computer Science, v 4759 LNCS.*, pp. 508–521, 2008.

[17] J. Scott, L. H. Lee, J. Arends, and B. Moyer, "Designing the low-power m-core architecture," in *International Sympsium on Computer Architecture Power Driven Microarchitecture Workshop*, 1998, pp. 145–150.

[18] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *IEEE 4th Annual Workshop on Workload Characterization*, 2001, pp. 83–94.

[19] M. Itoh, S. Higaki, Y. Takeuchi, A. Kitajima, M. Imai, J. Sato, and A. Shiomi, "Peas-iii: An asip design environment," in *Proceedings of the 2000 IEEE ICCD*, 2000, pp. 430 – 436.

[20] MIPS IV Instruction Set, Revision 3.2, September, 1995. MIPS Technologies, Inc. http://www.mips.com.