Energy-Efficient Physically Tagged Caches for Embedded Processors with Virtual Memory [*]

Peter Petrov
University of Maryland at College Park
ECE Department
ppetrov@ece.umd.edu

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

ABSTRACT

In this paper we present a low-power tag organization for physically tagged caches in embedded processors with virtual memory support. An exceedingly small subset of tag bits is identified for each application hot-spot so that only these tag bits are used for cache access with no perfromance sacrifice as they provide complete address resolution. The minimal subset of physical tag bits, i.e. the compressed tag, is dynamically *updated following the changes in the physical address space of the application. Special support from the operating system (OS) is introduced in order to maintain the compressed tag during program execution. The compressed tag is updated by the OS to match the current set of physical memory pages allocated to the application. We have proposed efficient algorithms that are incorporated within the memory allocator and the dynamic linker in order to achieve dynamic update of the compressed tags in the cases where the mapping between virtual and physical addresses is modified; such cases include memory allocation/deallocation and swapping physical pages on the secondary memory storage. The only hardware support needed within the I/D-caches is the support for disabling bitlines of the tag arrays. An extensive set of experimental results demonstrates the efficacy of the proposed approach.*

Categories and Subject Descriptors

B.3 [**Hardware**]: Memory structures; C.1 [**Computer Systems Organization**]: Processor Architectures; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems

General Terms

Algorithms, Design, Experimentation, Performance

1. INTRODUCTION

Power consumption has been recognized as one of the most important quality characteristics for a large number of ubiquitous digital systems. For a multitude of hand-held and wireless products, such as laptop computers, personal organizers, and cellular phones, to name just a few, power consumption has a direct impact on battery life, a quality parameter of utmost importance.

The memory subsystem is one of the most important components of any modern processor design, greatly determining its performance and usability. The cache subsystem is an important microarchitectural component serving to bridge the ever growing gap between memory access time and processor execution speed. Both tag and data arrays account for a significant part of the transistor budget and hence the total power [1].

The tag arrays are used to store a certain number of the most significant bits from the effective address in order to resolve data cache conflicts. Typically, the tag field from the effective address constitutes more than half of the entire address. The particular tag length depends on the cache organization (associativity, number of cache lines, and cache line size). Conceptually, the tags behave as keys associated to certain memory regions and are used to distinguish each of these memory regions in the cache. Given the general assumption that the application can access arbitrary memory regions, the whole tag field from the effective address needs to be stored in the tag array and used for cache conflict identification.

Practically all of the modern high-performance processors support some type of paged virtual memory controlled by an OS in order to facilitate efficient code and data relocation in support of processor sharing by multiple processes. Furthermore, virtual memory support enables the implementation of memory protection schemes, which greatly increases system reliability. The processor generated addresses, referred to as *virtual addresses*, are translated to real *physical addresses* before accessing the system memory. The *Translation Lookaside Buffer (TLB)* is a cache-like structure responsible for the dynamic translation of virtual addresses generated by the processor to physical addresses used to access the memory. The mapping between virtual and physical addresses is typically maintained by the OS and established by the OS loader, dynamic linker and memory manager. TLB misses result in trapping into the OS where the missed translation is retrieved from page tables maintained by the kernel.

In order to avoid cache consistency issues, it is a common practice to employ physically tagged L1 data and instruction caches. In the case of physically tagged caches no cache flushing is needed during a context switch. At the same time physically tagged caches preserve data consistency by eliminating the possibility of cache synonyms, a situation where a shared data or code maps to distinct cache locations for each process, thus introducing consistency hazards and cache capacity underutilization. However, accessing physically tagged caches requires a TLB lookup to obtain the physical tag. A single virtual address can be mapped to various physical addresses during program execution depending on the current physical location of the memory page. The physical location can vary due to memory page replacement. Consequently, the physical tags are unknown prior to executing the program but become available only after loading and during program execution.

In this paper we introduce a methodology that dynamically compresses the effective tag length by exploiting application and system information which is updated by the OS while executing the

[*]This work is supported by NSF Grant 0082325.

program. A small subset of tag bits is identified to have a complete resolution for the application memory set and only these tag bits are read from the tag arrays and compared against when accessing the cache. All the previous approaches in tag compression for low-power focus on statically allocated working sets, where the data/code memory layout is statically known after compiling/linking the application. These approaches are applicable only for virtually tagged caches where the code or data layouts are statically known. In this paper we focus on systems with virtual memory where the caches are physically tagged. For such systems, the physical address ranges of the code and data are not statically known and are only available after loading the program and could change during executing. The dynamic application information regarding the physical location of both data and instruction memory pages is captured and exploited through the co-operation of reprogrammable hardware support introduced within the TLB and the cache tag arrays, together with efficient OS support to dynamically update the effective tag size in situations where physical pages have been added or removed from the application memory map.

Consequently, only an exceedingly small subset of tag bits is read from the tag arrays and compared to the same subset of tag bits from the address referenced by the processor. This subset of tag bits is being either extended or reduced by the OS memory manager in the, rather infrequent for embedded systems, events of dynamic memory allocation and deallocation. The hardware support is programmable by software thus allowing the OS to dynamically update the active tag size to be used when accessing the instruction and data caches. By disabling the large number of unneeded tag bits, significant power reductions in the caches are achieved with practically no performance sacrifice.

The instruction and data on-chip caches have been long known as one of the major culprits for the high power consumption of modern high-end and embedded processors. They amount to 25% of the total power for the Alpha 21164 [2] and 43% for the StrongARM-110 [1]. This is mainly due to the fact that the data and tag arrays are usually implemented as SRAM in order to allow for fast clocks.

In set-associative cache designs, a significant amount of power is spent in accessing simultaneously all the cache ways. A *phased* cache [3] has been proposed to alleviate the resulting power problem by accessing the tag arrays initially. Therefore, the phased cache organization reduces power consumption, while paying the price of an additional cycle in cache access time.

The previous work in low-power tag architectures [4] focuses on virtually tagged caches and statically allocated data/code, where the address ranges used to access the cache are statically known after linking the application. In that work, the authors introduce static analysis algorithms that are incorporated within the compiler/linker in order to find the optimal reduced virtual tags for the given static data set. In this paper we attack the considerably more challenging problem of physically tagged caches and dynamic working sets.

2. VIRTUAL MEMORY AND CACHES

The presence of virtual memory introduces several options for accessing L1 caches. As data and code reside in the actual physical memory, the use of physical addresses to access the cache seems to be the solution that would work naturally. Nonetheless, obtaining the actual physical address from the virtual address generated by the processor requires a page translation process. It is the responsibility of the OS to allocate and map physical pages to virtual pages and to maintain this correspondence. The TLB is a hardware cache buffer storing the most recent virtual to physical translations. However, performing a TLB lookup prior to accessing the cache would impose a significant delay to the cache access time.

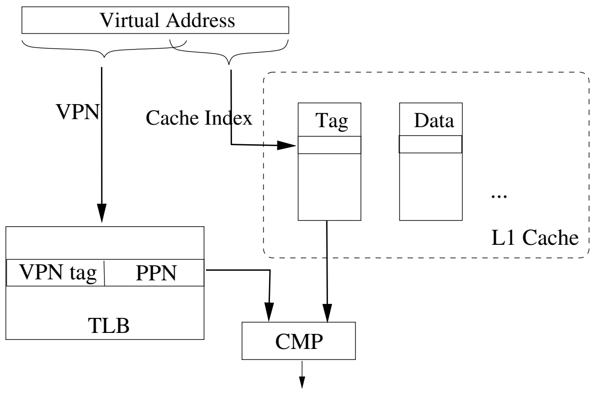

Figure 1: Virtually-indexed and physically-tagged cache

Alternatively, using the virtual address only to access the cache requires no overhead for virtual to physical translation. Unfortunately, virtual caches introduce the *synonym problem* [5, 6]. This problem appears when code or data is shared by multiple processes. For such cases, it is possible that a shared physical page is mapped to distinct virtual pages for each process. In this way, the same data would be placed in multiple cache locations resulting in data consistency problems as a process can modify the shared data in one cache line while the remaining data copies remain with their previous value.

In order to avoid increasing the cache access time and to facilitate the solution of the synonym problem, caches are typically indexed with the virtual address and tagged with the physical address. Such a cache access mechanism allows for the TLB lookup to be overlapped with the cache index operation. Figure 1 shows the architectural organization for virtually-indexed and physically-tagged caches. The *cache index* is obtained from the virtual address while the tag is obtained from the translated *Physical Page Number (PPN)*. If the cache index and the VPN do not overlap, no cache synonyms are possible. In the case of overlapping VPN and cache index, two basic approaches can be considered. One possibility, adopted in the Alpha 21264 processor [7], is to access all possible cache lines corresponding to the overlapped bits. This solution requires no OS or compiler involvement; nonetheless, it elevates significantly cache power consumption. The alternative software-only approach is to ensure that the overlapping VPN and cache index bits remain identical for both virtual and physical addresses. This is achieved with the *page coloring* approach [5] where the OS places physical pages at locations for which the overlapping bits are identical.

The dynamic tag compression methodology that we propose in this paper targets the tag arrays of physically tagged L1 caches. The effective physical tag size is drastically reduced with the active help of the OS and the introduced hardware support.

3. TAG COMPRESSION

Depending on the cache organization, the tag contains a large part of the most significant address bits. For instance, a fully associative cache requires no index and other than the few least significant bits that are used as a cache line index (usually 2 or 3 bits), the entire remaining part of the address constitutes the tag part.

As physical addresses are ultimately used in accessing data or code from the main memory, the fundamental purpose of the physical tags is to distinguish the memory regions that have distinct tags. We refer to such memory regions with identical tags as 0-tag regions. Consequently, any set of unique identifiers assigned to each such 0-tag region can be used instead of actual tags in identifying whether there is a cache hit or miss.

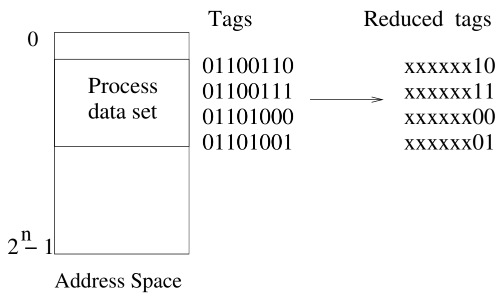

Figure 2: Tag resolution

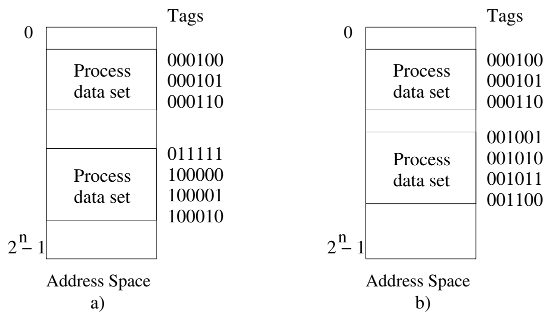

Figure 3: Minimal tags

3.1 Distinguishing the tags

Depending on the number of associativity sets, the cache is indexed with a few least significant bits from the address. The remaining part of the address is used as a tag when performing a lookup within the selected associativity set. Using all of the remaining most significant bits of the address as tags is necessary only under the assumption that the process code and data occupy the entire physical address space. This assumption is far from true in practically any system.

Figure 2 illustrates an example process with the data set occupying a set of contiguous 0-tag regions, exhibiting 4 distinct tags. One can immediately observe that the tags, shown on the right side of the memory map, can be completely distinguished by their two least significant bits only. The remaining most significant bits of these tags are not essential in providing unique 0-tag region identification. Consequently, if the process is guaranteed to access this data set only, only the two least significant tag bits need be utilized as effective tags instead. As these two bits provide complete tag resolution, the normal cache functionality is preserved and no deviation from the usual hit/miss behavior is introduced.

While using only the two least significant tag bits to perform tag lookup would have no performance implications, the net effect on the cache power consumption is quite significant. Instead of reading the entire tag from the tag arrays and comparing it to the tag generated by the processor, only the two least significant bits from the tag arrays are to be read and compared to the two least significant bits of the processor provided tag. The power associated to the remaining most significant bitlines, the corresponding sense amplifiers, and the comparator cells is drastically reduced.

In order to maintain the consistency of the aforementioned power optimization technique, for each process the cache architecture needs to be aware of the number of the tag bits to be utilized as an effective tag. This can be easily accommodated by a software-controlled register, which contains a bitmask value specifying the least significant tag bits to be used as tags. The value of this register is stored as part of the process state during a context switch.

3.2 Minimal tags

Figure 2 has presented an example working set consisting of consecutive virtual pages with four distinct tags. Complete tag resolution can be accomplished through the utilization of the two least significant bits. This effect is achieved since the four tags differ in these two bit positions.

In the context of uniquely identifying the set of tags, the remaining most significant bits, even though not identical for all the tags, offer no further assistance, as the two least significant bits are perfectly capable of full resolution. One can notice that the two least significant bits constitute an optimal solution in terms of the minimality of the number of bits necessary to achieve complete tag resolution as two bits of information is the information theoretic minimum for uniquely encoding a set of four elements. It is straightforward to observe that for any set of n *consecutive tags*, the $\lceil \log_2 n \rceil$ least significant bits always provide complete resolution for that set of tags as this group of bits would have a different value for the distinct tags. No conclusion can or need be drawn regarding the most significant bits which can be identical or distinct. As the $\lceil \log_2 n \rceil$ least significant tag bits uniquely identify the set of adjacent tags, the particular values of the remaining bits are of no importance in the process of accessing the cache.

Figure 3a shows an example of a working set that consists of seven tags, shown on the right side. The salient difference in this case is that not all the virtual pages are adjacent but that they are rather separated into two groups, each containing consecutive pages. Even though the address space occupied by these particular seven tags contains 30 distinct tags, it is evident that only the three least significant tag bits are necessary to provide complete resolution for the working set tags.

Figure 3b illustrates a similar situation of a working set containing seven tags. However, in this case the virtual pages of the working set occupy distinct addresses. Even though there are only seven distinct tags, it is not possible to utilize only the three least significant bits as effective tags. This is due to the fact that the first and the last tags exhibit exactly the same three least significant bits. To completely resolve this set of seven tags, four least significant bits have to be used instead.

In general, for a given set of n distinct tags, the minimal number k of least significant bits sufficient to provide complete tag resolution depends on the size of the address *range* occupied by the working set and the particular addresses within that space in which the data set resides. It is evident that if the occupied address space comprises of m distinct tags, the minimal required number k of least significant tag bits with complete resolution can be in the integer range of $[\lceil \log_2 n \rceil, \lceil \log_2 m \rceil]$. The case of $k = \lceil \log_2 m \rceil$ corresponds to the worst situation, where the entire address space spanned by the data set needs to be distinguished in order to resolve the set of tags.

3.3 Functional overview

It is a well known rule-of-thumb that programs spend 90% of their execution time in 10% of their code. Usually most of the program execution time is spent in a number of tight loops or functions, generally referred to as "hot-spots". For instance, a typical multimedia application spends most of its execution time within a set of heavily utilized DSP kernels. Each program hot-spot would access only a limited number of 0-tag regions, thus providing ample opportunities for applying the physical tag power optimization as outlined in the previous subsections. Consequently, the proposed approach is applied on the application hot-spots in order to achieve significant physical tag reductions, while covering a large fraction of the program execution.

To apply the proposed technique, application information regarding its data/instruction physical working set, and more specifically the minimal number of least significant bits of the physical tags required for complete tag resolution, needs to be provided to the cache. Physical addresses are not known prior to entering the hot-spot as pages might have been replaced while executing another hot-spot. Consequently, it is the responsibility of the OS to make a decision regarding the minimal number of physical tag bits to be used for cache access. This is achieved through the co-operation between the TLB and the OS. Prior to entering a hot spot, a special bit associated with each TLB entry is reset. This bit denotes whether the page associated to that particular TLB entry has been considered for identifying the compressed physical tag. The first time a page is accessed, this bit would be reset and a trap to the OS would be effected. At this point the OS would dynamically update the number of least significant bits from the physical tags to be used as actual tags and subsequently set the bit to indicate that the page has been accommodated by the current compressed physical tags. This mechanism would enable the utilization of minimal physical tags for each instance of the hot-spot depending on the current memory requirements of the application.

Initially, the program's hot-spots are identified by profiling while the working set for all these hot-spots is identified after compiling and linking the program. At this stage a special setup code is inserted prior to entering the hot-spot that includes an OS call informing the OS that a hot-spot is about to be entered.

As mentioned earlier, there are a number of circumstances that result in an absence of information regarding the actual physical tags prior to entering the hot-spots. Such situations include swapping pages while executing another part of the program, page thrashing while executing the hot-spot, dynamically allocating/deallocating data, and dynamically linked libraries. Therefore, for such cases, the OS would be responsible for adjusting the size of the minimal physical tags when changes in the instruction and data working sets occur. In order to accomplish this process, an efficient algorithm for updating the minimal size of the physical tags is needed. This algorithm would be integrated within the OS and executed whenever a new page is placed in the system memory.

We proceed by giving a more detailed explanation of the required OS and compiler involvement; subsequently, we outline the algorithmic support required by the compiler and the OS. On a cache lookup, only the least significant physical tag bits are read from the tag arrays and compared against the physical tag of the address being looked up. Note that when a new cache line is allocated, the entire physical tag is stored into the tag array. This allows for the size of the reduced physical tags to be changed dynamically.

3.4 Compiler and OS support

The proposed technique can be divided into two major phases. The first phase occurs while building the program executable code. During this phase, the application characteristics regarding its hot-spot regions are identified. A special setup code is inserted prior to entering the hot-spot. This setup code contains a system call that registers within the OS that a hot-spot is to be executed. Thereupon, the OS marks all the pages in the TLB as not being as of yet referred to by the application hot-spot. This is achieved by resetting the extra status bit, the *Referred (R)* bit, that we have introduced to each TLB entry. This special bit serves as an identification as to whether the physical page has been accessed already by the hot-spot and thus has been considered in determining the actual size of the compressed physical tags.

As the code or data physical addresses accessed by each hot-spot can possibly change for the different instances of the hot-spot, the number of reduced physical tag bits needs to be updated in such cases. This is achieved in the second phase of the proposed methodology during the execution of the program. This task of dynamically compressing the tags as new physical pages are being allocated to the application is handled by the OS. There are two cases when this support is invoked. The first case is when a physical page is being accessed for the first time within the hot-spot. This is indicated by the *R* status bit in the TLB entry being reset. Such a situation would trigger a trap into the OS, and the physical tags associated to the address range of the physical page will be considered for recomputing the actual tag size of the compressed tags. Note that this can also lead to having the compressed tag size unmodified in the cases where the currently identified least significant bits still provide a full tag resolution. The second case is when the OS is placing or removing a page into the physical memory. This can happen in the case of a page fault or when a new data is being allocated. In this case, the memory manager would invoke the routines for updating the compressed tag size after performing the page placement and/or page removal from the physical memory.

3.5 Algorithmic overview

The problem of identifying the minimal tag bits for complete tag resolution can be formally specified in the following way. The working set of the process, which consists of t groups of adjacent 0-tag regions, can be specified as a set of integer pairs (S_i, L_i), for all $1 \le i \le t$. Each pair represents a group of adjacent virtual pages, with the first integer S_i corresponding to the first tag of the group and L_i denoting the number of distinct tags for that group. Within this formal specification of the problem, finding the minimal number k of least significant tag bits with complete resolution maps to the problem of finding the minimal k such that the integer intervals $[S_i \, mod \, 2^k, S_i \, mod \, 2^k + L_i]$, for all $1 \le i \le t$ are non-overlapping. The value of k is exactly the number of least significant tag bits that provide complete tag resolution.

The algorithm starts by assigning $k = \lceil \log_2 n \rceil$, where n is the total number of distinct tags, and performs a check for the aforementioned non-overlapping property. The overlap check can be implemented quite efficiently since the modulo operations can be implemented as simply masking all but the k least significant bits and performing the comparisons. If the set of intervals is overlapping, the algorithm increments k and performs the same overlap check with the new value of k. The timing complexity of the overlap check operations is $O(t * log(t))$, where t is the number of groups with adjacent tags. The efficiency in terms of computational order, coupled with the practical restriction that the number of groups typically encountered does not exceed 10, results in an algorithm whose online performance makes it quite suitable for dynamic tag compression as it in no way impacts overall performance.

As the instruction or data working set can change during program run-time, OS support is needed in order to update the current minimal k with respect to the newly added or removed group of virtual pages. In this case, an algorithm is required, which when given a data or instruction working set and its corresponding reduced tags identifies whether the size of these tags needs to be increased.

Formally, this problem can be described in the following way. Given a set of integer pairs (S_i, L_i), $1 \le i \le t$, an integer k, such that the set of integer intervals $[S_i \, mod \, 2^k, S_i \, mod \, 2^k + L_i]$, $1 \le i \le t$ is non-overlapping, and an additional integer pair (S_{t+1}, L_{t+1}), determine the amount by which the value of k should be increased, if at all, so that the new set of integer intervals $[S_i \, mod \, 2^k, S_i \, mod \, 2^k + L_i]$, $1 \le i \le (t+1)$ is non-overlapping. The first step of the algorithm is to find out whether the current value of k still makes the new set of integer intervals non-overlapping and if not, what the previous intervals that overlap with the newly introduced one are. This step can be performed in linear time, $O(t)$ in the number of intervals, by simply traversing them and checking for overlap by extracting the k least significant bits for each interval and performing the two comparisons. If no previous interval that overlaps with the new one exists, then the current value of k provides a complete tag resolution. If there exists a subset of intervals that overlap with the new interval, the algorithm needs to consider only them in trying to find by how much to increase the value of k. This value is being incremented iteratively until there is no longer an overlap between the affected integer intervals. Correctness is preserved through consideration of the initially overlapped intervals only since it is impossible to introduce a new overlap when increasing k if there was no overlap in the first place. This algorithm is integrated within the dynamic linker and the memory manager and executed when a new portion of memory is mapped to the address space of the program.

A somewhat converse algorithm is required in the case of data deallocation by the memory manager. Of course, it is possible simply to ignore this case, as deallocating data and removing pages can only decrease the set of tags associated to the program hot-spots and thus introduces no conflicts with the currently selected minimal tag. Nonetheless, it is possible that in the face of a diminishing set of tags, the value of k can be decremented while still achieving full tag resolution. In this case, the algorithm has to proceed with performing the overlap check procedure for the new set of intervals after removing the interval corresponding to the deallocated data. The value of k in the process is repeatedly decremented until a conflict occurs. In this way the minimal number of k least significant tag bits is identified and updated for all the affected hot-spots.

4. HARDWARE SUPPORT

Since the tag power optimization technique we propose effectively compresses and eliminates most of the tag bits, the only required hardware support from the cache is to have tag arrays with bit positions that can be disabled/gated. In this way only the tag bits selected to be used as an effective tag would be read from the tag arrays, while the remaining bit positions will remain inactive and disabled. As the number of least significant tag bits depends on the program and even on the particular hot-spot, we require that the hardware support allows software control upon which particular tag bit positions can be disabled.

A typical implementation for cache structures employs tag and data arrays implemented as SRAM arrays. It is possible that these SRAM arrays are split into multiple banks in order to optimize the access time. There are also variations as of to how the horizontal and vertical lines are organized, however, the general principle of operation is similar. An address decoder selects a row within the array commonly referred to as a *wordline*. The wordline selects the memory cells within that array row. When performing a memory access, all the vertical lines corresponding to the bit positions and referred to as *bitlines* are precharged in order to detect the voltage level in the selected row; the cell content is determined according to what voltage level the bitline goes to. Since discharging of long bitlines exhibiting high capacitance can be quite a slow operation, sense amplifiers are normally used at the end of each bitline in order to quickly determine a voltage level change. Charging and discharging the SRAM bitlines coupled with the sense amplifier circuitry for voltage level change detection are the major contributors to the SRAM power consumption [8]. An alternative approach for a highly associative cache structure is the use of a *Content Addressable Memory (CAM)* to implement each associativity set of the tag arrays. Nonetheless, the underlying circuit techniques are similar to the ones used for SRAM-based arrays; hence, we outline the hardware support for SRAM based cache organizations.

Disabling a bit position from the tag directly translates to disabling the precharge of its corresponding bitline and consequently the associated activity of its sense amplifier circuitry. Gating the bitline can be trivially achieved through a simple *AND* circuitry in order to block the activity on the bitline. A special register, called the *Bitline Enable (BE)* register, is introduced as a part of the specialized hardware support. The BE register is software accessible and contains a bitmap value that determines which bitlines are to be disabled. Each bit in the BE register maps to a bitline in the tag arrays. Through the BE register, the OS and the application assert software control over the disabling of bitlines in the tag arrays.

The only modification to the TLB is the introduction of an additional status bit, the *Referred (R)* bit, which is associated to each TLB entry. The R bit, as mentioned in Section 3.3, signifies whether the page has been already accessed by the application hot-spot.

5. EXPERIMENTAL RESULTS

As virtual memory effects are challenging to faithfully simulate through traditional architectural simulation, a specialized framework has been constructed in order to provide a quantitative analysis of the proposed technique. The framework that we have built provides modeling of logically indexed, physically-tagged caches, a finite memory system with paging behavior, and a traditional clock-based page replacement algorithm. The framework was combined with the SimpleScalar [9] simulator in order to execute various multimedia kernels. The programs chosen are part of the Mediabench set [10] of benchmarks, and represent frequently used and important consumer multimedia algorithms, consisting of operations necessary to handle modern compressed sound, image, and video formats. The algorithms presented in this work were applied to test the effective tag compression technology in the context of a real system. The compression efficiency for each kernel and the effect of paging upon the efficiency was observed.

In order to precisely evaluate the achieved power saving, the data cache tag subsystem was implemented and simulated in SPICE, version 3f5. The tag array, including the gating logic circuitry, was designed in TSMC 0.25u CMOS process operating at 2.5V as traditional SRAM blocks of 64, 128, 256 and 512 wordlines, and 18-21 bitlines, corresponding to various cache organizations that were analyzed. Each bit-line is composed of the precharge unit, the SRAM cell and the sense amp. Precharge is implemented with three CMOS type-n transistors, while a traditional six transistor SRAM cell is utilized. Detailed power savings for compressed tags for each kernel can be seen in the subsequent tables.

The baseline cache architecture consists of 512 sets of 8-byte blocks organized in 4 associativity ways. The physical page size is 4K, a typical value for many virtual memory systems. It is important to note that associativity does not affect the energy reduction percentage, the ratio of tag bit comparisons being identical for each associativity level, but power savings increase in absolute value as associativity increases.

The following tests were run with varying amounts of physical memory available to the kernels: first with sufficient memory to prevent paging as shown in Figure 4, then with only ninety percent of the required memory for each kernel, and subsequently with only eighty percent; the latter two cases causing page faults to relocate code and data. These two cases are represented in Figures 5, and 6, respectively. Each table shows the energy reduction in the tag arrays. The tag power is typically a large fraction of the total cache power, for instance, it was reported to be 54% of total cache power in [11]. The tag contribution is even larger for low-power and high-associativity cache organizations.

The amount of paging incurred for each benchmark depends

	I-cache	D-cache
adpcm	94.9%	89.6%
epic	70.0%	58.9%
g721	75.0%	80.0%
gsm	80.6%	89.1%
jpeg	65.8%	60.5%
mp3	77.1%	69.2%
mpeg2	67.0%	40.9%

Figure 4: Power savings with sufficient memory

	I-cache	D-cache	Page Faults
adpcm	94.9%	89.6%	2
epic	70.4%	58.7%	31
g721	75.0%	75.0%	5
gsm	77.0%	84.1%	14
jpeg	65.8%	56.6%	14
mp3	77.2%	69.1%	13
mpeg2	66.9%	40.1%	27

Figure 5: Power savings with 90% memory requirements

	I-cache	D-cache	Page Faults
adpcm	94.9%	84.6%	5
epic	70.2%	62.2%	43
g721	75.0%	75.0%	6
gsm	80.6%	89.1%	16
jpeg	62.9%	56.5%	21
mp3	78.6%	65.9%	23
mpeg2	64.5%	55.4%	352

Figure 6: Power savings at 80% of memory requirements

	I-cache	D-cache
adpcm	99.9%	99.5%
epic	82.4%	72.8%
g721	88.2%	88.2%
gsm	94.2%	93.0%
jpeg	73.1%	71.0%
mp3	85.1%	79.2%
mpeg2	76.4%	63.4%

Figure 7: Power savings with sufficient memory: No Paging

upon how much of the working set of each hot-spot can be contained in the memory space given. The number of page faults are given to correlate with the effect on compression behavior. The gradual reduction in the available physical memory enables one to observe the effect of page swapping on the effectiveness of the dynamic tag compression algorithm. As expected, when there is a small amount of paging, the tag compression algorithm efficiently identifies the minimal number of tag bits with results similar to the case of no page swapping. The small deviations are only due to the differences in page locations when the OS brings in and out pages. In the only case where a significant amount of page swapping occurs, the *mpeg2* benchmark with memory at 80% of its requirements, it can be observed that the dynamic tag compression algorithm achieves better results compared to the base case. This can be easily explained by the fact that in any instance of the program execution much smaller data/code working sets are allocated in the physical memory, while a large number of pages have been swapped out to the secondary storage. Evidently, working with a smaller number of 0-tag regions results in fewer physical tag bits needed for cache access.

Additional tests, shown subsequently in Figure 7, evince that as cache size increases, the effectiveness of our methodology also increases. These tests were performed with a cache configuration of 2048 sets, 4-way associativity, and 16-byte blocks, for a total cache size of 128K, yielding a base tag size of 17 bits for addresses and cache lines. Despite the decrease in tag size and thus a decrease in the potential tag bit comparisons for the case of an uncompressed cache, the cache size increase results in more pages fitting within a single 0-tag region, allowing the allocation of multiple pages before requiring additional tag bits to differentiate each region. It can be seen that frequently no increase in the compressed tag size is suffered in the case of multiple page allocations.

6. CONCLUSIONS

In this paper, we have presented a methodology for low-power physically tagged instruction and data caches. The proposed technique dynamically identifies the minimal number of least significant tag bits that would be sufficient for complete tag resolution. Only the selected few tag bits would be read from the physical tag arrays and used as effective tags, thus resulting in significant power savings. An efficient OS algorithmic support is introduced to dynamically determine the compressed tags in the face of changing physical working set. While the power consumption of the caches is thus drastically reduced, there is no impact on the miss rate and no additional hardware on the critical path of accessing the cache is introduced. The cost-effective and software-controlled hardware support enables the application of the proposed methodology to practically any high-performance embedded processor.

7. REFERENCES

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

[2] J. Edmondson et al., “Internal organization of the Alpha 21164, a 300MHz 64-bit Quad-issue CMOS RISC Microprocessor”, *Digital Technical Journal*, vol. 7, n. 1, pp. 119–135, 1995.

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

[4] P. Petrov and A. Orailoglu, “Power Efficient Embedded Processor IP’s through Application-Specific Tag Compression in Data Caches”, in *DATE*, pp. 1065–1071, March 2002.

[5] M. Cekleov and M. Dubois, “Virtual-address caches. Part 1: problems and solutions in uniprocessors”, *IEEE Micro*, vol. 17, n. 5, pp. 64–71, September 1997.

[6] J. Kim, S. Min, S. Jeon, B. Ahn, D. Jeong and C. Kim, “U-cache: a cost-effective solution to synonym problem”, in *HPCA*, pp. 243–252, January 1995.

[7] R. Kessler, “The Alpha 21264 Microprocessor”, *IEEE Micro*, vol. 19, n. 1, pp. 24–36, March/April 1999.

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

[9] T. Austin, E. Larson and D. Ernst, “SimpleScalar: An infrastructure for computer system modeling”, *IEEE Computer*, vol. 35, n. 2, pp. 59–67, February 2002.

[10] 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.

[11] 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.