Energy Frugal Tags in Reprogrammable I-Caches for Application-Specific Embedded Processors*

Peter Petrov
University of California at San Diego
CSE Department
ppetrov@cs.ucsd.edu

Alex Orailoglu
University of California at San Diego
CSE Department
alex@cs.ucsd.edu

ABSTRACT

In this paper we present a software-directed customization methodology for minimizing the energy dissipation in the instruction cache, one of the most power consuming microarchitectural components of high-end embedded processors. We target particularly the instruction cache tag operations and show how an exceedingly small number of tag bits, if any, are needed to compute the miss/hit behavior for the most frequently executed application loops, thus minimizing the energy needed to perform the tag reads and comparisons. The proposed methodology exploits the fact that the code layout structure of the program loops can be identified after compile and link, and that it typically resides in a very confined memory location, for which very few bits from the effective address can be utilized as a tag. Subsequently, we present an efficient, programmable implementation to apply the suggested energy minimization technique. The experimental results show a significant decrease in energy dissipation for a set of real-life applications.

1. INTRODUCTION

The ever growing improvements in process technology have made the utilization of system-on-a-chip (SOC) design approaches highly attractive. Improved time-to-market, cost-efficient designs, easy design reuse, and flexible implementation constitute some of the many SOC advantages. Embedded processor cores are being utilized widely in such systems in order to achieve better time-to-market, lower design cost, and easily reprogrammable implementations. However, the increased silicon integration, together with the ever increasing clock frequencies, have led to proportional increases in terms of power consumption.

Minimizing power consumption is becoming one of the major requirements for a large class of products including PDAs, cell phones, laptop computers, and even high-end servers, as reduced power consumption translates to longer battery life and increased chip density due to reduced cooling requirements. Consequently, techniques for minimizing system power consumption are of crucial importance for product quality. Circuit-level power minimization techniques have been the dominant approach in designing energy efficient designs so far [1, 2]. However, architecture-level approaches are becoming in recent years popular due to their ability to eliminate redundancies on higher, microarchitectural levels, thus resulting in even greater power optimizations [3, 4, 5].

Hardware/software co-design methodologies are extremely important in designing complex SOCs comprising processor cores and dedicated ASIC modules. Processor speeds globally and performance predictability locally are important design trade-off characteristics taken into account in building such a hardware/software co-design system [6]. A fundamental issue is the partitioning problem of system functionality between hardware and software. While shifting as much as possible of the functionality onto software reduces system cost and time-to-market, performance and power consumption suffer, thus diminishing the advantages of general-purpose processors; the end result typically is either a costly system including a large amount of custom hardware, or an extremely power hungry design, utilizing processor cores of very high performance. Yet, processor cores are still the most viable solution in implementing complex SOCs if special care is taken to prevent the fundamental disadvantages of their general-purpose nature. Customizing the processor microarchitecture to particular application-specific needs has been shown to be an efficient technique for boosting processor performance and significantly reducing its power consumption [7].

In this paper, we propose a technique for software-controllable customization of the instruction cache (I-cache) subsystem of high-performance microprocessors, a microarchitectural component with significant contribution to the total power consumption [3]. We describe an architecture, capable of utilizing application-specific information in a programmable way; hence the ability to re-customize in a post-manufacturing fashion helps effectively cover diverse applications with no need for spinning new silicon, an important advantage in terms of flexibility for a number of high-performance embedded systems.

The proposed methodology utilizes information about the application loops and more specifically the possible I-Cache conflicts of loop code or functions called within the loop. The tag operations associated to the I-cache are typically designed for a worst case scenario and they utilize the entire effective address. In high-associativity caches, the tag length approaches the length of the entire effective address, which results in power expensive tag reads and comparisons.

The I-cache tag operations are not only quite expensive in terms of power but usually also highly redundant, as conflicting references are frequently close to each other in the address space, thus necessitating only a few tag bits for conflict identification. A straightforward optimization for an architecture with no fetch buffers, already proposed in [8], consists of avoiding tag operations when

*This work is supported by NSF Grant 0082325.

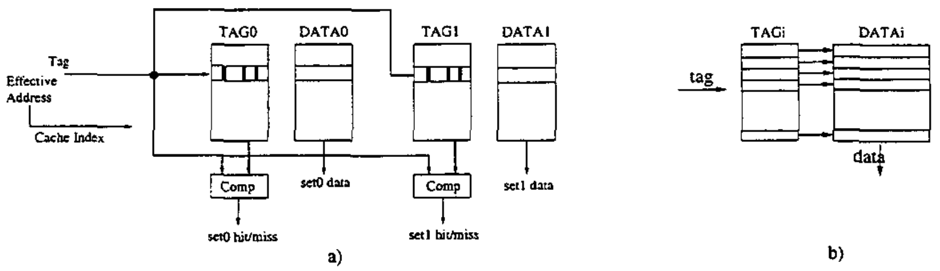

Figure 1: Set associative cache organizations

sequentially fetching within a cache line, and only performing tag checks when crossing cache line boundaries. The latter happens when a control changing instruction (branch, subroutine call, etc.) has been encountered, or the sequential program flow has reached a new cache line.

We show how depending on the code layout of the application loops, the minimal number of tag bits needed for identifying cache conflicts is determined. In the extreme case of a loop that fits in the I-cache, no tag operations are required at all for identifying conflicts.

The hardware support for the proposed methodology needs to be able to utilize only a certain number of least significant bitlines from the tag arrays and disable the rest. We complement our methodology with a discussion of such a hardware implementation for the proposed I-cache customization. Not only is the hardware solution efficient, but it is also programmable. Consequently, the described hardware implementation constitutes a unified microarchitectural solution, capable of handling a large set of important applications through in-field re-customization and of maintaining the market benefits of high-volume embedded processor production runs.

2. RELATED WORK

The area of architecture-level cache power optimizations is a relatively new one. In [9], a small and energy efficient L0 data cache is introduced in order to reduce power consumption of the memory hierarchy. The price paid is increased miss rate and longer access time. In [3], an L0 instruction cache with run-time techniques for fetching only the frequently executed basic blocks is proposed. The small size of this cache translates directly to power consumption reductions.

In set-associative cache designs, a significant amount of power is spent to access simultaneously all the cache ways and at the end select the data from only one of them. A *phased* cache design [10] can be used to alleviate the resulting power problem by accessing only the tag arrays initially. If a hit is identified in a particular associativity way, the data from this way is read in the next cycle. Therefore, the phased cache organization reduces the power consumption, while paying the price of an additional cycle to the cache access time. The *way-predicting* set-associative cache organization [11] attacks this problem by first predicting in which associativity way the data resides and consequently accesses only the data array of the predicted way. At the same time, the tag arrays are being read and the prediction validated. In the case of a misprediction, an additional cycle for accessing the data is needed if a cache hit has occurred in another associativity way, or the data is brought from the next level of the memory hierarchy in the case of a miss.

3. TAG MINIMIZATION

3.1 Cache organization

The I-cache is used to bring the executable code closer to the processor core, so that the time needed to fetch an instruction is minimized. Figure 1a outlines the architecture of a 2-way set associative cache. The referenced effective address, which in the case of the I-cache is the content of the Program Counter (PC), is separated into *block index*, *cache index*, and *tag*. The block index is used to address a word within a cache line, while the cache index is used to address the cache line; the tag field checks whether there is a conflict with a memory location with the same cache index. Increased set-associativity results in shorter cache index (smaller number of sets) and thus wider tags. In the extreme case of fully associative cache, almost the entire address (except the block index) forms the tag. The tag field associated to each cache line is stored in a separate tag memory array. Each time an access is performed to the I-cache, the tag associated to the cache line is read and compared to the tag of the effective address being referred.

A power efficent implementation of highly associative cache organizations was demonstrated in the design of the StrongARM microprocessor [12]. The tags corresponding to each associativity set are stored in a fully associative buffer, typically implemented as a CAM array. Figure 1b shows this structure for a particular associativity set. The tag look-up is performed in a fully associative manner in the CAM buffer and a hitting reference directly selects its corresponding data line from the data array. In this architecture most of the power is spent in performing the fully associative tag look-up, while the total power consumption is similar to that of a traditional 2-way set associative cache. Because of the high associativity, the tag length in this cache organization is significant; consequently, a technique that effectively reduces the number of tag bitlines being accessed and compared would have a crucial impact on the total power consumption of the cache.

Conceptually, the tag plays the role of a key for distinguishing two distinct memory addresses being mapped to the same cache line. If the referred locations are close in the address space, a large part of these tag fields is consequently redundant. Depending on the size of the memory region in which the memory locations reside and the cache organization, a very small number of least significant tag bits typically suffices to play the cache line "key" role and distinguish all the possible conflicting addresses for this line. A large amount of power is spent in reading, comparing and writing unnecessarily large tags, unless the aforementioned redundancy is eliminated.

3.2 Tag usage analysis

Programs typically spend most of their execution time in a small part of the application code. A well known rule of thumb is that "A program executes about 90% of its instructions in 10% of its code" [13]. Usually, this small part constitutes a set of loops that executes a limited number of static instructions. Therefore, the code executed in the loop spans a small fraction of the memory space. Figure 2 shows an example of such a loop and the code layout in the address space.

Figures 2b and 2c show the memory layout of the code from the example. The program memory space is divided into regions that correspond in size to one I-cache associativity-way and is aligned on the memory boundaries for which the *cache index* part of the address is zero. Consequently, for each of these memory regions the tag part of the address is a constant and this particular tag value is associated to addresses only within this memory region. Thereafter, we denote these memory regions as *0-tag regions*. It is evident that if a particular code resides within a 0-tag region, there will be no instruction cache conflicts, because all the addresses will be distributed to distinct cache lines, since the tag part of these addresses is a constant. Therefore, for such a code, no tag operations are needed and only the cold misses for the loop need to be handled.

Figure 2b shows a configuration in which the loop code resides within four 0-tag regions. In this situation cache conflicts are possible, since there might be references to addresses with differing tags. It is evident though, that in such a case the two least significant bits from the tags suffice for cache conflict identification. It can be observed that these two bits differ for all the tags within this memory region formed by four consecutive 0-tag regions. Therefore, these two tag bits provide a full resolution for the set of memory addresses referencing this memory region. In this case the remaining more significant tag bits are identical for all the tags, but as the next example shows, it is possible to have differing most significant tag bits and still achieve a complete cache conflict identification only through utilization of a small number of least significant bits.

Figure 2c shows a configuration in which the same loop spans another memory region again comprising four 0-tag regions. As in the previous example, the two least significant tag bits would be enough to provide a complete address resolution and serve as "reduced" tags for cache conflict identification. In this case though, the remaining most significant tag bits may contain varying values, which would still not prevent the utilization of only the two least significant bits as a conflict indicator. This is a direct consequence of the fact that the tag role in the cache designs is to provide separation between memory locations that can overlap in the cache. Therefore, any set of distinct "keys" associated to the memory addresses that overlap in the cache can be used as tags. Evidently, the two least significant bits in this example can play this role and provide complete resolution for cache conflict identification.

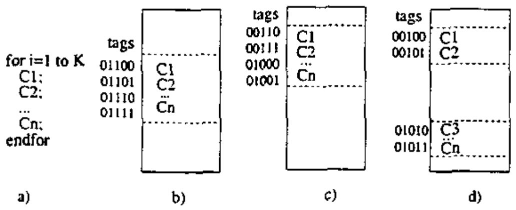

Figure 2: Memory layout alternatives

In all these examples, the loop code happens to occupy consecutive 0-tag regions. In such a case, it is evident that the number of least significant tag bits needed for complete tag resolution is $\lceil \log_2 n \rceil$, where n is the number of 0-tag regions. Since, a loop body can contain calls to functions placed by the linker in various memory locations, a situation with non-contiguous 0-tag regions needs to be considered as well. Figure 2d shows such a code layout. The first part of the loop code occupies two 0-tag regions, while the rest of the code resides in another two 0-tag regions, but separated from the first two. If one considers the unused memory space between the loop parts as a part of the loop, then the loop code spans a total of eight 0-tag regions, thus necessitating a total of three least significant tag bits for cache conflict identification. It is evident that this is a worst case scenario, and as the example shows, in the particular case actually only two least significant bits would suffice for complete tag resolution. Consequently, one can observe that for this general case of loop code layout the number of least significant tag bits is in the range $[\lceil \log_2 n \rceil, \lceil \log_2 m \rceil]$, where n is the number of 0-tag regions utilized by the loop code, and m the total number of 0-tag regions that the loop code spans in the memory space. The number depends on the particular tag values of the tag regions which the loop code occupies.

3.3 Removing the tag redundancy

The general-purpose cache architecture, seen from this perspective, is fundamentally a worst case assumption, with the program taking up residence within the entire addressable memory space. A large amount of redundancy in performing the tag operations therein exists, particularly if the program code is sparse but distributed haphazardly, with no consideration of dynamic code execution proximity, throughout the 0-tag regions.

A consequence of the above observations is that typically there exist large levels of redundancy in reading the entire tag from the tag array and comparing it to the effective address tag. Given that application-specific information about the loop code layout in the program memory is present during program execution, a large part of this redundancy can be eliminated, resulting in significant power savings. The application information can be obtained after compile/link time when the code layout is already known and provided to the I-Cache microarchitecture in a programmable fashion.

As discussed in the previous subsection, if the code of the frequently executed application parts resides within a 0-tag region in the memory space, then no I-cache conflicts are possible; hence no tag operations are needed. When the code spans n consecutive 0-tag regions, then exactly $\lceil \log_2 n \rceil$ least significant tag bits can be utilized as new shorter tags, thus effectively eliminating the significant redundancy from all tag operations. In the general case of loop code that spans non-contiguous memory regions, the minimal number of least significant tag bits for cache conflict identification needs to be found. As we showed in the previous subsection, this number lies in the integer region $[\lceil \log_2 n \rceil, \lceil \log_2 m \rceil]$. A straightforward algorithm for finding this number starts from $\lceil \log_2 n \rceil$ tag bits and keeps adding a tag bit, until all the tags in the code 0-tag regions can be distinctly identified by the values formed by the current number of least significant tag bits.

Furthermore, certain compile/link time optimization techniques can be applied, in order to layout the loop code in such a way that the number of required tag bits for I-cache conflict identification is minimized. Placing the loop code as close as possible is a basic technique for code positioning that aims to minimize the I-cache miss rate [14]. Consequently, while our methodology does not directly modify the code layout, it can significantly benefit from code positioning techniques for I-cache miss rate minimization.

4. APPLYING I-CACHE TAG REDUCTION

4.1 Compile-time support

The methodology starts with application profiling in order to identify the major application loops. After compile and link, the loop code layout including the functions that are called within the loop is fully specified. At this stage the analysis that we present in the previous section for identifying the minimal number of least significant tag bits is performed. The next step is to insert special control code just before entering the loop and right after the exit. The purpose of this code is to set up the hardware support, which is described in the next section, so that only the identified number of least significant tag bits are utilized in the I-cache microarchitecture. This special code includes an instruction that writes into a special *Tag Register* (TR) associated to the hardware support for the tag optimized cache controller. Depending on the processor architecture in which this cache organization is implemented, the special register can be either memory mapped or accessed as an I/O. The particularities of this access and the type of instruction needed to access such a hardware register are of no importance to the proposed methodology and are completely processor dependent. Noteworthy is that the *tag analysis* step must consider the small number of control instructions that will be inserted subsequently just preceding the major application loops so as to determine the 0-tag region spans of these loops.

4.2 I-cache operation

The only requirement for ensuring correct I-cache operation in this scheme is the detection of whether a particular cache line is associated with a full tag or a reduced tag. This is essential as cache lines from previously executed application loops can contain tag bits from these loops, which are deficient in enabling correct conflict identification for the current loop. Consequently, when entering a loop operating with shorter tags, care needs to be taken to ensure the invalidation of the cache lines left from a previously executed loop utilizing reduced tags. Any application code brought into the I-cache while performing full tag operations can still be utilized when in reduced tag mode, of course, by simply reading the corresponding least significant tag bits and disabling the rest.

This functionality can be easily achieved by assigning a special status bit called RT (reduced tag) bit, indicating whether the cache line is associated with a reduced tag. When in normal I-cache operational mode, this bit will always be reset, thus indicating the utilization of complete tags. After entering a loop in reduced tag mode, if the RT bit associated to the cache line being accessed is zero, then a complete tag operation is performed and the RT bit is set to one. In subsequent accesses to this cache line only reduced tag operations will be performed. Prior to entering the particular loop all cache lines still having the RT bit asserted need to be invalidated so that possible conflicts across application loops are correctly resolved. When in normal operational mode, if a cache line has its RT bit set to one, a miss is forced and a full tag is stored while the RT bit is reset. The insignificant penalty of this forced invalidation is the only performance penalty introduced by our approch. A quantitative evaluation of this effect is presented in section 6.

4.3 OS interaction and interrupts

If a context switch or interrupt occurs, there are several ways of dealing with such a situation. If the cache architecture supports process identifiers (pid) as part of the whole tag in order to avoid invalidating the cache, then the same pid's can be used with the minimized tags proposed in this paper. Of course, if the task being scheduled utilizes the same tag minimization technique, then the TR register needs to be preserved as part of the process state.

If an interrupt occurs, then a possible approach consists of locking the cache for the interrupt routine code. An alternative solution is to treat the interrupt routine as a code executing in normal tag mode (i.e., bringing full tags into the cache and resetting the RT bits), and then safely return to the loop execution in reduced tag mode. The utilization of the RT status bits completely avoids any consistency issues that otherwise could have been introduced. Both approaches can be easily implemented and utilized depending on the type and complexity of the particular interrupt being handled.

5. IMPLEMENTATION

The proposed methodology necessitates a hardware support that would be able to dynamically enable only the minimum required bits from the tag array for the program loops. The hardware requires information as to how many tag bits exactly are needed for a particular application loop.

We present an efficient hardware for manipulating the tag memory array so that only the required minimal number of tag bits are used per application loop. The tag array in the cache subsystem is typically implemented as an SRAM array (a similar architecture can be utilized for the CAM based tag arrays in high-associativity caches), possibly divided into multiple banks. The SRAM data array contains *wordlines* for each tag data and a *bitline* for each bit within the tag word. Figure 3 shows the organization of a typical tag SRAM array.

A read operation from the SRAM array is performed in the following way. The address decoder selects the wordline to be read from the array. All the bitlines are precharged and if the selected memory cell by the wordline contains logic zero, then the bitlines start to get discharged. Since the discharge is quite a slow process, there is a sense amplifier at the end of each bitline. If a small drop in the voltage level is detected, a logic zero is registered. The precharge and discharge of bitlines are the most energy consuming operations in SRAM data arrays [15].

By eliminating most of the bitline precharge and discharge operations, our approach greatly reduces the energy dissipation in the tag SRAM array. This is achieved by gating the bitlines according to the minimal number of tag bits required to check for I-cache conflicts. Only the needed bitlines, if any, are precharged and discharged, thus effectively eliminating the redundant reads. The sense amplifiers for the disabled bitlines are gated as well. Furthermore, the tag comparator cells are gated in order to perform the comparison only on the required tag bits.

The number of tag bits for each loop needs to be determined before entering the loop, so that the appropriate number of bitlines

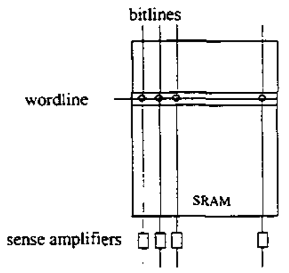

Figure 3: SRAM tag array

	ADPCM	G.721	GSM	EPIC	JPEG	MPEG	PEGWIT
#Hits	6.62	275.16	233.77	52.78	159.27	1,133.75	32.36
#Misses	578	781	80348	3290	16,700	61,713	3,720
MR	0.0087	0.0003	0.0344	0.0062	0.0105	0.0054	0.0115
Loops	1(0)	1(0)	1(0)	1(2)	2(2,2)	3(2,1,1)	1(2)
MR-flush	0.0087	0.0003	0.0344	0.0344	0.0220	0.0058	0.0115
E (mJ)	6.59	273.84	232.73	52.53	158.52	1,128.37	32.21
E opt. (mJ)	5.22	217.16	184.56	45.44	139.64	987.84	27.86
Reduction (%)	20.70	20.70	20.70	13.50	11.91	12.45	13.50

Figure 4: Execution and energy statistics for 32K DM I-cache

are enabled. Since this number is fixed for the loop, it can be stored in a special control register, the *TR* register defined in the previous sections, before entering the loop. Each bit in the TR directly corresponds to an enable signal of bitline and sense amplifiers. The default value of this register specifies that all tag bitlines are enabled. The actual value of this register is used to determine the number of bitlines to enable. The only delay imposed by this implementation consists of the insignificant delay of the gating logic, roughly corresponding to the delay of a simple *and* gate.

The proposed implementation is highly cost efficient, while inherently reprogrammable. It does not impose a timing constraint to the I-cache organization and can be reprogrammed in a post-manufacturing fashion.

Setting the value for enabling the tag bitlines, and invalidating the cache are the operations that are performed in software. Noteworthy is that all of them are performed outside the loop, thus obviating the need for any additional instruction (or equivalently performance degradation) inside the loop.

6. EXPERIMENTAL RESULTS

In our experimental studies, we have used the media benchmark collection [16]. Initially, the benchmarks were profiled and the major application loops identified. This step was performed by utilizing the gnu compiler *gcc* and profiler *gprof* on a Linux workstation. As a next step, the memory layout of the loops and all the functions invoked within them was inspected by analysing the memory map report of the gnu linker. Here the compilation was performed for the SimpleScalar [17] toolset, since the cache statistics were obtained by performing architectural simulations utilizing this simulator. The baseline I-cache characteristics and the I-cache characteristics for the major loops, including the cache invalidation prior to these loops, were generated using the cache simulator from the SimpleScalar toolset. The power models for the utilized cache configurations were obtained by using the Cacti tool [18] for 0.18u technology process. The total I-cache energy dissipation was computed by using the execution statistics from SimpleScalar and the static power model produced by Cacti.

As a baseline cache architecture, we have assumed a phased cache or equivalently (in terms of power), a way-predicting set-associative cache with perfect prediction accuracy. As described in [11], the typical way prediction accuracy for I-caches is higher than 96% and this cache organization avoids the performance disadvantages of the phased cache. These cache organizations are frequently utilized as low-power cache designs. The optimal cache configurations in terms of data and tag bitline and wordline segmentation of 32K direct-mapped, two, and four way associative caches were generated by Cacti together with their power characteristics. The I-cache statistics for the benchmarks and the Cacti power numbers were utilized to compute the baseline and optimized energy consumption.

Tables 4, 5, and 6 present the experimental results for 32K direct-mapped, two, and four way associative I-caches, respectively. The first row gives the number of I-cache hits in millions. The second row of the tables corresponds to the number of I-cache misses, while the third row shows the miss rate percentage. The number of major application loops and the number of least significant tag bits utilized by our methodology is shown in row four of the tables. The number in the brackets shows the utilized number of tag bits for the corresponding application loop. The next row in the tables presents the miss rate after the effect of invalidating the entire I-cache before starting and after leaving each application loop. As was discussed in section 4.2, complete cache invalidation is not needed as long as the RT status bits are utilized and only the cache lines that can potentially cause a consistency issue are invalidated on demand. Nonetheless, we have experimented with the most conservative approach of fully invalidating the cache; we show that even this most pessimistic option results in practically non-existent miss rate increases. For *adpcm*, *g.721*, *gsm*, *epic*, and *pegwit* where the entire application is comprised of a single loop, the miss rate remains unaffected as can be seen in the fifth row in the tables. For the remaining benchmarks, which contained more than one major loop, one can observe that the increase in miss rate is less than 0.1%! This insignificant increase is all the more remarkable since the selected loops account for more than 95% of the execution cycles.

A baseline cache working with full tags has been accounted for within the total power numbers for the part of the code remaining outside the major loops. Finally, the last three rows present the I-cache energy dissipation for the baseline, the tag-optimized cache architectures and the achieved percentage energy reduction. It is noteworthy that the energy data for the baseline 2-SA is slightly higher than the DM organization, while the 4-SA is slightly less. This can be easily explained by the fact that our base cache design is a phased (or way-predicting) cache, in which a single data array is being read. Therefore, the data energy decreases while the tag energy contribution increases as cache associativity is increased. Consequently, since the energy spent in the data array is higher, the data energy decrease in the 4-SA cache is sufficient to reduce the total energy below that of a direct-mapped architecture.

As the result tables show, the energy reductions are directly correlated with increased cache associativity. This effect is to be expected, since in our low-power base cache architecture, the tag power contribution increases with the cache associativity. The only exception is the *gsm* for direct-mapped and two-way set-associative cache, easily explained since in the case of direct-mapped cache, the application lies within a single 0-tag region, thus requiring no tag bits for cache conflict identification. In this case, we turn off completely the cache tag logic. In the case of a two-way set associative cache, one tag bit is required, necessitating an active tag decoder, and thus reducing the total power improvement slightly compared to the direct-mapped case.

By considering the definition of 0-tag regions, it can be trivially observed that increased set associativity leads to a higher number of 0-tag regions spanned by the loop code. At the same time, in-

	ADPCM	G.721	GSM	EPIC	JPEG	MPEG	PEGWIT
#Hits	6.62	275.16	233.82	52.78	159.28	1,133.80	32.21
#Misses	573	770	26,968	2,377	3,618	13,609	2,252
MR (%)	0.0087	0.0003	0.0115	0.0045	0.0023	0.0012	0.0070
Loops	1(0)	1(0)	1(1)	1(3)	2(3,3)	3(3,2,2)	1(3)
MR-flush (%)	0.0087	0.0003	0.0115	0.0045	0.0139	0.0017	0.0070
E (mJ)	6.80	282.98	240.49	54.28	163.81	1,166.01	33.13
E opt. (mJ)	5.19	215.65	191.31	44.49	137.73	971.95	27.15
Reduction (%)	23.79	23.79	20.45	18.05	15.92	16.64	18.05

Figure 5: Execution and energy statistics for 32K 2-SA I-cache

	ADPCM	G.721	GSM	EPIC	JPEG	MPEG	PEGWIT
#Hits	6.62	275.16	233.85	52.78	159.28	1,133.80	32.08
#Misses	573	768	3,816	2,175	2,804	9,129	2,189
MR (%)	0.0087	0.0003	0.0016	0.0041	0.0018	0.0008	0.0068
Loops (tags)	1(0)	1(0)	1(2)	1(4)	2(3,4)	3(4,3,2)	1(4)
MR-flush (%)	0.0087	0.0003	0.0016	0.0041	0.0137	0.0014	0.0068
E (mJ)	6.45	268.31	228.02	51.47	155.32	1,105.58	31.28
E opt. (mJ)	4.75	197.67	176.56	41.23	127.62	900.86	25.06
Reduction (%)	26.33	26.33	22.57	19.91	17.84	18.52	19.91

Figure 6: Execution and energy statistics for 32K 4-SA I-cache

creased associativity implies longer tags and higher energy contribution of the tag arrays. Consequently, even though more tag bits need to be utilized, the net power savings are larger.

7. CONCLUSION

In this paper we have presented a programmable customization methodology for power reduction in the I-cache of high-performance embedded processors. The proposed framework consists of compile/link time support identifying the minimal number of tag bits for complete I-cache conflict resolution for the major application loops. The methodology transfers this application information to the I-cache microarchitecture by software, and dynamically utilizes it to further eliminate the redundancy in the tag operations. An efficient programmable implementation was proposed for supporting the suggested power optimization technique. It preserves the fundamental advantage of processor-based implementations of flexibility, design reuse, and high-volume productions.

Power consumption is a crucial quality factor in numerous modern applications. The experimental results demonstrate the strength of the proposed approach on a set of real-life applications and prove the viability of the power minimization technique for a large range of important applications.

8. REFERENCES

[1] K. Ghose and M. B. Kamble, "Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation", in *ISLPED*, pp. 70–75, August 1999.

[2] M. B. Kamble and K. Ghose, "Analytical energy dissipation models for low-power caches", in *ISLPED*, pp. 143–148, August 1997.

[3] N. Bellas, I. Hajj and C. Polychronopoulos, "Using dynamic cache management techniques to reduce energy in a high-performance processor", in *ISLPED*, pp. 64–69, August 1999.

[4] A. Ma, M. Zhang and K. Asanovic, "Way memoization to reduce fetch energy in instruction caches", in *Workshop on Complexity-Effective Design, 28th ISCA*, June 2001.

[5] E. Witchel and K. Asanovic, "The span cache: software controlled tag checks and cache line size", in *Workshop on Complexity-Effective Design, 28th ISCA*, June 2001.

[6] W. H. Wolf, "Hardware-Software Co-Design of Embedded Systems", *Proceedings of the IEEE*, vol. 82, n. 7, pp. 967–989, July 1992.

[7] P. Petrov and A. Orailoglu, "Performance and power effectiveness in embedded processors - Customizable Partitioned Caches", *IEEE TCAD*, vol. 20, n. 11, pp. 1309–1318, November 2001.

[8] R. Panwar and D. Rennels, "Reducing the frequency of tag compares for low power I-cache designs", in *SLPE*, pp. 57–62, October 1995.

[9] J. Kin, M. Gupta and W. H. Mangione-Smith, "The filter cache: an energy efficient memory structure", in *30th MICRO*, pp. 184–193, April 2001.

[10] A. Hasegawa et al, "Sh3: high code density, low power", in *IEEE Micro*, pp. 11–19, 1995.

[11] K. Inoue, T. Ishihara and K. Murakami, "Way-predicting set-associative cache for high-performance and low energy consumption", in *ISLPED*, pp. 273–275, August 1999.

[12] J. Montanaro et al., "A 160Mhz, 32b 0.5W CMOS RISC Microprocessor", in *IEEE ISCC*, pp. 214–229, February 1996.

[13] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA, 1996.

[14] K. Pettis and R. C. Hansen, "Profile guided code positioning", in *SIGPLAN*, pp. 16–27, June 1990.

[15] N. Bellas, I. Hajj and C. Polychronopoulos, "A detailed, transistor-level energy model for SRAM-based caches", in *ISCAS*, pp. 198–201, June 1999.

[16] C. Lee, M. Potkonjak and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", in *30th MICRO*, pp. 330–335, December 1997.

[17] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", Technical Report 1342, University of Wisconsin-Madison, CS Department, June 1997.

[18] G. Reinman and N. Jouppi, "An Integrated Cache Timing and Power Model", Technical report, Western Research Lab, 1999.