# Chapter 4
# Power-efficient Memory and Cache

The memory subsystem plays a dominant role in every type of modern electronic design, starting from general purpose microprocessors to customized application specific systems. Higher complexity in processors, SoCs, and applications executing on such platforms usually results from a combination of two factors: (1) larger amounts of data interacting in complex ways and (2) larger and more complex programs. Both factors have a bearing on an important class of components: memory. This is because both data and instructions need to be stored on the chip. Since every instruction results in instruction memory accesses to fetch it, and may optionally cause the data memory to be accessed, it is obvious that the memory unit must be carefully designed to accommodate and intelligently exploit the memory access patterns arising out of the very frequent accesses to instructions and data. Naturally, memory has a significant impact on most meaningful design metrics [31]:

**Area** Memory related structures dominate the area of most processors and ASICs. In modern processors, the cache memory structures easily account for more than 60% of the chip area.

**Delay** Since large amounts of program and data are accessed from memory, the access delays have an immediate impact on performance metrics such as total execution time and response time.

**Power and Energy** Every instruction being executed one or more memory accesses. Large amounts of data and code lead to larger energy consumption in the memory because of both the memory size and the frequency of accesses.

Power optimizations targeting the memory subsystem have received considerable attention in recent years because of the dominant role played by memory in the overall system power. The more complex the application, the greater the volume of instructions and data involved, and hence, the greater the significance of issues involving power-efficient storage and retrieval of these instructions and data. In this chapter we give a brief overview of how memory architecture and accesses affect system power dissipation, and mechanisms for reducing memory-related power through diverse means: optimizations of the traditional cache

P.R. Panda et al., *Power-efficient System Design*, DOI 10.1007/978-1-4419-6388-8_4, 
© Springer Science+Business Media, LLC 2010

memory system, architectural innovations targeting application-specific designs, compiler optimizations, and other techniques. In addition to caches used in general purpose computer based systems, we also give considerable emphasis in the chapter on power-efficient memory optimizations in ASICs and SoCs found in embedded systems.

## 4.1 Introduction and Memory Structure

As applications get more complex, the memory storage and retrieval plays a critical role in determining power and energy dissipation: larger memories lead to larger static power, and frequent accesses lead to larger dynamic power.

### *4.1.1 Overview*

Figure 4.1 shows the typical external interface of a generic memory module. The interface consists of three components:

**Address** The address bus is an input to the memory and specifies the location of the memory that is being accessed. The width of the address bus depends on the number of memory locations. When the bus width $m$ equals the number of address bits, it can be used to access a maximum of $2^m$ locations. In some memories, especially Dynamic Random Access Memory (DRAM), the address bus is time-multiplexed to carry different parts of the memory address at different times.

**Data** The data bus carries the associated data for the memory operation. The bus is usually bidirectional, allowing data to be either an input or output to the memory, depending on the type of operation. The width of the data bus depends on the typical datapath width used in the design. For example, if the design is

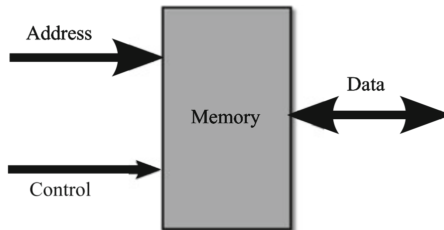



**Fig. 4.1** External interface of a typical memory. The *Address* bus input gives the memory location. The *Data* bus is usually bidirectional – it is an input for a WRITE operation, and an output for a READ operation. The *Control* input consists of information such as type of operation (READ/WRITE) and chip enable

dominated by 32-bit operations, then the memory data bus is likely to be 32 bits wide. DRAM systems may be organized differently: the data bus of an individual DRAM chip usually has a smaller width (say 4 bits), and a 32-bit data bus is composed of data bits provided by a set of eight DRAM chips.

**Control** In addition to the address and data buses, memories usually have some control signals which are used to indicate important information such as whether the memory module is selected in the current clock cycle and an encoding of the operation to be performed. When the memory access protocol is more complex, these signals may carry other information such as clocks and row/column address strobe.

Memory accesses essentially constitute two types of operations: Read and Write.

**Read** The Read operation takes as input an address and returns the data stored at the corresponding memory location. The address is provided through the *address* input port in Fig. 4.1, and the data returned through the *data* port. Control signals are set to the appropriate encoding to indicate this operation.

**Write** The Write operation takes as input an address and the data, and stores the given data at the memory location corresponding the specified address. The address port is used as in the read operation, but the data bus is now an input port to the memory. Again, the control signals are set so as to indicate the write operation.

In addition to the above basic operations, memories may implement additional functionality such as *burst read* – fetch the data stored at a sequence of memory locations starting from a given address.

### *4.1.2 Memory Structure*

Figure 4.2 shows a simplified view of a typical memory structure. The core storage area is organized into rows and columns of memory cells, conceptually forming a two-dimensional matrix.

The address is split into two parts as shown in Fig. 4.3: a *row address* consisting of the higher order bits, and a *column address* consisting of the lower order bits.

A Read operation can be thought of as consisting of a sequence of three phases as shown in Fig. 4.4 and Fig. 4.5. In the first phase, the row address is decoded in the *row decoder*, resulting in the activation of one of the *word lines*. This selects one row of cells in the memory, which causes transfer of data between the cell and the *bit lines* in the second phase – these lines run through all the rows of the memory. A *sense amplifier* detects the transitions on the bit lines and transmits the result on the data bus. In the third phase, a *column decoder* selects the bits from the row of cells and transmits the addressed bits to the data bus.

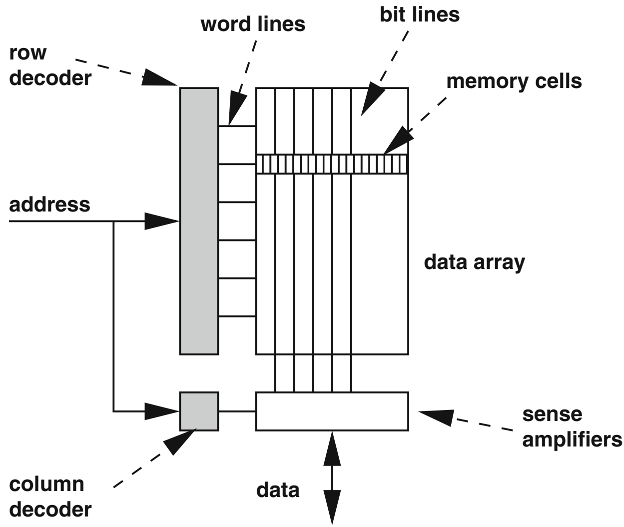



**Fig. 4.2** Simplified view of typical memory structure. The most significant address bits are decoded by the *row decoder* to select a row through the *word line*. *Bit lines* attached to the selected row carry the data to *sense amplifiers*. The *column decoder* now selects the data from the right column and forwards to the data bus

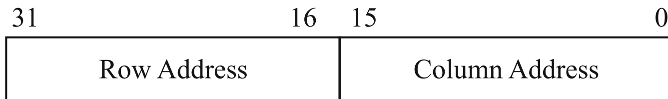



**Fig. 4.3** Division of the memory address into Row Address and Column Address. The row address is the most significant part, and is used by the row decoder to select the word line. The column address is the least significant part, and is used by the column decoder to select the right data from within the selected row

### *4.1.3 Cache Memory*

Modern applications present an inherently contradictory set of requirements from the memory system: large amounts of data have to be stored and retrieved, yet the access delay and energy dissipation should be small. Obviously, larger memories lead to longer access times and larger energy per access. To alleviate this so called *memory wall* or *memory bottleneck*, system architects usually resort to a *memory hierarchy*, consisting of several levels of memory, where higher levels comprise larger memory capacity and hence, longer access times. The memory hierarchy operates on the principle of *locality of reference*: *programs tend to reuse instruction and data they have used recently (temporal locality) and future accesses are likely to be in the same vicinity as past accesses (spatial locality)*. Thus, the first time an instruction or data is accessed, it might have to be fetched from a higher memory level, incurring

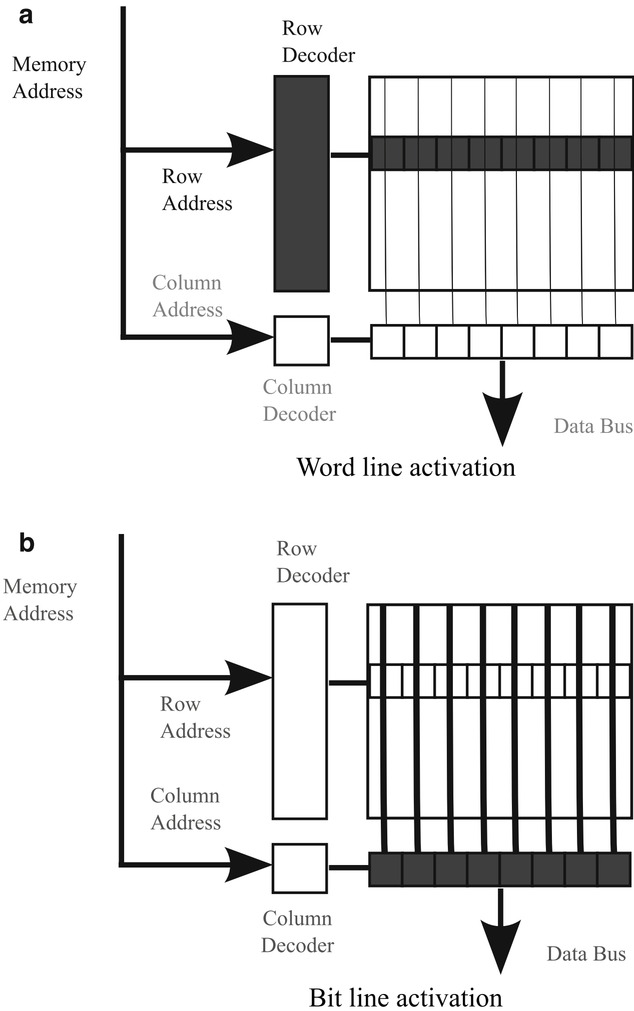



**Fig. 4.4** (**a**) Phase 1 of Memory READ operation: the row address is decoded by the row decoder and a word line is activated. (**b**) Phase 2 of Memory READ: data in the memory cells are transferred via the bit lines to the sense amplifiers

a relatively higher memory access time penalty and energy dissipation. However, it can now be stored in a lower memory level, leading to faster retrieval on subsequent accesses to the same instruction or data. The different memory levels used in most processor architectures are usually: register, cache memory, main memory, and secondary memory.

Cache memory is the next memory level after registers and stores recently accessed memory locations – *instruction cache* stores recently accessed instructions and *data cache* stores recently accessed data. The two are sometimes combined into a single cache. Lower levels of cache usually reside on-chip with the processor and

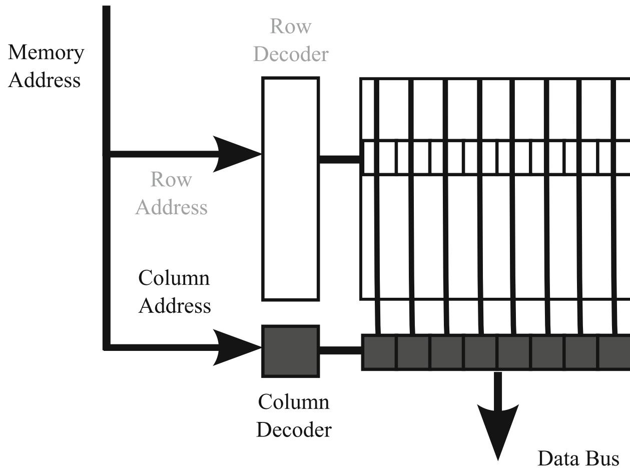



**Fig. 4.5** Phase 3 of Memory READ operation: the column decoder selects the correct offset from the column address and the addressed data is forwarded to the data bus

access times for cache memory usually range from one to a few CPU cycles. On-chip caches in modern commercial general-purpose microprocessors could be as large as a megabyte. Beyond the last level of cache lies the main memory, which usually resides off-chip and is also volatile – the contents disappear when the power is reset. The main memory may be backed by some form of non-volatile secondary storage such as disk or flash memory.

Figure 4.6 shows a generalized memory hierarchy with the levels of hierarchy described above. The register file is usually incorporated into the CPU. The cache, in turn, could consist of multiple levels of hierarchy, of which the lower levels are usually located on-chip with the processor, and higher levels could be in off-chip SRAM (Static Random Access Memory). The main memory is typically implemented in DRAM (Dynamic Random Access Memory) technology, which affords higher density than SRAM, but lower access speed.

The principle of locality of reference leads to data and instructions being found in the lowest level of the cache memory hierarchy closest to the processor most of the time. When the required data or instruction is found in the level of memory that is being searched, a *Cache Hit* is said to have occurred. *Cache misses* occur when instructions or data requested by the processor are not present in the cache, and need to be fetched from the next level of the hierarchy. Cache misses can be classified into three categories [12]:

**Compulsory misses.** These are caused when a memory word is accessed for the first time at the current cache level. Since it is being accessed for the first time, it is obviously absent from the cache and needs to be fetched from the next level of the memory hierarchy.

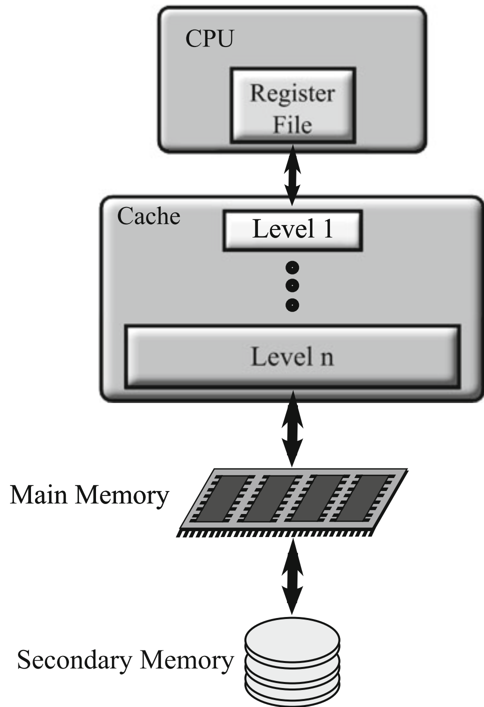



**Fig. 4.6** Hierarchical memory structure: the register file is closest to the CPU, followed by the cache levels, followed by main memory (DRAM) and secondary storage (Disk, Flash memory)

**Capacity misses.** These are caused when cache data that would be needed in the future is displaced due to the working data set being larger than the cache. The cache designer's efforts to anticipate and store the required data from the next level may not be always successful because of the cache's limited size.

**Conflict misses.** These are caused when data present in the cache and useful in the future, is replaced by other data, in spite of the availability of cache space. This happens because of limitations of the mechanism used to search and replace memory words in the cache. These limitations arise out of access time constraints imposed by the system.

The *Cache Miss Ratio* is defined by the equation:

$$\text{Cache Miss Ratio} = \frac{\text{Number of Cache Misses}}{\text{Number of Cache Accesses}} \tag{4.1}$$

The cache miss ratio is a fraction between 0 and 1, often expressed as a percentage, and indicates the fraction of accesses that could not be serviced from the cache, and led to accessing of the next cache level.

*Cache Hit Ratio* is defined as: 1 − Cache Miss Ratio.

A *cache line* consists of a set of memory words that are transferred between the cache and main memory on a cache miss. A longer cache line reduces the compulsory misses, but increases the *cache miss penalty* (the number of CPU cycles

required to fetch a cache line from main memory), and would also increase the number of conflict misses.

An elementary question that determines the working of the cache is the address mapping between the memory address and the cache location. For this purpose, the main memory is divided into blocks of the cache line size. Given a memory block address, a mapping function determines the location of the block in the cache.

The simplest cache design is a *direct-mapped* cache. Here, every memory block can be stored in exactly one cache location given by the equation:

$$\text{Cache Line} = (\textit{Block Address}) \bmod (\textit{Cache Size}) \tag{4.2}$$

where *Block Address* refers to the main memory block number and cache size is the number of lines in the cache. The mapping of memory blocks to cache locations is illustrated in Fig. 4.7, with a memory size of 1023 blocks and a cache size of 8 lines. Memory block $n$ maps to cache line $n$ mod 8. Since the cache is smaller, multiple blocks will map to the same cache line. Hence, the limited cache space needs to be managed effectively. In our example, suppose we access memory block 2 first, followed by block 26. Since both blocks map to the same location, block 26 displaces block 2 from the cache. Thus, if block 2 is needed later, we incur a cache miss due to the conflict between the two blocks.

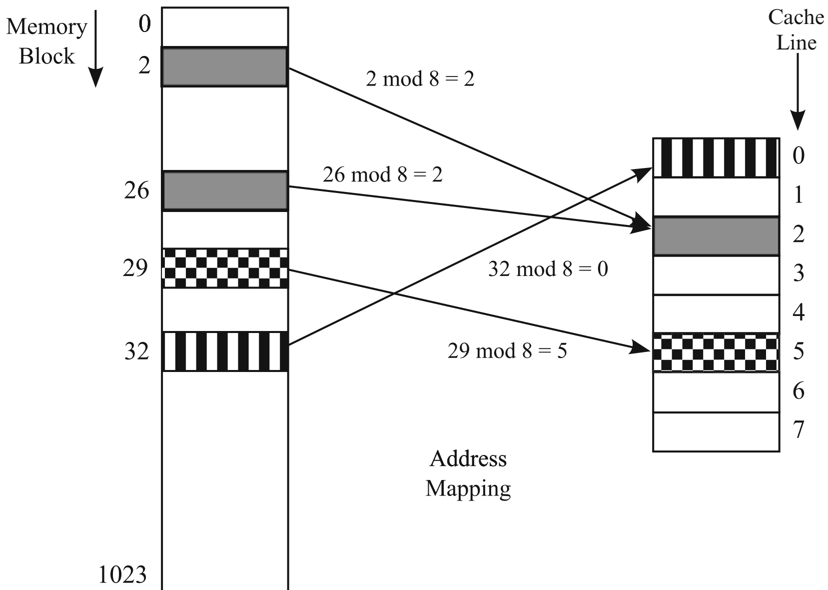



**Fig. 4.7** Direct-mapped cache. For a cache with 8 lines, the cache location of memory block address $n$, is given by $n$ mod 8. Cache conflicts occur when two blocks (2 and 26) map to the same cache location

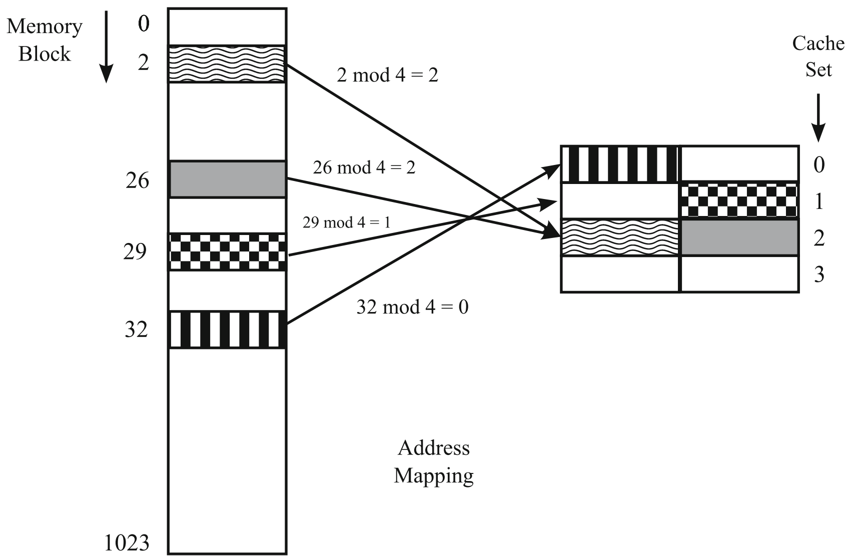



**Fig. 4.8** A two-way set-associative cache. The 8 lines are divided into 4 sets of 2 lines each. The cache set for memory block address $n$, is given by $n$ mod 4. The block can stay at either way of the selected set. This resolves cache conficts between the two blocks at adddress 2 and 26

Set-associative caches help reduce the cache conflict problem mentioned above. An $A$-way set-associative cache is divided into *sets* of $A$ lines each. Each memory block maps to exactly one set, but within the set, the block could reside in any of the constituent $A$ lines. The address mapping in a 2-way set-associative cache ($A = 2$) is illustrated in Fig. 4.8, with a memory size of 1023 blocks and cache size of 8 lines. Blocks 2 and 26 no longer conflict in the cache because they are accommodated in the two lines corresponding to the two cache ways.

In a fully associative cache (illustrated in Fig. 4.9), a given memory block can reside at any of the cache locations. As long as the *working set* of a program is smaller than the cache, conflict misses do not occur in these caches, but capacity misses may still occur when the *working set* of a program is larger than the cache. A fully associative cache is essentially an $N$-way set-associative cache (where $N$ is the number of cache lines), whereas a direct mapped cache is a 1-way set-associative cache.

The number of cache lines in a direct mapped cache, and the number of sets in a set-associative cache, is an exact power of two (of the form $2^m$), so that the mapping function is very simple to implement – the $k$ lower order bits of the block address gives the cache line/set location.

While set-associative caches typically incur a lower miss ratio than direct-mapped ones by eliminating some cache conflicts, they are also more complex, and are characterized by longer access times, because, now, $A$ searches need to be performed to determine if a data element exists in the cache, as opposed to a single search in the case of direct-mapped caches. Further, the additional work leads to an

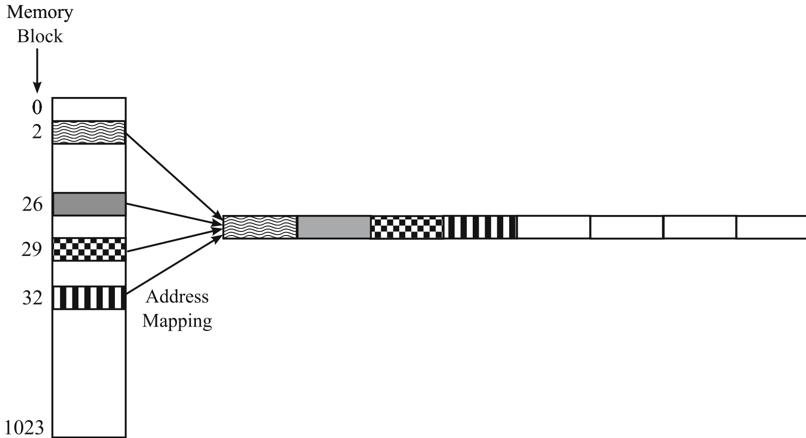



**Fig. 4.9** A fully associative cache. Any memory block can reside at any of the 8 cache lines. There are no cache conflicts, but capacity misses can occur

increase in the cache energy dissipation. Conflict misses can be avoided by using a fully associative cache, but due to access time and power constraints, most cache memories employ a limited-associativity architecture.

An additional feature in associative caches is the need to implement a *replacement policy* for deciding which cache line to evict from a cache set when a new cache line is fetched. In Fig. 4.8, if another block at memory address 34 (which also maps to set 34 mod 8 = 2) were accessed, the replacement policy would help decide which of block 2 and 26 is replaced. A common replacement policy is *Least Recently Used* (LRU) [12], in which the cache line that has not been accessed for the longest duration is replaced, in keeping with the principle of locality.

### *4.1.4 Cache Architecture*

A simplified diagram of the architecture of a typical direct-mapped cache is shown in Fig. 4.10. The memory address presented to the cache consists of three logical fields:

**Offset** This field, consisting of the lower order address bits, indicates which word within a cache line is to be accessed. If the cache line has $2^l$ words, then the offset field has $l$ bits.

**Index** This field indicates the address of the set within the cache where the line will reside, if it is present in the cache.

**Tag** This corresponds to the higher order bits of the address, and is stored along with the data lines. The tag bits are used to identify which specific line (out of the several lines in memory which could possibly be mapped into the indexed location) currently resides in the cache.

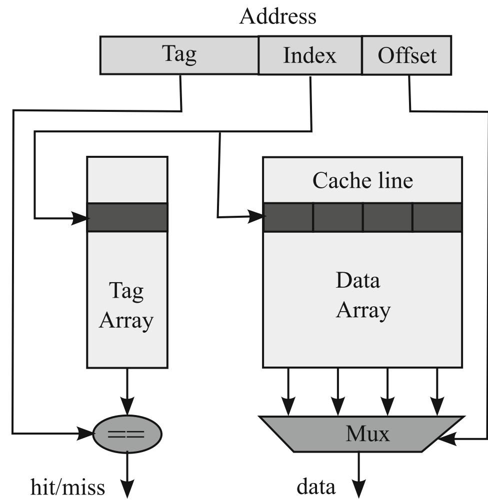



**Fig. 4.10** Simplified architecture of Direct Mapped Cache. Two separate memories store the tag and data arrays. The index is used to fetch the tag and cache line from the two arrays. If the fetched tag matches the tag part of the address, then we have a cache hit, and the offset part of the address is used to select the right data from the cache line. If the tag does not match, we have a cache miss

The two major components of a direct-mapped cache are the Data Array and the Tag Array (Fig. 4.10). Suppose we access memory line $L$ located at memory address $A$, and the index, tag, and offset fields of address $A$ are given by $i(A)$, $tag(A)$, and $o(A)$ respectively. The contents of $L$ are stored in the data array of the cache in anticipation of temporal and spatial locality, at address $i(A)$. The $i(A)$ location of the tag array contains $tag(A)$. When a new address $B$ is presented to the cache, the cache line data at the $i(B)$ address is fetched from the data array. Simultaneously, the tag bits stored at address $i(B)$ in the tag array are also fetched. A comparator compares these stored tag bits with $tag(B)$. If the comparison succeeds, then we have a Cache Hit and the data bits read from the data array are the correct data. The offset field $o(B)$ is used to select the appropriate data from among the different words in the cache line. If the comparison fails, we have a Cache Miss and the address now needs to be sent to the next level of the memory hierarchy.

Figure 4.10 omits some other components of the cache such as control bits (Valid, Dirty, etc.) that are stored along with the tag bits in the cache. The Valid bit is used to distinguish cache contents from random values that might be stored at initialization. The Dirty bit is used to ensure that when a cache line is replaced, it is written back to the next memory level only if it is modified at the current level.

The architecture of a 4-way set-associative cache is illustrated in Fig. 4.11. There are four banks of data and tag arrays. The indexed cache line is read out from all the four data arrays. The tag bits are also read out from the four tag arrays and compared to the tag bits of the address. If there is a match with any of the stored tags, the output of the corresponding comparator will be '1' (and that of the others will be '0'). This leads to a cache hit and the comparator output bits are used to select the data from the correct data array. If all comparisons fail, then we have a cache miss.

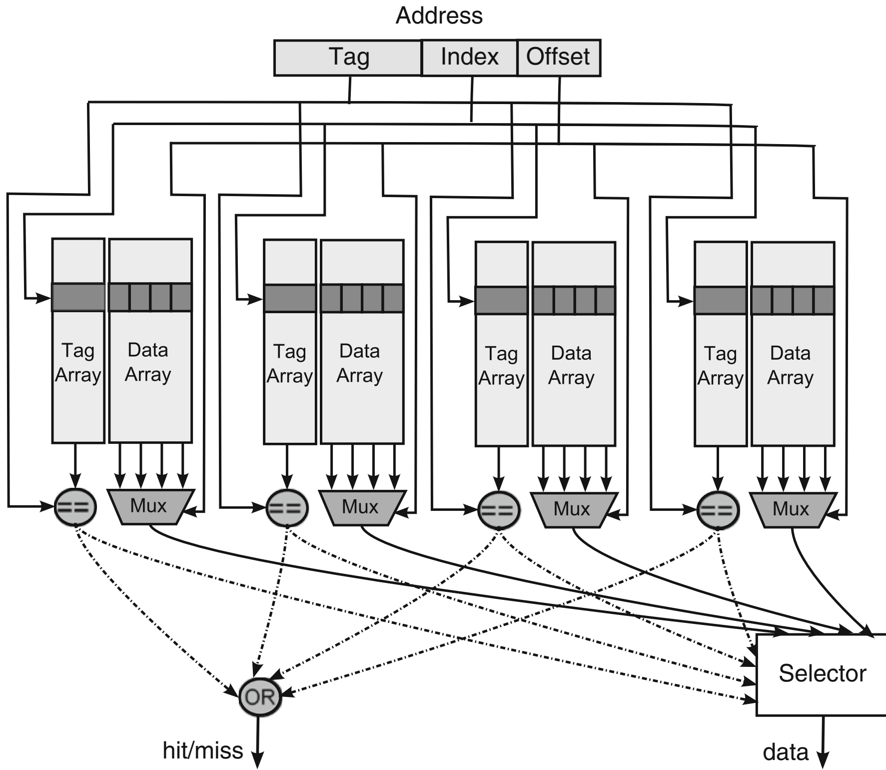



**Fig. 4.11** Simplified architecture of 4-way associative cache. We have four different tag and data arrays. The index is used to fetch the tag and data from all four arrays. If the address tag matches any of the fetched tags, we have a cache hit, and the data at the corresponding cache line is selected

## *4.1.5 Power Dissipation During Memory Access*

Power dissipation during memory accesses can be attributed to three main components:

1. address decoders and word lines
2. data array, sense amplifiers, and the bit lines
3. the data and address buses leading to the memory.

All three components are significant as each involves the driving of high capacitance wires that requires a considerable amount of energy: word lines, bit lines, and data/address buses. Power optimizations for the memory subsystem indirectly target one of these components and can be classified into the following broad categories:

- **Power-efficient memory architectures** – novel architectural concepts that aid power reduction, both in traditional cache memory design and in other unconventional memory architectures such as scratch pad memory and banked memory.

- **Compiler optimizations targeting memory power** – where code is generated for general-purpose processors targeting power reduction. This topic is explored in more detail in Chapter 5.
- **Application specific memory customization** – where the memory system can be tailored for the particular application, leading to superior solutions than a standard memory hierarchy.
- **Transformations: compression and encoding** – these are known techniques from other domains that are also applicable to memory power reduction.

How is memory optimization for high performance different from memory optimization for low power? There are some classes of optimizations that result in improving both performance and power – these are the optimizations that attempt to reduce the number of memory accesses. If the number of accesses to memory is reduced, then performance is improved because this results in reduced total latency. Similarly, reduced memory access count also means reduced energy. Most optimizations belonging to the classes of techniques summarized above specifically target low power, and are orthogonal to standard performance improving memory optimizations targeted by standard compilers. We must point out that most advanced cache architecture features that aim at improving performance by effectively reducing the number of cache misses (i.e., reducing the number of accesses to the next level of memory), also improve power as an obvious consequence. However, since they were proposed as primarily performance enhancement techniques, we do not discuss them in detail in this book. The reader is referred to a standard computer architecture text such as [12] for a more comprehensive discussion of all performance-oriented cache features that also improve power by the simple consequence of reducing the miss rate.

## 4.2 Power-efficient Memory Architectures

The memory subsystem in embedded processor based systems usually consists of cache memory, along with other memory modules possibly customized for the application. Because of the dominating role of instruction and data caches, new low power memory architectures have been in the area of improving traditional cache designs to make them power-efficient using a variety of techniques.

### *4.2.1 Partitioned Memory and Caches*

Partitioning the memory structure into smaller memories (or banks) is one of the mechanisms used to reduce the effective length of the bit-lines driven during memory operations, thereby reducing power dissipation. In the multi-bank memory example shown in Fig. 4.12, the memory is physically partitioned into four banks, each with 1/4 the size of the original monolithic memory. This causes each bit line

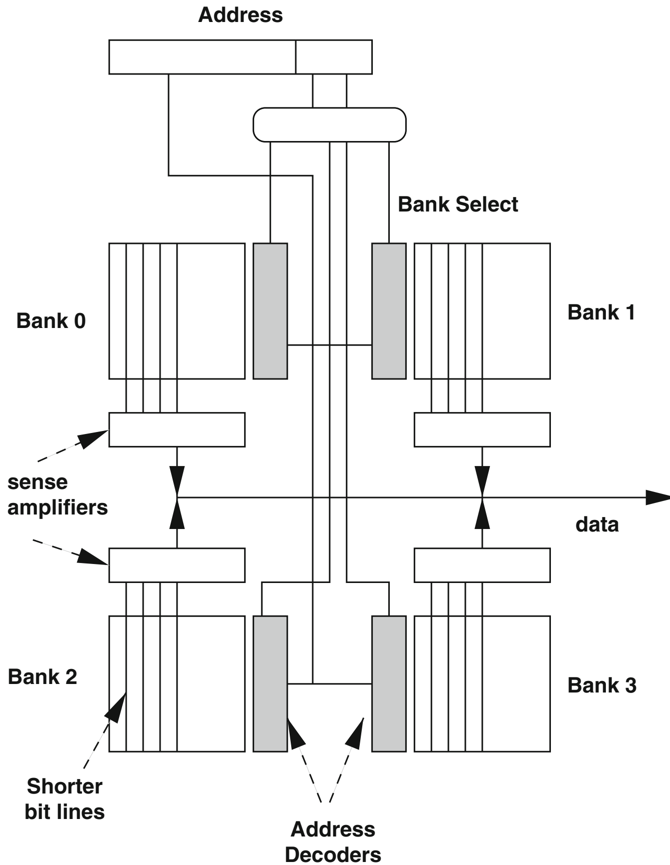



**Fig. 4.12** Memory banking reduces bit-line capacitance. The memory array is physically divided into multiple banks. Each cell now needs to drive a smaller bit-line within a bank

to be of 1/4 the original length, and a proportional decrease in the switching bit line capacitance. The sense amplifiers are replicated in each bank. The lower order bits of the address (two LSB bits in Fig. 4.12) are used to generate a *select* signal to the right bank. Since only one of the banks would be activated during a memory operation, this would lead to reduced power consumption in the overall structure because of the considerably reduced energy from the switching of the smaller bit-lines.

The concept of banking is also naturally applicable to cache memory, since the bulk of the cache consists of two internal memories: the data and tag arrays. Cache banking and other partitioning studies (such as bit-line segmentation that are conceptually similar) are reported in [10, 23, 36]. One proposed variant is to make the smaller partitioned units *complete caches* as opposed to just memory banks [21]. The added flexibility here is that the different caches need not be homogeneous.

A prediction mechanism such as *most recently used* is employed to predict which sub-cache will be accessed next, and the result is used to turn the other sub-caches into low power mode.

### 4.2.2 Augmenting with Additional Memories

A large number of low power cache ideas have been formulated around one central principle: add an extra cache or buffer, usually small in size, and design the system to fetch data directly from this buffer in the steady state, thereby preventing an access to the L1 cache altogether. Since the buffer is relatively small, we can achieve significant power savings if we can ensure a high hit rate to the buffer.

The technique of *block buffering* [36] stores the previously accessed cache line in a buffer. If the next access is from the same line, then the buffer is directly read and there is no need to access the core of the cache. This scheme successfully reduces power when there is a significant amount of spatial locality in memory references (Fig. 4.13). The idea of a block buffer can be extended to include more than one line instead of just the last accessed line. A fully associative block buffer is illustrated in Fig. 4.14 [10]. Recent tags and data are stored in fully associative buffers associated with each set. If a new tag is found in this buffer, then the tag array read is inhibited. The matching address in the buffer is used to fetch the corresponding data from the data buffer. The correct size of this fully associative buffer will have to be determined based on an engineering trade-off because a fully associative lookup in

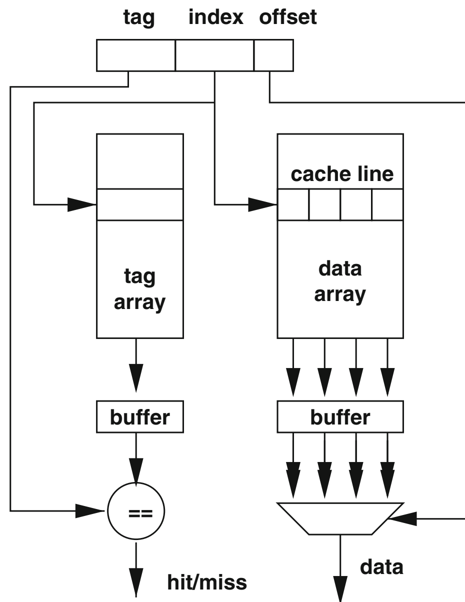



**Fig. 4.13** Block Buffering. The last tag and cache line data are buffered. If the current tag matches the buffered tag, then there is no need to fetch from the memory array, thereby saving power

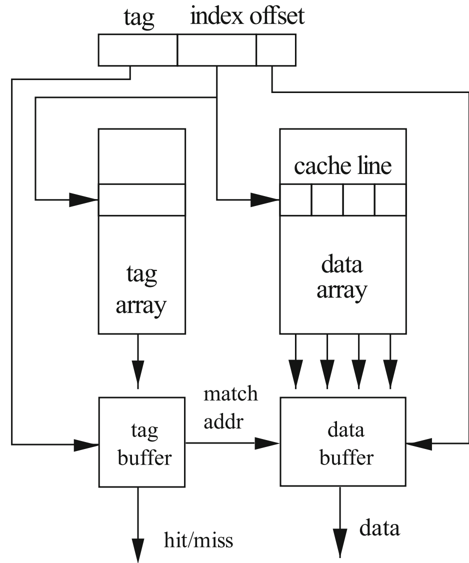



**Fig. 4.14** Fully associative block buffer. An extension of the block buffer idea. Recent tags and data are stored in the buffer and looked up on a cache access. If found, then there is again no need to access the memory array. This saves power, as long as the buffer is not too big, and manages to catch a substantial number of accesses

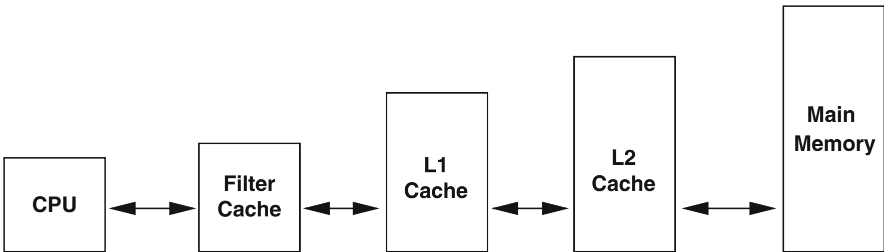



**Fig. 4.15** Filter cache. A small cache placed between the CPU and the L1 cache aims at catching a significant number of accesses. Power is saved by keeping the filter cache small

a buffer is a very power-hungry operation. If the buffer is too large, then the power overheads due to the associative lookup may overwhelm the power savings due to the hit in the buffer. If the buffer is too small, then it may result in too many misses. In practice, such fully associative lookups usually restrict the size to 8 or less.

One simple power reduction strategy in caches is to introduce another level of hierarchy before the L1 cache, placing a very small cache (called a *filter cache* [22]) between the processor and the regular cache. This causes a performance overhead because the hit ratio of this small cache is bound to be lower, but it leads to a much lower power dissipation on a cache hit. If the hit ratio is reasonably high, then there may be an overall power reduction. This is illustrated in Fig. 4.15. No overall modification is proposed to the overall cache hierarchy, except that the filter cache is unusually small compared to a regular L1 cache.

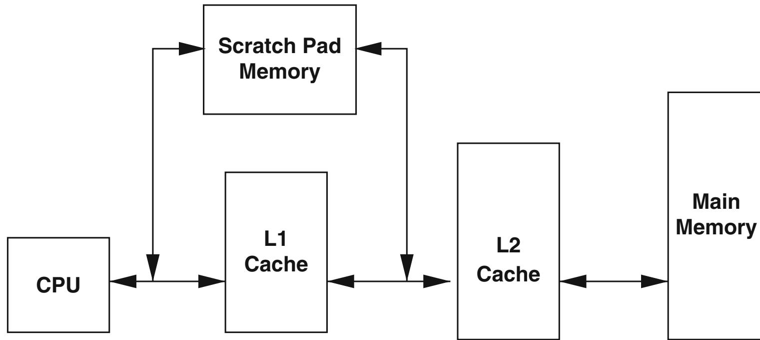



**Fig. 4.16** Scratch pad memory in a memory hierarchy. The SPM is small and fast, and resides at the same level as the L1 cache. Power is saved because there is no hardware management of SPM contents, unlike in caches

Data and instructions can also be statically assigned to an additional on-chip memory whose address space is disjoint from the cached part. In *Scratch Pad Memory* [32], data is statically assigned by the compiler keeping in mind the data size and frequency of access (Fig. 4.16). Unlike in caches, scratch pad memory contents are never automatically evicted – the compiler or programmer explicitly manages the space and “hits” are guaranteed. Techniques to exploit this architectural enhancement are discussed in Section 4.4.

The well known observation that programs tend to spend a lot of time inside relatively small loop structures, can be exploited with specialized hardware. *Loop cache* [3] is one such structure consisting of an augmentation to the normal cache hierarchy. Frequently executed basic blocks within loops are stored in the loop cache. The processor first accesses the loop cache for an instruction; if it is present, there is no need to access the normal cache hierarchy, else the instruction cache is accessed. The *Decoded Instruction Buffer* [1] is analogous to the loop cache idea, but here, the decoded instructions occurring in a loop are stored in the buffer, to prevent the power overhead associated with instruction decoding. The decoded instructions are written to the buffer in the first loop iteration; in subsequent iterations, they are read off the buffer instead of the L1 instruction cache (Fig. 3.16).

### *4.2.3 Reducing Tag and Data Array Fetches*

For performance-related reasons, the tag array and the data array in the cache are accessed *simultaneously* so that by the time the tag bits of the address of the resident cache line are fetched from the tag array and compared with the tag bits of the required address to detect hit or miss, the corresponding cache line data is already available for forwarding to the processor [12]. Thus, the fetching of the cache line data is initiated even before we know whether the access is a hit or a miss; on a miss, it is simply discarded. Since we expect most accesses to be hits, this parallel

access strategy improves performance significantly. In a set-associative cache, all the tag arrays and data arrays are accessed at once. While designed for optimal performance, this overall strategy results in waste of power, since in a $k$-way associative cache, at least $k-1$ fetches from the data array are discarded. Since cache lines are usually wide (cache lines of length 8-32 words are common), the power wasted here is substantial, leading to a significant scope for trade-offs between performance and power.

The simplest power optimization addressing the above issue is to sequentialize the accesses to the tag and data arrays – that is, to fetch from the data array only if the tag fetch indicates a cache hit. This prevents dynamic power dissipation incurred when data is fetched from the data array in spite of a cache miss [11]. Moreover, data only needs to be fetched from the way that matched, not from the other ways. The idea is illustrated in Fig. 4.17. Shaded blocks indicate data and tag arrays that are active in the respective cycles. In the conventional cache of Fig. 4.17(a), all tag and data arrays are shaded, indicating that all are accessed in the same cycle. In the low

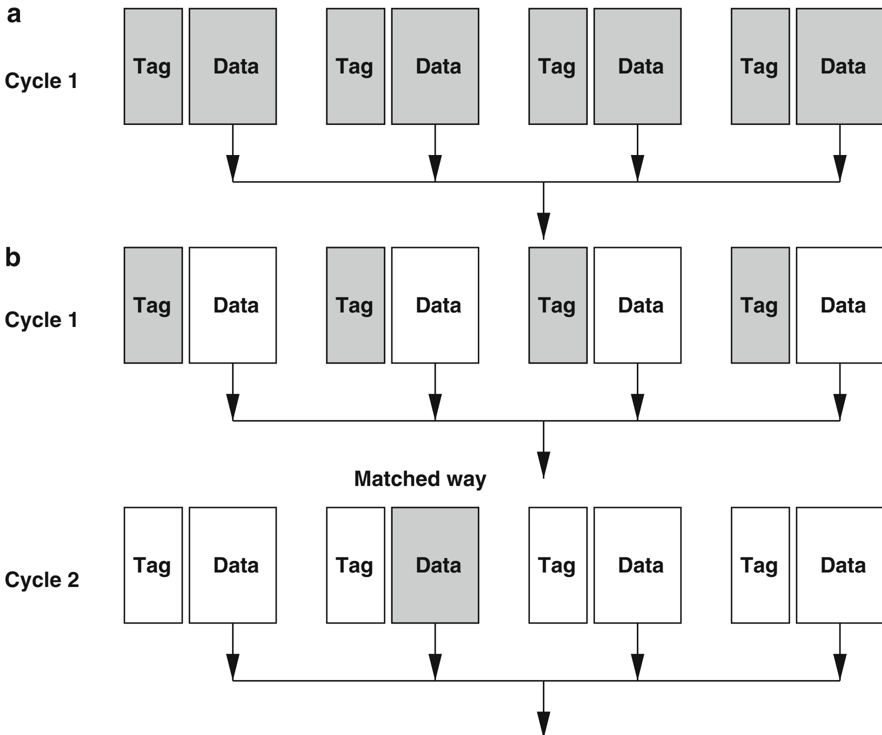



**Fig. 4.17** Power saving by accessing the data array on successful tag match. (**a**) In a conventional set-associative cache, all tags and data arrays are accessed simultaneously. (**b**) When we sequentialize tag and data accesses, we first fetch only the tags. If a match is found, then data is fetched only from the matching way. Power is saved due to the avoided accesses, at the expense of time

power cache of Fig. 4.17(b), in the first cycle, all the tag arrays are accessed using the index field of the address and the tag bits are read and compared with the tag field of the address – only the tag arrays are shaded in Cycle 1 of Fig. 4.17(a). In Cycle 2, the data array of only the matched way is accessed. This leads to a performance penalty of an extra cycle, but leads to a straightforward dynamic energy reduction due to the three ways for which the data arrays are not accessed.

The above approach reduces cache access energy but compromises on performance. Another simple idea is (in case of instruction cache) to retain the address of the last accessed cache line, and to fetch from the tag array only if the next instruction refers to a different line [33]. If the reference is to the same line, then we are guaranteed a cache hit, and power is saved by preventing the redundant access of the tag array. The above is similar to the block buffering strategy, but can be generalized in an interesting way: we can assert that if there has been no cache miss since the last reference to a basic block of instructions, then there is no need to fetch the tag from the instruction cache in the next reference (since the previously fetched instruction has not had an opportunity to get evicted yet). The information about whether the target of a branch instruction exists in the cache is recorded in the Branch Target Buffer, a commonly used structure in modern processors. If the condition is satisfied, then the fetch from the tag array is disabled, saving memory activity in the process [15].

The observation that, in set-associative caches, consecutive references tend to access data from the same way, can be exploited in a mechanism that predicts the way number for the next access to be the same as the previous one. On the next access, only the tag and data arrays of the predicted way are activated, resulting in a significant amount of dynamic power savings when the prediction is correct [14]. When the prediction turns out to be incorrect, the rest of the ways are fetched in the next cycle, which incurs a performance penalty. This is illustrated in Fig. 4.18.

An alternative method of incorporating way prediction is through the *location cache* – a proposal for the L2 cache [28]. This is an extra cache that is used to indicate which way in the actual cache contains the data. A hit in the location cache indicates the way number, and hence, we only need to access the specific way in the L2 cache, thereby avoiding reading all the tag and data arrays of the set-associative cache. A miss in the location cache indicates that we do not have a prediction, and leads to a regular access from all the ways in the L2 cache. This is illustrated in Fig. 4.19. The location cache needs to be small in order to ensure that its power overheads do not overwhelm the saved power.

A certain amount of flexibility can be built into set associative caches to control accesses to the different ways – ways can be selectively enabled or disabled depending on the characteristics of an application. For example, in the L2 cache, we can reserve separate ways for instruction and data so as to prevent conflicts. Also, for small programs where instruction cache conflicts are not expected, some of the ways assigned to instructions can be disabled to reduce activity in their tag and data arrays [26]. In the *way-halting cache* [38], some least significant tag bits from each way are stored in a separate array, which is first accessed and the corresponding bits

Cache hit on predicted way. Single cycle.

Cycle 1 Tag Data Tag Data Tag Data Tag Data

Cache miss on predicted way. Two cycles

Predicted way

Cycle 1 Tag Data Tag Data Tag Data Tag Data

Predicted way

Cycle 2 Tag Data Tag Data Tag Data Tag Data

**Fig. 4.18** Power saving by Way Prediction in associative caches. Tag and data are fetched only from the predicted way. If prediction is correct, this reduces power by avoiding accesses to all the other ways. If incorrect, then all other ways are accessed, losing some time

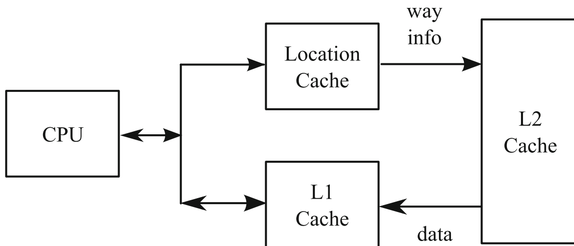



**Fig. 4.19** A Location Cache stores the predicted L2 way. When L2 is accessed, only the predicted way is looked up first

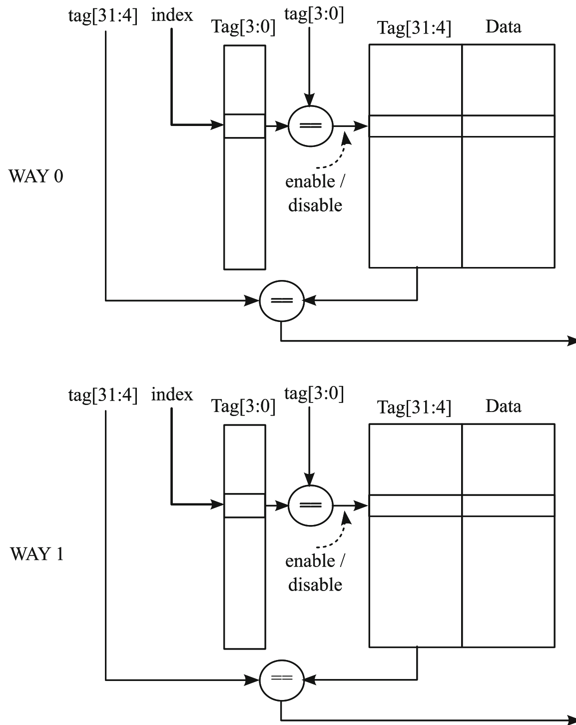



**Fig. 4.20** Way-halting cache. A subset of the tag bits are first compared for equality. If unequal, then there is no need to fetch the remaining tag bits

of the address compared. If the comparison fails, then a matching failure is guaranteed for that way, and hence, the actual fetch of the tag and data arrays is prevented, saving power. This is shown in Fig. 4.20. The *enable/disable* signal is generated for each way by comparing the least significant 4 bits of the tag. On mismatch, the fetching of the remaining tag bits and data are disabled.

## *4.2.4 Reducing Cache Leakage Power*

The techniques discussed in previous sections target dynamic power consumption in caches. As mentioned earlier, the importance of static power has been growing in recent years, and with smaller geometries, the contribution of static power to the overall power consumption is growing. Static power is dissipated as long as a voltage is supplied to the circuit, and can be eliminated by turning power supply off (in which case memory data is lost) or reduced by turning the voltage down

(in which case data can be retained, but accessing the data requires us to raise the voltage again). A few strategies have been proposed to address the static power dissipation in caches.

An important observation regarding lifetime of cache contents is that, the data tends to be accessed frequently in small intervals of time, and is essentially dead for large periods when it is never accessed. This leads to an interesting question – can we turn off power to a cache line if we predict that the line will not be accessed in the near future? The *cache decay* technique [20] relies on this approach. A counter is maintained for each line; when it reaches a threshold value with no access to the cache line, the power to the line is turned off after updating the next level cache with the new value if necessary. The counter is reset on an access to the cache line. To keep the overhead of maintaining the counters low, there is only one global counter, but a two-bit derived counter is placed in each line to control the power supply. The threshold value of the counter is determined from the values of the static energy dissipated and the energy expended in re-fetching the line from the L2 cache. This is illustrated in Fig. 4.21.

An alternative technique to turning off the power to cache lines is to turn down the voltage so that data is retained, but cannot be accessed directly. In order to access the data, the line would have to be first switched to high voltage (causing a performance overhead). The power saved per line in this manner is smaller than that in the decay technique where the power supply to the line is turned off, but this may permit more lines to be moved into the *drowsy* state [8]. The idea, called *drowsy cache*, is proposed to be used for all lines in the L2 cache, and some lines of the L1 cache. A simple strategy of putting the entire cache to drowsy mode once in a while works quite well, as opposed to introducing active circuitry to maintain counters on a per line basis. A variation on this theme is to use predictive schemes to selectively move lines to and from drowsy state. Information about cache lines with high temporal locality is maintained in the BTB, and these lines are kept active. When a cache line is accessed, the sequentially following line is moved to active state, anticipating its use. A very fine grain control can be exercised by permitting the compiler to insert instructions to turn individual cache lines off when its use is not anticipated for a long time [13, 39].

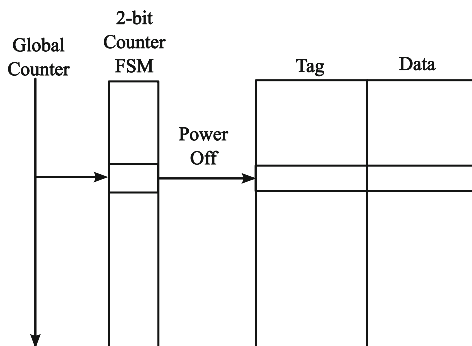



**Fig. 4.21** Cache decay. If a cache line has not been accessed for some time, then turn off the power to the line

## 4.3 Translation Look-aside Buffer (TLB)

The concept of virtual memory was developed to relieve the programmer of the burden of managing physical memory during program execution. The operating system automatically manages the loading and relocation of the program at execution time.

For efficient memory management, the address space is divided into *pages* which are analogous to cache blocks/lines, but at a higher level of granularity. The page size typically varies from 4096 to 65536 bytes. A CPU with virtual memory generates a virtual address which gets translated into a physical address using hardware and/or software approaches. This is referred as *address translation*. A *page table* is used for mapping virtual address to physical address as shown in Fig. 4.22. The page table contains mapping information of virtual pages to the available physical pages. Thus, the page table size depends on the available physical memory size and the size of each page. Typical page tables could be as large as 4 MB and are hence usually stored in main memory. A miss in virtual memory would require an access to secondary memory – usually hard disk – which may exhibit access latencies of millions of processor cycles. Since the miss penalties are very high, a fully-associative strategy is used for placing blocks in the main memory.

Since the translation would impose an additional memory access overhead on every memory access, fast address translation is performed using a special on-chip cache called *Translation Look-aside Buffer* (TLB). TLB is a small cache that stores the virtual to physical mapping information of the most recently accessed memory locations. Typical TLB sizes vary from 8 to 128 entries. The structure of a TLB is similar to that of a normal cache, with the tag array containing the virtual address that is looked-up, and the data array containing the physical address to which it maps. When the CPU requests a memory access, the virtual address is looked up

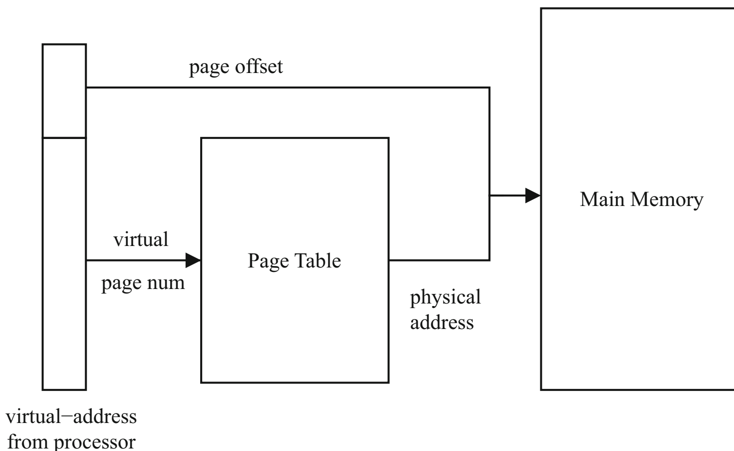



**Fig. 4.22** Address Translation: the virtual address generated by the CPU is translated into a physical address using the page table

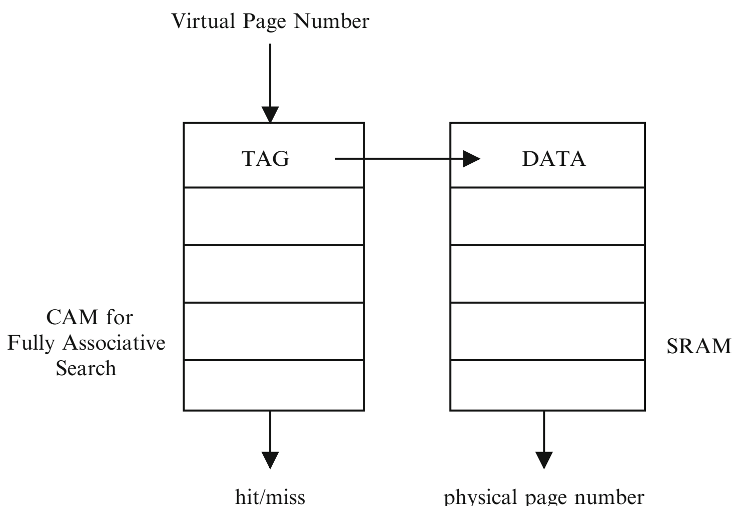



**Fig. 4.23** TLB architecture: logical view of the TLB structure

simultaneously in both the L1-cache as well as the TLB. In order to simplify the hardware, TLBs for instruction and data memories are kept separate.

The instruction TLB (ITLB) is used in every cycle. Since consecutive instructions are usually mapped to the same page, the translation lookups for instruction references can be optimized. In case of data TLB (DTLB), the number of lookups per cycle depend on the number of parallel data accesses supported. For a wide issue superscalar processor, the TLB would require multiple read ports. The logical structure of the TLB is shown in Fig. 4.23. To support fully associative lookup, a content addressable memory (CAM – generally considered to be expensive in terms of power) is used for storing the tags (virtual addresses). A simple SRAM is used for storing data (the corresponding physical address). In a well managed memory system, TLB miss rate is very small – usually less than 1%.

The basic cell implemented for CAM logic would require 9 transistors (shown in Fig. 4.24), compared to 6 transistors for a simple SRAM memory cell. It consists of the standard 6 transistor (6T) structure for storage and three additional transistors for comparing the stored bit with the content on bit lines. All bits in the tag share a common *match* signal. During CAM lookup, all the match lines are precharged and on a mismatch they are discharged by the NOR transistor at the bottom.

In a TLB lookup operation, a maximum of one entry would result in a match, which is then used for reading the corresponding data from the SRAM. Hence, during a lookup, every tag entry (except the matched one) would charge and discharge the match lines leading to a large power dissipation. Larger cell size and matching lines lead to a CAM cell occupying larger area and consuming higher power than a standard SRAM cell. The fully associative structure leads to significant power consumption in the TLB, in spite of it having relatively small number of entries in comparison to caches. In embedded processors with small cache sizes, TLB power

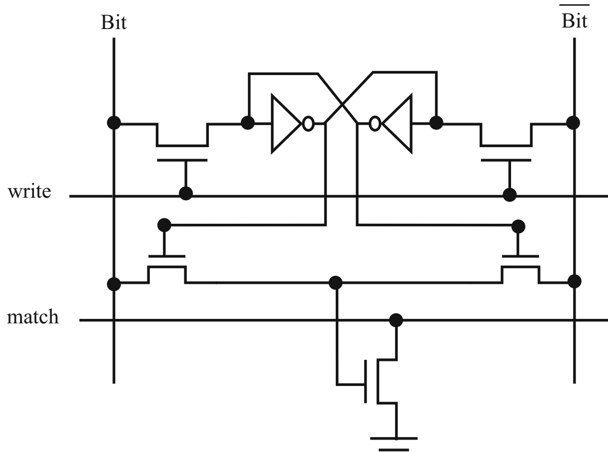



**Fig. 4.24** CAM cell: in the 9-transistor structure of a CAM cell, the standard 6T structure is used for storing the bit while the additional transistors are used for comparing the content with the bit lines for a match

forms one of the major components of the total system power consumption. For instance, in the StrongARM processor, TLB is reported to account upto 17% of the total power.

In this section we discuss some of the important optimizations and trade-offs considered in designing power-efficient TLBs.

### *4.3.1 TLB Associativity – A Power-performance Trade-off*

Ideally, we would like the TLB to be fully associative to minimize the accesses to main memory and hence obtain both power and performance benefits. However, it is well established that the returns in terms of cache hits actually diminish with increasing cache sizes. Since the TLB is also inherently a cache, this principle is also applicable to TLBs. As a consequence, power dissipated in the fully associative search for every lookup in a large cache can overshadow the power saved from reducing the accesses to the main memory. Hence, large fully associative TLBs can have a negative impact on power with little performance improvement. The trade-off between power and performance needs to be carefully evaluated while designing a TLB for a system.

### *4.3.2 Banking*

TLB banking is an attractive low power solution [17, 27]. In a banked architecture shown in Fig. 4.25, the TLB is split into multiple tag and data banks. Each tag

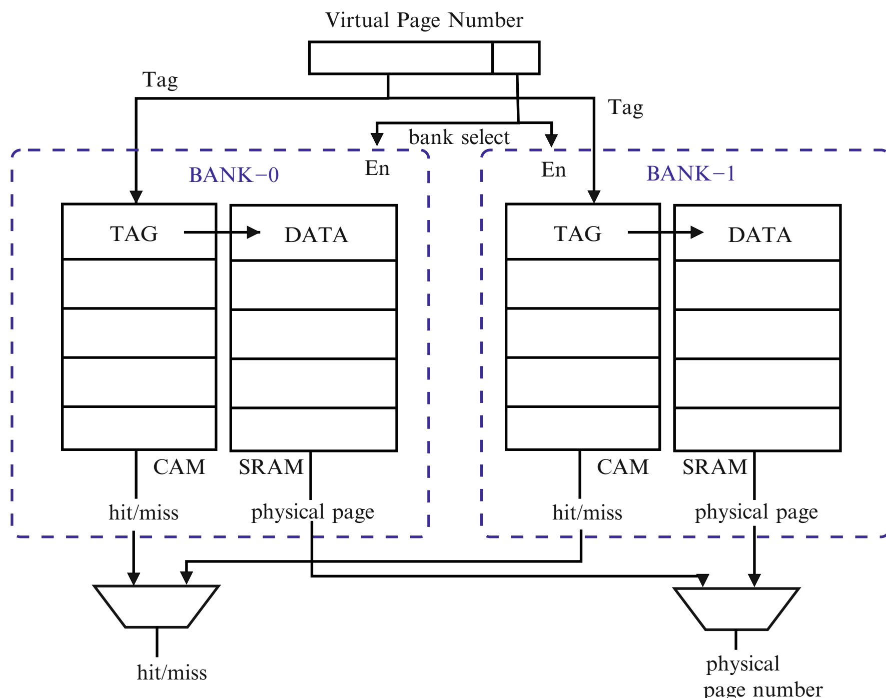



**Fig. 4.25** 2-way Banked TLB architecture: In a banked TLB structure, some bits of the virtual address are used for bank selection. Each bank has a fully associative tag storage and the corresponding data entries in SRAM. During lookup only one bank is activated, thereby reducing the access energy

bank is fully associative, while each data bank is a traditional SRAM. A lookup request to TLB would need to access only one bank, thereby reducing the access energy per lookup by $N$ times, where $N$ is the number of banks. For the same TLB *size* (number of entries), an $N$-way banked architecture would have an associativity of $\frac{size}{N}$ which would result in a performance loss. Thus, unlike traditional cache architectures, banked TLB architecture needs careful study of power-performance trade-offs.

Some advanced architectural enhancements and allocation schemes for reducing performance loss due to banking are discussed below:

**Victim Buffers** – A victim buffer can be used to hold $N$ recently replaced TLB entries, similar to the concept used in caches. The victim buffer is shared by all banks. During a TLB lookup operation, the victim buffer and the appropriate TLB bank are searched in parallel. A hit either in the bank or victim would result in a TLB hit. Thus, victim buffers can alleviate most of the capacity misses caused due to banking and hence improve the performance [5]. The size of the victim buffer

should be selected in a way that the power consumed by searching the buffer is small compared to the power saved due to reduced main memory accesses.

**Allocation Policy** – A more aggressive allocation policy can be used while replacing TLB entries in a banked architecture. In general, if a free entry is not available in the selected bank to hold the mapping for a new page, an existing page is replaced. In a more aggressive policy, the replaced entry could be placed in other banks instead of being discarded, even at the expense of a second replacement. During a TLB lookup operation, first the TLB bank corresponding to the address being looked-up is searched; on a miss, all the other banks are searched. If the entry is not found in any of the banks, then a request to the main memory is sent. This scheme would essentially ensure the same miss rate as that of a fully associative search. However, it is more power efficient if the "hit ratio" to the bank that is searched first is reasonably high, which is generally the case [5].

### *4.3.3 Reducing TLB Lookups*

Since each TLB lookup is very costly in terms of power, intelligent techniques that reduce number of lookups without having an adverse impact on performance form attractive power optimization alternatives.

#### 4.3.3.1 Deferred Address Translation

By employing virtually indexed and virtually tagged caches at the L1 level, address translation would be required only during an L2 access (assuming L2 is physically addressed). Thus, TLB needs to be accessed only on L1 misses. Though this would require an extra cycle for all L1 cache misses, the number of accesses, and hence, energy dissipation in the TLB is considerably reduced [18]. Similarly, if the L2 cache is also virtually indexed and virtually tagged, then the translation could be further deferred and could even be implemented in software by the operating system.

#### 4.3.3.2 Using Address Mapping Register

Modern processors employ separate TLB structures for instructions and data to allow concurrent lookups for both data and instruction references. The ITLB is used whenever an instruction reference requires address translation. Due to the temporal locality property of instruction addresses, there is a very high probability that successive accesses would belong to the same memory page. With page sizes of 4KB to 64KB, one can expect a large number of accesses to the same page before proceeding to the next. This property is exploited by storing the mapping for

the most recently accessed page in a special hardware register. During an ITLB lookup, this register is accessed first for address translation and only if it is not found in this register, the power hungry TLB lookup operation is performed [18]. Since this is stored in a register close to the processor, the overhead in timing on a miss is negligible. The concept is similar to the idea of block buffering (Section 4.2.2).

## 4.4 Scratch Pad Memory

Scratch Pad Memory (SPM) refers to on-chip memory whose contents are explicitly managed by the compiler or programmer [32]. A typical architecture is shown in Fig. 4.26. Address and data buses from and to the CPU could lead to on-chip caches and scratch pad memory, but both these memory modules are optional. If the data or instruction requested by the CPU is present on-chip, in either the scratch pad or the caches, then it is accessed from the respective module. Otherwise, the next level of memory hierarchy (off-chip) is looked up. The implementation of the scratch pad could be in either SRAM or embedded DRAM on chip. The main logical characteristic of the scratch memory is that, unlike caches where the management of the memory content is decided transparently by hardware, in scratch pad the management is explicitly performed by the compiler or programmer. This could have both positive and negative consequences. The advantages are that data and instructions stored in the scratch pad are guaranteed to be present where they were last stored, until they are explicitly moved, which makes access times more deterministic. This not only enables predictability that is of crucial importance in real-time systems, but also simplifies the hardware considerably – there is no need for the tag memory storage, access, and lookup, which saves energy per access when data is

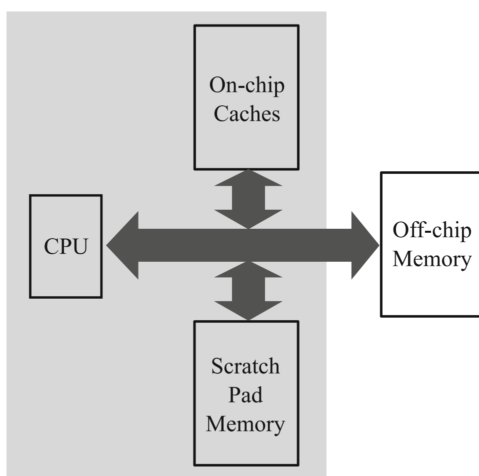



**Fig. 4.26** Scratch pad memory. The CPU's request for on-chip data/instruction can be served from either SPM or on-chip cache. Management of SPM contents must be performed in software

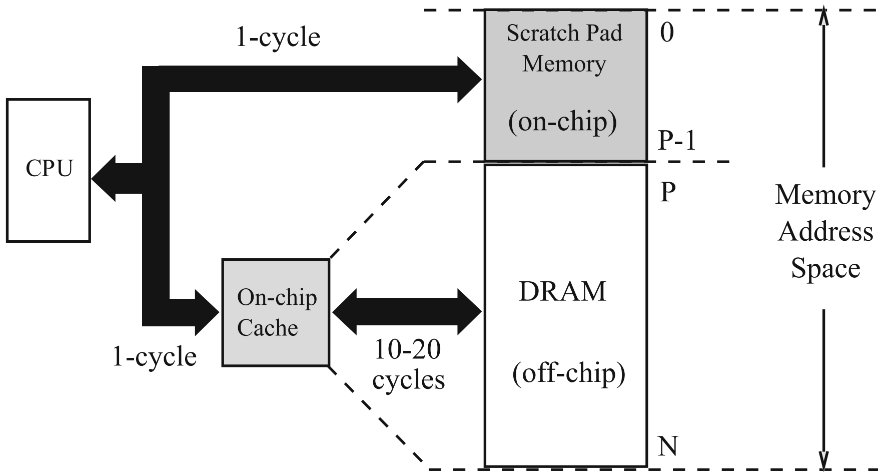



**Fig. 4.27** Address mapping in scratch pad memory. Addresses $0..P-1$ are in SPM. Addresses $P..N-1$ are accessed through the cache. Access to both SPM and on-chip cache can be assumed to be fast

found in the scratch pad. The latter makes scratch pads more energy efficient than caches as long as the memory contents are efficiently managed statically. The disadvantages of SPM based architectures is that, often, program and data behavior may not be easily analyzable by the compiler, making it difficult for it to exploit the SPM well.

A given architectural platform could omit either the scratch pad or the caches. In this section we will assume both are present on-chip, but most of the decision making process about data and instruction mapping into scratch pad memory remains unchanged even if the on-chip caches are absent. Figure 4.27 shows a typical view of the memory address space $0..N-1$ divided into on-chip scratch pad memory and off-chip memory (typically implemented in DRAM). Addresses $0..P-1$ are mapped into scratch pad memory, and $P...N-1$ are mapped to the off-chip DRAM and accessed through the cache. The caches and scratch pad are both on-chip and result in fast access (1 cycle in Fig. 4.27). Accesses to the DRAM, which occur on cache misses, take relatively longer time and larger power (access time is 20 cycles in Fig. 4.27). If the cache is not present in the architecture, then the connection to off-chip DRAM is usually through a Direct Memory Access (DMA) engine.

### *4.4.1 Data Placement in SPM*

A significant role can be played by the compiler when the architecture contains scratch pad memory structures, as these memories are directly managed by the compiler. Compile-time analysis is involved in determining which data and instructions

should be mapped to the scratch pad memory. Since SPM space is limited, we would like to store relatively critical data in it [32]. Criticality can be defined in terms of two major factors:

- size of data – smaller data sizes are preferred for SPM.
- frequency of access – higher frequency is preferred for SPM.

A problem of this nature maps approximately to the well known *Knapsack Problem* in computer science [9], where we are given a knapsack of a fixed size, and can fill it with objects of different sizes and profit values. The objective is to select a subset of the objects that fit into the knapsack while maximizing the profit. In the SPM application of the knapsack problem, the SPM size is the equivalent of the knapsack size; object size corresponds to the data/array size; and the profit can be measured in terms of the number of accesses to the variables mapped into the scratch pad. This is because SPM access involves less energy dissipation than a cache access; this is a consequence of the SPM being a simple SRAM with no additional tag-related circuitry characterizing the cache, and hence, no associated dynamic power dissipation. In terms of performance, the guaranteed "hit" to the scratch pad ensures no cache-miss related delays.

Standard knapsack heuristics can also be applied in a straightforward manner to the SPM data allocation problem. The *profit density* metric, defined as $P_i/S_i$ characterizes each object in the knapsack problem in terms of the profit per unit size. The greedy heuristic fills the knapsack by objects in decreasing order of profit density, as long as the object size is smaller than the remaining knapsack space. The same approximate heuristic can also be used for SPM allocation, where we sort all arrays in terms of access frequency per unit array size, and consider arrays for SPM assignment in decreasing order. Scalar variables can be all stored in the SPM, as they may not amount to too much total space.

The above simple formulation can be used to obtain a reasonable SPM allocation of arrays, but several other factors can also be taken into account in a more comprehensive SPM allocation solution. First, several arrays can reuse the same SPM space because their *lifetimes* can be non-overlapping. Secondly, when there is a possibility of conflicts in the caches between different arrays accessed repeatedly, one of them could be diverted into the SPM to ensure good overall memory access behavior for both arrays. An example is shown in Fig. 4.28. Here, arrays $a$ and $b$ are accessed in a regular manner, whereas accesses to $c$ are data-dependent. Cache conflicts between arrays $a$ and $b$ could be avoided by suitably aligning the start positions of the arrays

```
for (i = 0; i < 100; i++) {
   b [i] = a [i] + 1; // regular access to 'a', 'b'
   c [b [i] ] = a [i]; // irregular access to 'c'
}
```

**Fig. 4.28** Arrays with irregular accesses could benefit from SPM allocation. Arrays $a$ and $b$ are accessed regularly, and if properly laid out, should exhibit good cache behavior. However, access to $c$ is irregular – not much locality might exist, and $c$ could benefit from SPM allocation

in memory. However, unpredictable and unavoidable cache conflicts with array *c* could occur. The conflicts could be avoided by assigning *c* to the SPM, where it is guaranteed to not interfere with the cache contents.

### *4.4.2 Dynamic Management of SPM*

We observe that the SPM allocation strategies in Section 4.4.1 assign an array to the SPM for the entire duration of its lifetime. This has some obvious disadvantages. An array may be occupying the SPM even if it is currently not being used, thereby precluding a different, more relevant, array from occupying the valuable space.

Figure 4.29(a) shows an example with arrays *a* and *b* accessed in two different loops, with *a* being allocated to the SPM, and *b* not allocated. This causes the second

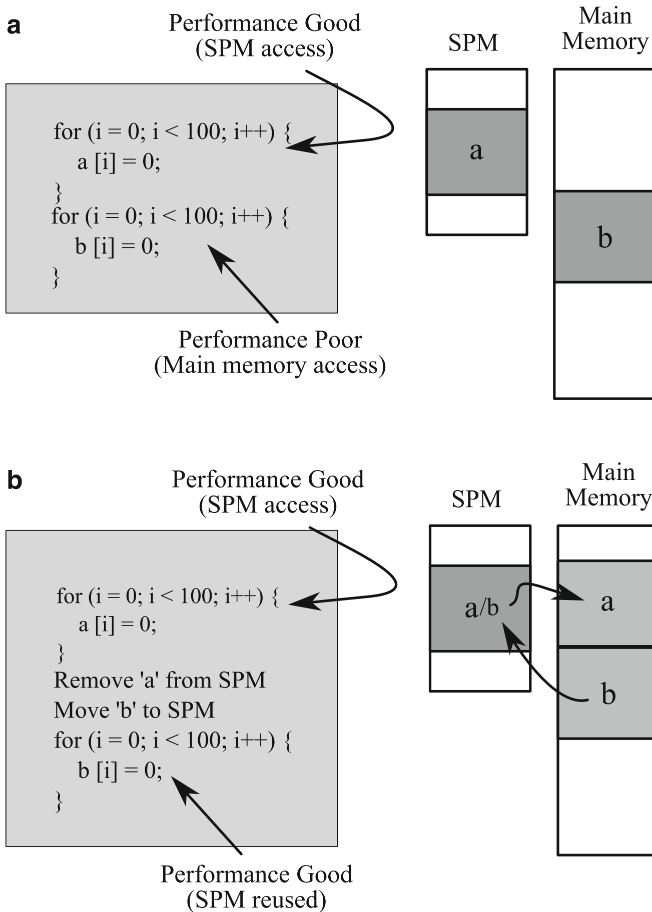



**Fig. 4.29** Space reuse in scratch pad memory. (**a**) Without reuse of SPM space, only one of *a*, *b* fits into SPM. The second loop performs poorly. (**b**) After the first loop, *a* is replaced in the SPM by *b*

loop to perform poorly. However, noticing that $a$ is not accessed in the second loop, we could substitute $a$ by $b$ before control enters the second loop (Fig. 4.29(b)). This calls for a more general strategy for identifying program points where we transfer data between the SPM and the background memory. The decision of whether to transfer an array to the SPM would depend on a comparison of the expected performance and energy improvement from fetching the data against the overhead incurred in doing so [37].

The next relaxation called for in the dynamic management of SPM is to permit a portion of an array to occupy space in the SPM. This allows us to assign SPM space to arrays that have heavy reuse but are larger than the SPM. One useful entity in this context is the idea of a *block* or *tile* – a portion of a multi-dimensional matrix that appears in the context of the *blocking* or *tiling* compiler transformation. Computations involving large matrices often show poor performance when the arrays are too large to fit into the data cache. Spatial and temporal locality exist in the computation, but capacity misses prevent data reuse. To overcome the problem, the arrays are divided into small blocks or tiles that fit into the cache, and the loops are appropriately restructured. This results in a significant performance improvement. Loop tiling is illustrated in Fig. 4.30 with a matrix multiplication example.

A similar conceptual transformation can be effected in the SPM. Arrays can be divided into tiles, moved into the SPM before being processed, and moved back later. The process is illustrated in Fig. 4.31 with the same matrix multiplication example. Array tiles are first transferred into the SPM with the READ_TILE routine. After processing, the tile $Z'$ is written back to memory with the WRITE_TILE routine [19].

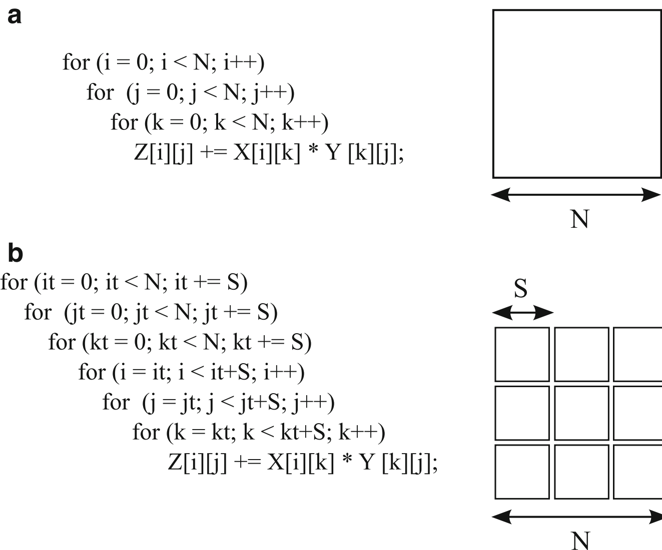
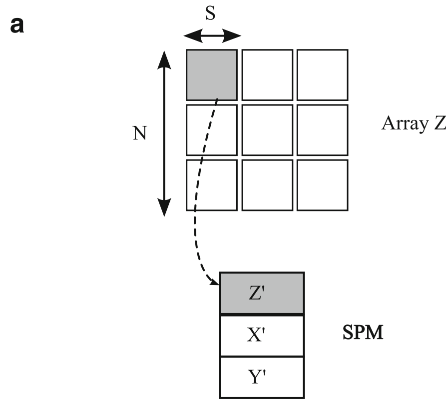



**Fig. 4.30** Blocking or tiling transformation. (**a**) Original loop: arrays might be too large to fit into on-chip memory. (**b**) Tiled loop: tile size is chosen so that it fits into on-chip memory




**b**

```
for (it = 0; it < N; it += S)
  for  (jt = 0; jt < N; jt += S)
    for (kt = 0; kt < N; kt += S) {
       READ_TILE Z[it...it+S-1, jt...jt+S-1] -> Z'[0...S-1][0...S-1]
       READ_TILE X[it...it+S-1, kt...kt+S-1] -> X'[0...S-1][0...S-1]
       READ_TILE Y[kt...kt+S-1, jt...jt+S-1] -> Y'[0...S-1][0...S-1]
       for (i = 0; i < S; i++)
         for  (j = 0; j < S; j++)
           for (k = 0; k < S; k++)
             Z'[i][j] += X'[i][k] * Y'[k][j];
       WRITE_TILE Z'[0...S][0...S] -> Z[it...it+S-1, jt...jt+S-1]
    }
```

**Fig. 4.31** Tiling transformation in SPM. (**a**) Tiles are first transferred to the SPM. (**b**) Updated ones are written back to main memory after processing

### *4.4.3 Storing both Instructions and Data in SPM*

An argument for scratch pad memory allocation can also be made in the context of instructions. Frequently executed instructions or basic blocks of instructions can be mapped to SPM so as to prevent the energy and performance-related overheads due to being evicted from the instruction cache. Power is saved both on account of the elimination of tag storage and access, and both performance and energy improves because of reduced cache misses. In fact, a unified formulation can use the same scratch pad memory to map either instructions or data [16, 35].

## 4.5 Memory Banking

The presence of multiple memory banks creates interesting optimization opportunities for the compiler. Traditionally a few DSP processors used a dual-bank on-chip memory architecture, but in modern systems, banking is used in various contexts for various objectives. In synchronous DRAMs (SDRAMs), banking is used to improve performance by keeping multiple data buffers from different banks ready for data

access. In application specific systems, dividing a monolithic memory into several banks leads to considerable performance improvement and power savings. The performance improvement comes from the ability to simultaneously access multiple data words, while the power savings arise from smaller addressing circuitry, word lines, and bit lines, as observed earlier.

The power optimization problem is to assign data to memory banks in order to minimize certain objective functions. In terms of performance, we would like to be able to simultaneously access data in different banks so that computation time decreases, assuming multiple datapath resources are available. In terms of power dissipation, we have the possibility of moving specific banks to *sleep mode* during periods of inactivity.

Figure 4.32 illustrates the memory bank assignment problem with a simple example. In the schedule shown in Fig. 4.32(a), nodes in dark, labeled $A, B$, etc., represent memory load and store nodes, and result in memory accesses. $M1$ and $M2$ are the two memory banks. Initially, variables $A$, $C$, $E$, $F$, and $G$ are assigned to bank $M1$. Variables $B$ and $D$ are assigned to $M2$. Assume that the memory banks can be either in *active* mode or *sleep* mode. Further, assume that the memory can transition from active to sleep state instantaneously, whereas it requires one cycle to transition from sleep to active mode (i.e., one cycle has to be spent in *wake-up* mode during this transition). The power dissipation during active and wake-up modes is high, and the power during sleep mode is low. The schedule in Fig. 4.32(a) does not permit any transition to sleep modes due to the lack of sufficiently long periods of inactivity. In order to exploit the sleep mode without compromising the schedule length, we can alter the bank assignment. Let us interchange the bank assignment of $C$ and $D$, which results in an alternative schedule shown in Fig. 4.32(b). We notice that now bank $M2$ is inactive for a sufficiently long time, permitting us to move it to sleep mode for two cycles, before returning to active mode via one cycle in wake-up mode [25].

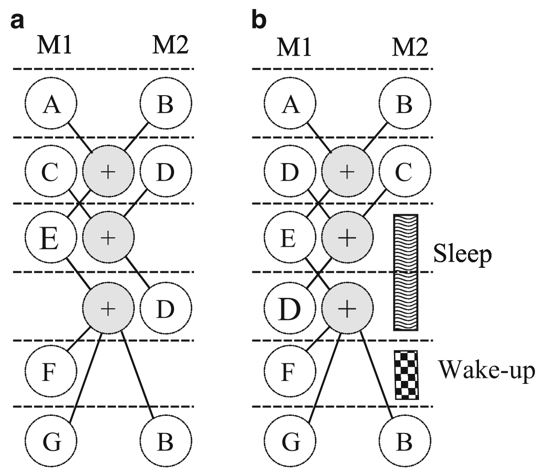



**Fig. 4.32** Bank assignment targeting power efficiency. Transition of a bank from *active* to *sleep* state is instantaneous, while transition back requires one cycle. (**a**) Bank assignment affords no possibility to move either bank into sleep state. (**b**) Variables re-assigned to permit $M2$ to be moved to sleep state for two cycles

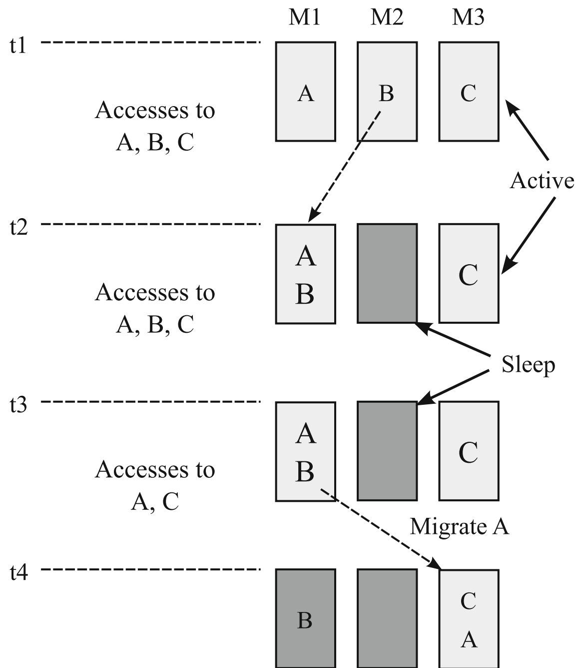



**Fig. 4.33** Data migration. At $t2$, $B$ is moved to $M1$, permitting us to move $M2$ to sleep mode. $B$ is not accessed between $t3$ and $t4$. At $t4$, $A$ is moved to $M3$, permitting us to move $M1$ to sleep mode

Often, the presence of conditionals prevents a proper static analysis of decisions relating to the setting of memory banks to sleep mode for saving power. Dynamic approaches involving the run-time *migration* of data to different memory banks may be needed. We can keep track of variable- and bank-referencing patterns, and can attempt to cluster recently referenced variables into the same bank, thereby creating an opportunity to maximize the number of inactive banks and move then to sleep mode.

Figure 4.33 illustrates the basic idea of a data migration strategy. Initially, at time $t1$, three arrays A, B, and C are stored in three memory banks M1, M2, and M3. Our monitoring hardware detects accesses to both A and B between times $t1$ and $t2$, enabling the migration of array B to module M1. Assume that M1 is large enough to accommodate A and B, but is not large enough to accommodate all three arrays. With this migration, module M2 can be set to sleep mode. Between $t2$ and $t3$, we detect accesses to all of A, B, and C, so we just retain the current modes. Between $t3$ and $t4$, we detect accesses to A and C. This leads us to migrate A to module M3, causing both M1 and M2 to be set to sleep mode.

In order to ensure the effectiveness of the data migration approach, some issues that need to be addressed are:

- the delay and energy overhead of the data being migrated needs to be accounted for; large array variables can result in large overheads.
- data migration needs to account for the sizes of the memory banks.

- additional hardware is needed to keep track of temporal correlation of different variables. Since this can lead to a significant power overhead if done every cycle, a sample-based approach may be necessary [6].

## 4.6 Memory Customization

One of the most important characteristics of embedded SoCs is that the hardware architecture can be customized for the specific application or set of applications that the system will be used on. This customization can help improve area, performance, and power efficiency of the implementation for the given application. Even when the overall platform may be fixed, different subsystems can be independently tailored to suit the requirements of the application. The memory subsystem is a fertile ground for such customization. Cache memory can be organized in many different ways by varying a large number of parameters: cache size, line size, associativity, write policy, replacement policy, etc. Since the number of possible configurations is large, an explicitly simulation-based selection strategy may be too time-consuming. A static inspection and analysis of the application can reveal several characteristics that help determine the impact of different parameter values without actual execution or simulation.

Data caches are designed to have cache lines consisting of multiple words in anticipation of spatial locality of memory accesses. How large should the cache line be? There is a trade-off in sizing the cache line. If the memory accesses are very regular and consecutive, i.e., exhibit good spatial locality, a longer cache line is desirable, since it saves power and improves performance by minimizing the number of off-chip accesses and exploits the locality by pre-fetching elements that will be needed in the immediate future. On the other hand, if the memory accesses are irregular, or have large strides, a shorter cache line is desirable, as this reduces off-chip memory traffic by not bringing unnecessary data into the cache. An estimation mechanism could be used to predict the impact of different data cache line sizes based on a compiler analysis of array access patterns. The cache line size is bounded from above by the DRAM burst size, which represents the maximum number of data words that can be brought into the cache with a single burst access.

A given amount of on-chip data memory space could be divided in various ways into data cache and scratch pad memory. An associated memory customization problem is to determine the best division of the space. The division that results in the least off-chip memory accesses would again maximize performance as well as minimize power. Figure 4.34 shows a typical variation of the total number of off-chip memory accesses with increasing data cache size ($D$), with the total on-chip memory fixed to a constant $T$. Thus, the choice of a larger cache size $D$ results in a correspondingly smaller scratch pad memory size $T - D$. We note that when the cache size is too small or too large, the number of memory accesses is relatively higher. When the cache size is too small, it is essentially ineffective due to serious capacity misses. When the cache is large, occupying all of the on-chip memory, then there is no

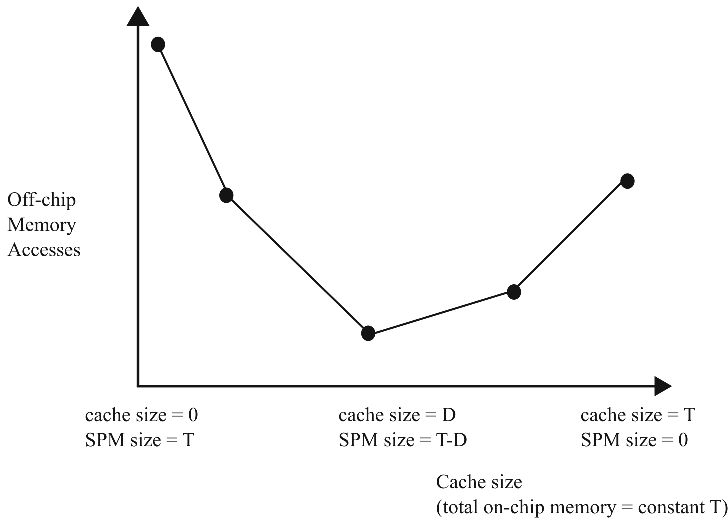



**Fig. 4.34** Division of on-chip memory space T into cache and scratch pad. When cache is too small, latency increases because of capacity misses. When cache is too large, latency also increases because there is no room for effective SPM utilization

room for scratch pad memory, thereby losing the advantages of SPM. The optimal on-chip memory configuration lies somewhere in between the two extremes, with some space devoted to both data cache and SPM, augmented by an intelligent compiler strategy that allocates data to the two components.

Other than the cache vs. scratch pad trade-off above, the total on-chip memory space allocated to the application can also be a variable, to be customized depending on the requirements of the application. In general, data cache performance, as measured by hit rate, improves with increasing cache size. Similarly, increasing scratch pad memory performance also leads to higher performance. A memory exploration loop can vary the total on-chip memory space, and study the performance variation, as depicted in Fig. 4.35. Each data point (A, B, C, D, and E) could represent the best result obtained from a finer-grain exploration of scratch pad vs. cache partition for a fixed total on-chip memory size [30]. Figure 4.35 shows that the performance, as measured by hit rate (or, equivalently, in terms of total off-chip memory traffic), improves with increasing on-chip memory size, but tapers off beyond a certain point. Design points such as C in Fig. 4.35 are strong candidates, since they represent the boundary beyond which it may not be worth increasing the memory size – for higher sizes, the hit-rate improves marginally, but the resulting larger memory size represents an overhead in both memory access time and energy.

Embedded SoCs allow the possibility of customized memory architectures that are tailored to reducing power for a specific application. Figure 4.36 shows an example of such an instance. A simple loop is shown in Fig. 4.36(a), with two arrays *a* and *b* accessed as shown. A default computation and memory architecture for implementing this system is shown in Fig. 4.36(b). The computation is performed in

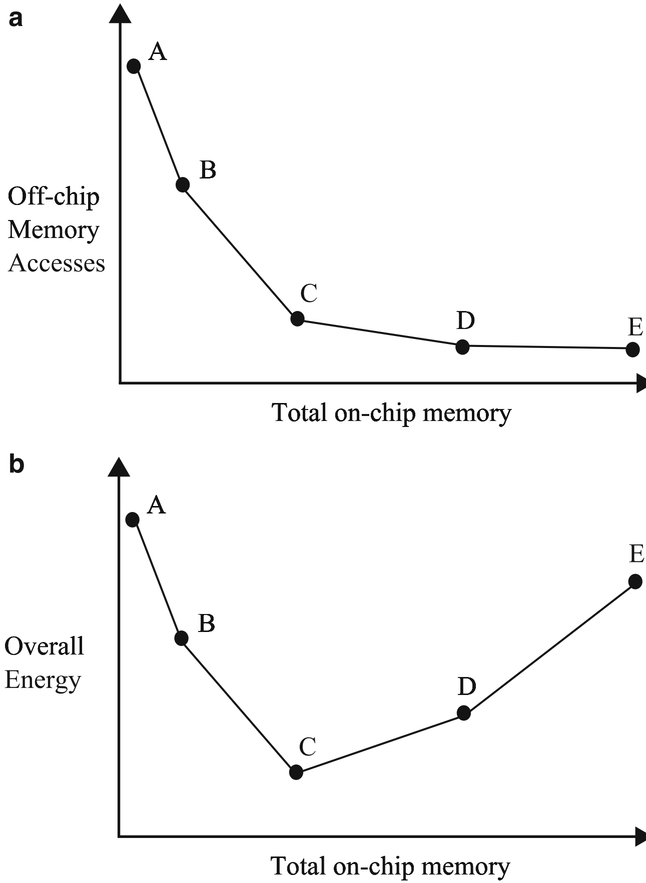



**Fig. 4.35** (**a**) Off-chip memory access count decreases with increasing on-chip memory size. Beyond point $D$, performance does not improve (but power continues increasing). (**b**) Energy decreases with increasing on-chip memory, but the trend reverses for larger on-chip memory because the per-access energy of larger memories is greater

the *proc* block, and arrays $a$ and $b$ are stored in memory module *Mem* of size $N$ words. Assuming there is no cache structure here, every array reference results in an access to *Mem* block, resulting in the standard power dissipation associated with reading or writing of an $N$-word memory.

A more careful analysis of the array reference patterns reveals some optimization possibilities [29]. Figure 4.36(c) shows the elements of $a$ that are accessed in one iteration of the $j$-loop, with $i = 5$ and $j = 2$. We assume that $a$ is an $8 \times 8$ array and $L = 4$. In Fig. 4.36(d), we show the elements of $a$ accessed in the next $j$-iteration, i.e., $i = 5$, $j = 3$. We notice an overlap of three array elements, indicating a significant data reuse resulting from temporal locality of data access. In general, out of the $L$ array elements accessed in the innermost loop, we have already accessed $L - 1$ elements in the previous $j$ iteration. A suggested power optimization here is that,

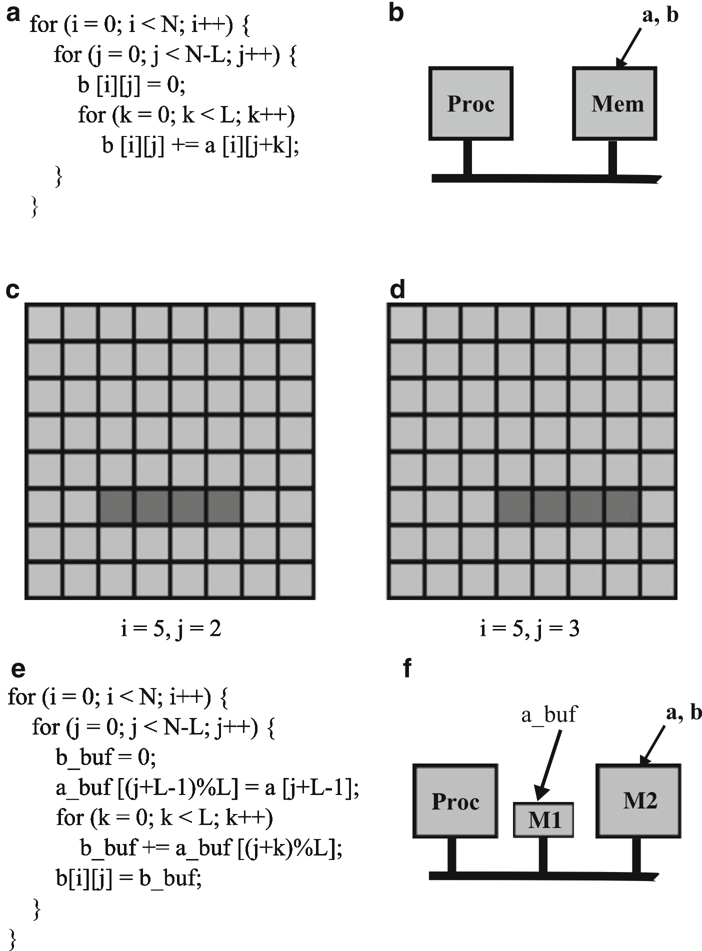



**Fig. 4.36** Memory Customization Example (**a**) Original Loop (**b**) Default memory architecture (**c**) 'a' elements accessed in inner loop for i = 5, j = 2 (**d**) 'a' elements accessed in inner loop for i = 5, j = 3 (**e**) Modified loop (**f**) Customized memory architecture. Only the relatively smaller $M1$ is accessed in the inner loop, saving power

we can instantiate a small buffer *a_buf* of size $L$ words which would store the last $L$ elements of $a$ accessed in the previous $j$-iteration. Since $L$ is much smaller than $N$, we would be accessing data only from the much smaller $M1$ memory module shown in Fig. 4.36(f) in the innermost loop. Since the energy cost for access from $M1$ is expected to be much smaller than that due to access from *Mem*, we can expect a significant energy saving for the entire loop nest. The modified loop is shown in Fig. 4.36(e).

An explicit physical partitioning of the logical memory space could also be performed based on the dynamic profile of memory address references. This could be useful when a static analysis of the array references is difficult either due to complex conditionals in the specification, or due to data dependence. The frequency of memory references for an application is generally not uniformly distributed over the address space – certain parts of the memory are likely to be more heavily accessed than others. Figure 4.37(a) shows an example of a memory access frequency distribution over the address space. The accesses could be logically grouped into three windows of different characteristics – the first 256 addresses have a relatively high access frequency; the next 2048 elements have a low access frequency; and the final 1024 elements have a medium frequency. Such variations in access distributions could occur in typical code because different data arrays are accessed in different ways. Based on the above grouping, the memory space could be partitioned into three physical modules. Figure 4.37(b) shows a default memory architecture in which all memory accesses are made to one large memory module storing the entire address space. This could, however, lead to unnecessarily high memory access energy because every access would be made to the large memory. Figure 4.37(c)

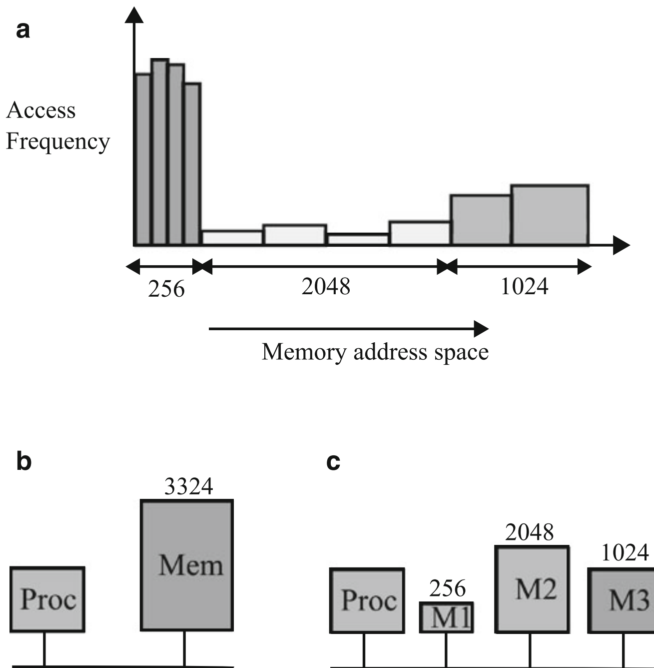



**Fig. 4.37** Memory Partitioning. (**a**) Distribution of access frequency over memory address space. (**b**) Default memory architecture. (**c**) Partitioned memory. The most frequent accesses are made to the smaller $M1$, saving power

shows an example memory partitioning with the three logical address ranges identified above mapped to three separate physical modules. This partitioning ensures that the high frequency accesses are restricted to the relatively smaller memory module, thereby leading to a significant energy saving over the default architecture.

## 4.7 Reducing Address Bus Switching

The memory address bus is typically long because of several reasons. The memory core might have been separately obtained and instantiated, and it may not be possible to physically place it next to the address generation logic. The memory may be serving several units which are independently generating addresses and accessing data. Hence, bits switching on the address lines lead to significant power dissipation because of the large capacitance associated with each line. This provides an important power optimization opportunity – power can be reduced by reducing the total number of bits switching on the memory address lines. This reduction can be effected by two broad approaches: (i) encoding the address lines; and (ii) transforming the data layout.

### *4.7.1 Encoding*

The sequence of bits appearing over the memory address bus can be changed by, in general, inserting an *encoder* close to the address generation block, and a *decoder* near the memory in such a way that the same addresses are generated and seen by the memory as before, but the sequence of signals appearing at the address bus is modified. The difference is illustrated in Fig. 4.38(a) and (b). The encoder and

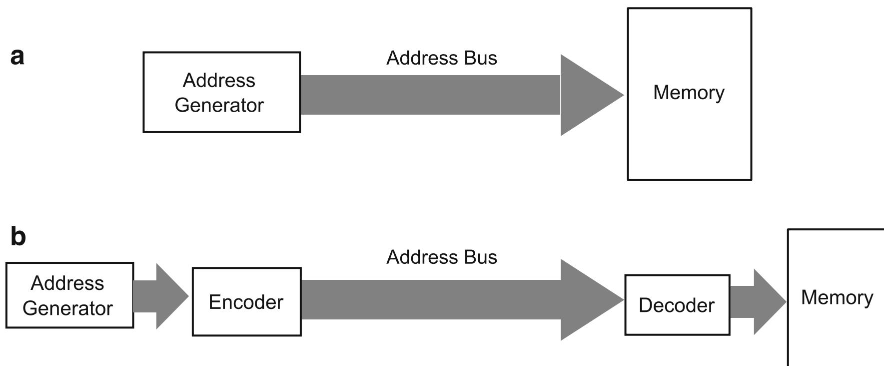



**Fig. 4.38** Address Bus Encoding. (**a**) Original architecture (**b**) Encoder and decoder for address bus. The objective of encoding is to have lower activity on the encoded address bus at the expense of a small encoding and decoding power

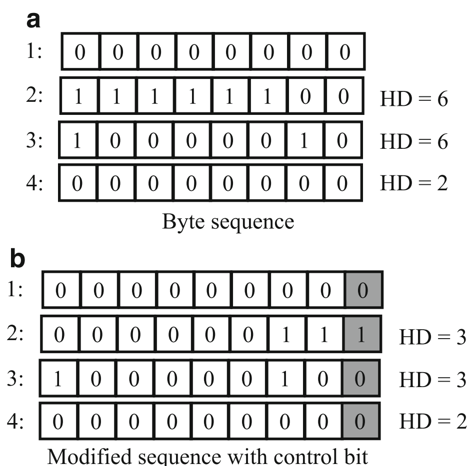



**Fig. 4.39** Bus Invert Coding. (**a**) Original sequence – total bits switching = 14 (**b**) Modified sequence – control bit = 1 indicates the data bits should be inverted. Total bits switching = 8

decoder logic blocks incur additional power overhead, but the power saved by reduced switching on the high-capacitance memory address lines is expected to be much larger. We study two very simple and effective encoding techniques in this section.

The *Bus-invert* encoding technique attempts to minimize the Hamming Distance between successive words appearing on the address bus [34]. An extra bit is added to the memory bus, indicating to the decoder whether or not the address word should be bitwise inverted. The choice of whether to invert or not is made depending on which option reduces the hamming distance between the current and previous words. This is illustrated in Fig. 4.39, with four successive addresses appearing on an 8-bit address bus. The four successive address values appearing on the address bus have hamming distances 6, 6, and 2, making a total of 14 bits switching. The bus-invert encoding mechanism adds an auxiliary bit to the address bus (shown in grey), making the width 9 bits. Noticing that 6 out of the 8 bits have switched in the second word, we decide to complement the address bits in the second word. The fact that the bits are inverted is transmitted by making the auxiliary bit '1'. This causes a total hamming distance of 3 between the first two words (two bits in the address word, and one due to the auxiliary bit). Comparing the second transmitted word with the third word, we notice a hamming distance of just 2, so the word is sent as is, with the auxiliary bit set to '0' (indicating that the word is normal, not complemented). Thus, whenever the number of bits switching is more than half the bus width, we can send the complemented bits, thereby ensuring that no more than half the bits in the bus will change from one transaction to the next. The total number of bits switching in the encoded bus is 8, as opposed to 14 in the original bus. The encoding is a general mechanism and is not address-bus specific.

An encoding scheme that is specifically tailored to the typical behavior of memory address buses is the *T0* encoding. It exploits the general observation that often, the address sequence generated on the instruction memory address bus of

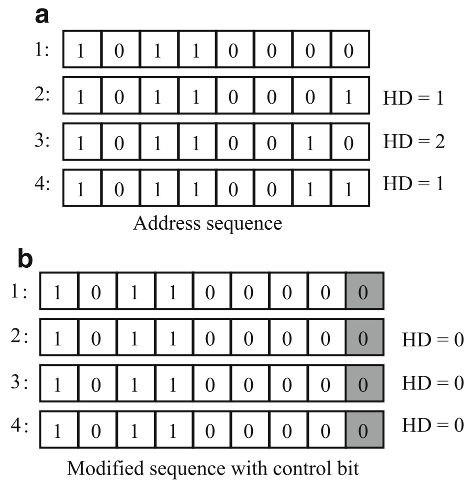



**Fig. 4.40** T0 Encoding. (**a**) Original sequence – total bits switching = 4 (**b**) Modified sequence – control bit = 1 indicates that the previous value should be incremented. Total bits switching = 0

a processor tends to have consecutive values because of spatial locality of reference in the instruction stream. The T0 code adds an extra line to the bus, which is used to indicate whether the next address is consecutive and is generated by incrementing the previous one [4]. This is illustrated in Fig. 4.40. The original sequence of four address has consecutive values, with the hamming distances between words being 1, 2, and 1, giving a total of 4 bits switching for the sequence (Fig. 4.40(a)). In T0 encoding the second address repeats the first, with the extra control bit being '0', indicating that the the previous address should be incremented to generate the new one. This continues for the remaining words, with the decoder expected to increment the previous address to generate the new one as long as the control bit is '0'. This scheme may lead to near zero transitions in the steady state when there is a sequence of consecutive addresses. In addition to short consecutive sequences occurring in the instruction address sequence, this also happens during array data accesses in loops.

### *4.7.2 Data Layout*

An orthogonal transformation to address encoding is to rearrange the data layout in memory so that the generated address sequence has lower switching when the data is accessed. Such layout transformations apply more to data memory than instructions.

Figure 4.41 shows an example of a simple data layout transformation that helps bring down the memory address bus switching. An example code is shown in Fig. 4.41(a), and its corresponding data memory access sequence is shown in Fig. 4.41(b). We notice that the memory address alternates between two regions of the memory corresponding to arrays $a$ and $b$. This usually results in a large number of address bits switching between every consecutive pair of address words (with

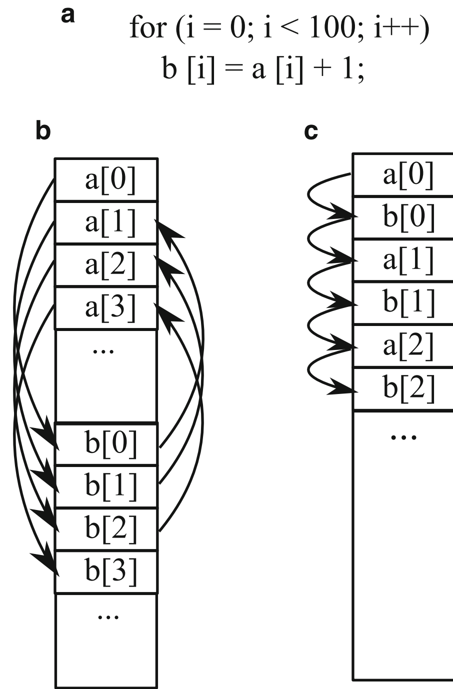



**Fig. 4.41** Data Interleaving. (**a**) Behavior (**b**) Original address sequence (**c**) Address sequence when arrays are interleaved. The interleaved address sequence usually has lower total switching, and is amenable to other encoding strategies such as gray code

the exception of the specific case where the corresponding data elements of equal width are separated by an exact power of two, in which case only one bit would be flipping). Since the sequence is deterministic, we can perform a simple transformation of interleaving the elements of the two arrays, as shown in Fig. 4.41(c). This causes the address sequence to be consecutive, which is much better behaved in terms of bit switching, and can then be exploited by other encoding and decoding mechanisms such as Gray Code or T0.

A more complex data transformation is shown in Fig. 4.42. In the example of Fig. 4.42(a), the two-dimensional array *a* is accessed in a *tiled* pattern visually depicted in Fig. 4.42(b). Using the standard storage conventions of row-major and column-major storage for multi-dimensional arrays, we see that the address sequence incurs large hops even within a tile, since we have more than one row and column accessed in each tile. The sequence for row-major storage is shown in Fig. 4.42(c). Again, the predictability of the behavioral array reference pattern can be used to use a more custom *tile-based* mapping to store array data in memory. Figure 4.42(d) shows the tiles laid out in memory in a manner that avoids the large hops in the memory address bus.

Detailed discussions on encoding and data transformations for reducing address bus power can be found in [4, 29]. Before deciding on the applicability of such encoding and transformations to a specific system design scenario, it is important to perform a careful cost-benefit analysis.

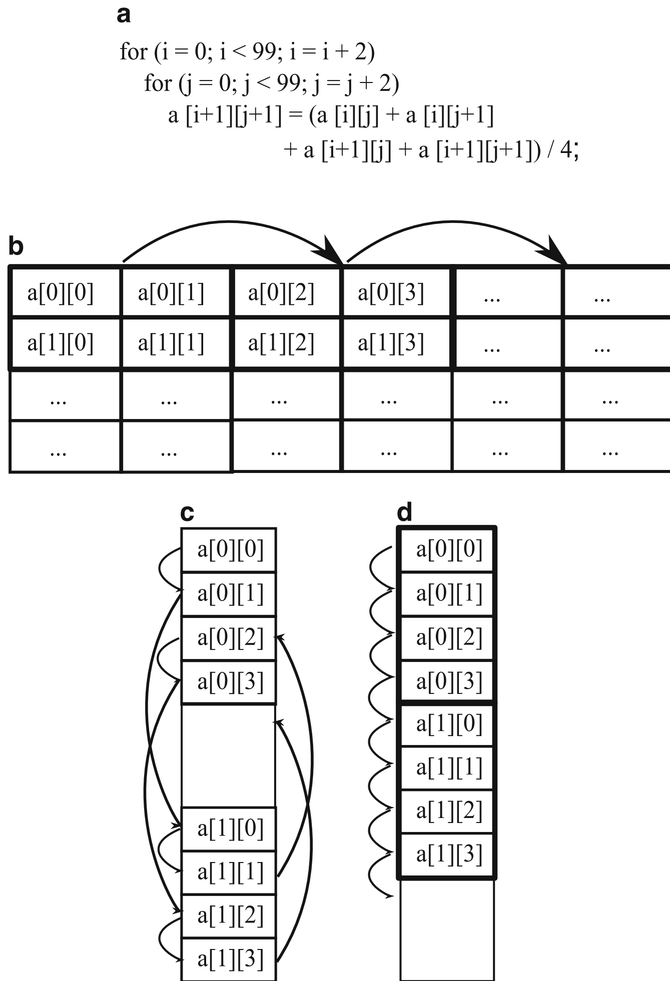



**Fig. 4.42** Tile-based data storage. (**a**) Behavior (**b**) Tiled memory access pattern (**c**) Address sequence using row-major storage (**d**) Address sequence using tile-based storage. Access pattern is more regular and shows lower total switching in tile-based storage

- The address bus encoding decision would depend on the actual address bus lengths in the system. If the on-chip bus has a relatively small length, the area and power overhead incurred in encoding and decoding circuitry may not be worthwhile.
- The actual memory architecture and address protocol may also influence the applicability of such optimizations, a good example being a DRAM. In DRAMs, the actual address is time-multiplexed into row and column addresses, using half the number of address bits. This obviously requires changes to the simple encoding techniques discussed above based on hamming-distance between complete

addresses – here, we need to worry about the hamming distance between the row- and column address as well.
- Address bus hamming distance is not the only metric of importance in determining total energy. It is possible that a hamming distance of 1 refers to a physically distant memory location if the switch happens in the higher order address bits. This may cause a switch in the *DRAM memory page*, which will lead to additional power costs in the DRAM and associated changes in the access protocol FSM that should be accounted for.
- A similar cost analysis has to be performed for data transformations targeting the memory address bus. Data transformations are more global in their effect – they go beyond the specific loop for which the transformation was derived. Transformations to the same arrays in different loops might be conflicting – they may be useful in one and harmful in another. Data transformations need to be performed after analyzing their effect on all sections of code where the data is accessed.

## 4.8 DRAM Power Optimization

As the amount of memory in computer systems continues to grow, the off-chip memory – the DRAM sub-system – increasingly consumes larger amounts of power, even though the CPU is designed against a tight power budget. Newer generations of DRAMs have exposed power management controls so that external controllers can implement aggressive optimization policies to reduce DRAM power. The most common such feature has been the introduction of low power modes in different DRAM states. These could be one of the following:

- *Active:* The DRAM could be moved to an *Active Power-down* mode. In this "shallow" low-power mode, one or more DRAM banks is open, and it is relatively fastest to bring the DRAM back to accessing data from this mode.
- *Precharge:* In the *Precharge Power-down* mode, all banks are closed, and it takes more time to return to the active state where we can access data again. Power consumption is lower than in active power-down.
- *Self-refresh:* In the *Self-refresh Power-down* mode, the power consumption is the lowest, and it takes the maximum time to recover to the active state.

The transitions to and from the above DRAM low power states is expected to be explicitly performed by the memory controller. The architectural details and programming possibilities continue to evolve with newer generation of DRAMs.

The above power reduction opportunities have led to attention being focussed on optimizations through various mechanisms – starting from the level of memory controllers that directly interface to DRAMs, all the way to page allocation policies in operating systems. DRAM power management functions can be performed by many different entities: the DRAM controller can use prediction techniques; the compiler can analyze the source code and insert explicit instructions to change DRAM power state; and the operating system scheduler can also monitor activity on the DRAM

and make the management decisions. Since the operating system is able to simultaneously monitor activities of different processes, it may discover reference patterns not visible to the compiler.

It is obvious that some variant of the standard predictive shutdown policy generally useful in several other low power controller strategies – predict the future idle time based on past behavior, and switch to low power mode if there is an expectation of power saving – is also applicable in the DRAM context (see Section 5.1.2 for an application of the same principle to voltage scaling decisions based on past CPU utilization). Basically, the controller's policy should ensure that the overhead of transitioning to and from the low power state is less than the expected power saving [7]. The choice of the controller policy would depend on the amount of performance penalty we are willing to tolerate. This is achieved by setting specific threshold parameters for every power state of the DRAM. If the chip has not been accessed for a time period beyond the threshold, then we can move the DRAM to the next low power state [24].

The data migration policy discussed in Section 4.5 – where data objects are grouped into a smaller number of memory modules so that others can be set to sleep mode – is an example of a high-level data organization concept that also applies to DRAM. The default page allocation of the operating system, which effectively does a random assignment of memory pages across the different memory chips present in the system, can be made power aware by exploiting the same data consolidation idea introduced in Section 4.5 – frequently accessed memory pages are migrated to a common memory chip so as to improve the possibility of power saving through moving unused chips to low power mode [24].

Finally, attempts can be made to bring successive DRAM accesses closer in time so that idle periods can be made artificially longer. Bringing two DRAM accesses together is usually achieved by delaying the first access, which may incur a performance penalty.

## 4.9 Summary

The storage and retrieval of large amounts of data and instructions in modern electronic and computer systems make the memory subsystem an important target component for power optimization. Memory related power reduction spans a broad spectrum across several levels of abstraction. Circuit and architecture level opportunities have been identified for trading off a small performance penalty for significant savings in power dissipation in processor caches. Scratch pad memories help reduce system power by avoiding expensive tag lookups associated with caches. Since SoCs permit flexible on-chip memory architectures, estimation based exploration can help determine the best memory configurations for an application scenario. Techniques such as encoding and data layout transformation could be used to reduce power dissipation on high-capacitance memory address buses.

The chapter covered some of the basic techniques in each of the categories described above. The reader is encouraged to look up the references indicated at various places in the chapter to obtain an idea of the more advanced proposals in the memory power optimization field. The area continues to draw significant interest from researchers as systems move in the direction of higher complexity. For example, as next generation DRAMs evolve to include more power controls, we can expect associated power optimization proposals to exploit them. Advanced transformations such as compression also help reduce power by reducing off-chip memory traffic and providing similar performance with smaller memory and cache sizes.

## References


1. Bajwa, R.S., Hiraki, M., Kojima, H., Gorny, D.J., Nitta, K., Shridhar, A., Seki, K., Sasaki, K.: Instruction buffering to reduce power in processors for signal processing. IEEE Transactions on VLSI Systems **5**(4), 417–424 (1997)
2. Bellas, N., Hajj, I., Polychronopoulos, C.: Using dynamic cache management techniques to reduce energy in a high-performance processor. In: International symposium on low power electronics and design, pp. 64–69. San Diego, USA (1999)
3. Bellas, N., Hajj, I.N., Polychronopoulos, C.D., Stamoulis, G.: Architectural and compiler techniques for energy reduction in high-performance microprocessors. IEEE Transactions on VLSI Systems **8**(3), 317–326 (2000)
4. Benini, L., Micheli, G.D.: System level power optimization: Techniques and tools. ACM Transactions on Design Automation of Electronic Systems **5**(2), 115–192 (2000)
5. Chang, Y.J.: An ultra low-power tlb design. In: DATE '06: Proceedings of the conference on Design, automation and test in Europe, pp. 1122–1127. European Design and Automation Association, 3001 Leuven, Belgium, Belgium (2006)
6. Delaluz, V., Sivasubramaniam, A., Kandemir, M., Vijaykrishnan, N., Irwin, M.J.: Scheduler-based dram energy management. In: Design Automation Conference. New Orleans, USA (2002)
7. Fan, X., Ellis, C., Lebeck, A.: Memory controller policies for dram power management. In: International symposium on low power electronics and design, pp. 129–134. Huntington Beach, USA (2001)
8. Flautner, K., Kim, N.S., Martin, S., Blaauw, D., Mudge, T.: Drowsy caches: simple techniques for reducing leakage power. In: International symposium on computer architecture, pp. 240–251. Anchorage, USA (2002)
9. Garey, M.R., Johnson, D.S.: Computers and Intractibility – A Guide to the Theory of NP-Completeness. W.H. Freeman (1979)
10. Ghose, K., Kamble, M.B.: Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In: International symposium on low power electronics and design, pp. 70–75. San Diego, USA (1999)
11. Hasegawa, A., Kawasaki, I., Yamada, K., Yoshioka, S., Kawasaki, S., Biswas, P.: SH3: High code density, low power. IEEE Micro **15**(6), 11–19 (1995)
12. Hennessy, J.L., Patterson, D.A.: Computer Architecture – A Quantitative Approach. Morgan Kaufman, San Francisco, CA (1994)
13. Hu, J.S., Nadgir, A., Vijaykrishnan, N., Irwin, M.J., Kandemir, M.: Exploiting program hotspots and code sequentiality for instruction cache leakage management. In: International symposium on low power electronics and design, pp. 402–407. Seoul, Korea (2003)
14. Inoue, K., Ishihara, T., Murakami, K.: Way-predicting set-associative cache for high performance and low energy consumption. In: International symposium on low power electronics and design, pp. 273–275. San Diego, USA (1999)

15. Inoue, K., Moshnyaga, V.G., Murakami, K.: A history-based I-cache for low-energy multimedia applications. In: International symposium on low power electronics and design, pp. 148–153. Monterey, USA (2002)
16. Janapsatya, A., Parameswaran, S., Ignjatovic, A.: Hardware/software managed scratchpad memory for embedded systems. In: Proceedings of the IEEE/ACM International Conference on Computer Aided Design (2004)
17. Juan, T., Lang, T., Navarro, J.J.: Reducing tlb power requirements. In: ISLPED '97: Proceedings of the 1997 international symposium on Low power electronics and design, pp. 196–201. ACM, New York, NY, USA (1997). DOI http://doi.acm.org/10.1145/263272.263332
18. Kadayif, I., Sivasubramaniam, A., Kandemir, M., Kandiraju, G., Chen, G.: Generating physical addresses directly for saving instruction tlb energy. In: MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture, pp. 185–196. IEEE Computer Society Press, Los Alamitos, CA, USA (2002)
19. Kandemir, M., Ramanujam, J., Irwin, M.J., Vijaykrishnan, N., Kadayif, I., Parikh, A.: Dynamic management of scratch-pad memory space. In: ACM/IEEE Design Automation Conference, pp. 690–695 (2001)
20. Kaxiras, S., Hu, Z., Martonosi, M.: Cache decay: exploiting generational behavior to reduce cache leakage power. In: International symposium on computer architecture, pp. 240–251. Goteberg, Sweden (2001)
21. Kim, S., Vijaykrishnan, N., Kandemir, M., Sivasubramaniam, A., Irwin, M.J., Geethanjali, E.: Power-aware partitioned cache architectures. In: International symposium on low power electronics and design, pp. 64–67. Huntington Beach, USA (2001)
22. Kin, J., Gupta, M., Mangione-Smith, W.H.: The filter cache: an energy efficient memory structure. In: International symposium on microarchitecture, pp. 184–193. Research Triangle Park, USA (1997)
23. Ko, U., Balsara, P.T., Nanda, A.K.: Energy optimization of multi-level processor cache architectures. In: International symposium on low power design, pp. 45–49. New York, USA (1995)
24. Lebeck, A.R., Fan, X., Zeng, H., Ellis, C.: Power aware page allocation. SIGOPS Oper. Syst. Rev. **34**(5), 105–116 (2000). DOI http://doi.acm.org/10.1145/384264.379007
25. Lyuh, C.G., Kim, T.: Memory access scheduling and binding considering energy minimization in multi-bank memory systems. In: Design automation conference, pp. 81–86. San Diego, USA (2004)
26. Malik, A., Moyer, B., Cermak, D.: A low power unified cache architecture providing power and performance flexibility. In: International symposium on low power electronics and design, pp. 241–243. Rapallo, Italy (2000)
27. Manne, S., Klauser, A., Grunwald, D.C., Somenzi, F., Somenzi, F.: Low power tlb design for high performance microprocessors. Tech. rep., University of Colorado (1997)
28. Min, R., Jone, W.B., Hu, Y.: Location cache: A low-power l2 cache system. In: International symposium on low power electronics and design, pp. 120–125. Newport Beach, USA (2004)
29. Panda, P.R., Catthoor, F., Dutt, N.D., Danckaert, K., Brockmeyer, E., Kulkarni, C., Vandercappelle, A., Kjeldsberg, P.G.: Data and memory optimization techniques for embedded systems. ACM Transactions on Design Automation of Electronic Systems **6**(2), 149–206 (2001)
30. Panda, P.R., Dutt, N.D., Nicolau, A.: Local memory exploration and optimization in embedded systems. IEEE Transactions on Computer Aided Design **18**(1), 3–13 (1999)
31. Panda, P.R., Dutt, N.D., Nicolau, A.: Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration. Kluwer Academic Publishers, Norwell, MA (1999)
32. Panda, P.R., Dutt, N.D., Nicolau, A.: On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. ACM Transactions on Design Automation of Electronic Systems **5**(3), 682–704 (2000)
33. Panwar, R., Rennels, D.: Reducing the frequency of tag compares for low power i-cache design. In: International symposium on low power design, pp. 57–62. New York, USA (1995)
34. Stan, M.R., Burleson, W.P.: Bus-invert coding for low power I/O. IEEE Transactions on VLSI Systems **3**(1), 49–58 (1995)

35. Steinke, S., Wehmeyer, L., Lee, B., Marwedel, P.: Assigning program and data objects to scratchpad for energy reduction. In: Design Automation and Test in Europe, pp. 409–417. Paris, France (2002)
36. Su, C.L., Despain, A.M.: Cache design trade-offs for power and performance optimization: a case study. In: International Symposium on Low Power Design, pp. 63–68. New York, NY (1995)
37. Udayakumaran, S., Dominguez, A., Barua, R.: Dynamic allocation for scratch-pad memory using compile-time decisions. ACM Transactions on Embedded Computing Systems **5**(2), 472–511 (2006)
38. Zhang, C., Vahid, F., Yang, J., Najjar, W.: A way-halting cache for low-energy high-performance systems. In: International symposium on low power electronics and design, pp. 126–131. Newport Beach, USA (2004)
39. Zhang, W., Hu, J.S., Degalahal, V., Kandemir, M., Vijaykrishnan, N., Irwin, M.J.: Compiler directed instruction cache leakage optimization. In: International symposium on microarchitecture, pp. 208–218. Istanbul, Turkey (2002)