Memory Architectures

13

Preeti Ranjan Panda

Abstract

In this chapter we discuss the topic of memory organization in embedded systems and Systems-on-Chips (SoCs). We start with the simplest hardware-based systems needing registers for storage and proceed to hardware/software codesigned systems with several standard structures such as Static Random-Access Memory (SRAM) and Dynamic Random-Access Memory (DRAM). In the process, we touch upon concepts such as caches and Scratchpad Memories (SPMs). In general, the emphasis is on concepts that are more generally found in SoCs and less on general-purpose computing systems, although this distinction is not very clearly defined with respect to the memory subsystem. We touch upon implementations of these ideas in modern research and commercial scenarios. In this chapter, we also point out issues arising in the context of the memory architectures that become exported as problems to be addressed by the compiler and system designer.

Acronyms

ALU	Arithmetic-Logic Unit
ARM	Advanced Risc Machines
CGRA	Coarse Grained Reconfigurable Architecture
CPU	Central Processing Unit
DMA	Direct Memory Access
DRAM	Dynamic Random-Access Memory
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
HDL	Hardware Description Language
PCM	Phase Change Memory

P.R. Panda (✉)
Department of Computer Science and Engineering, Indian Institute of Technology Delhi, New Delhi, India
e-mail: panda@cse.iitd.ac.in

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_14

SoC	System-on-Chip
SPM	Scratchpad Memory
SRAM	Static Random-Access Memory
STT-RAM	Spin-Transfer Torque Random-Access Memory
SWC	Software Cache

Contents

13.1 Motivating the Significance of Memory

The concept of memory and storage is of fundamental importance in hardware/-software codesign; it exhibits itself in the earliest stages of system design. Let us illustrate the ideas starting with the simplest examples and then proceed to more complex systems. With increasing system complexity, we go on to understand some of the larger trade-offs and decision-making processes involved.

Figure 13.1a shows a simple specification involving arrays and loops written in a programming language or Hardware Description Language (HDL), with some details such as type declaration omitted. Figure 13.1b shows a possible fully parallel implementation when the statement is synthesized into hardware, with four

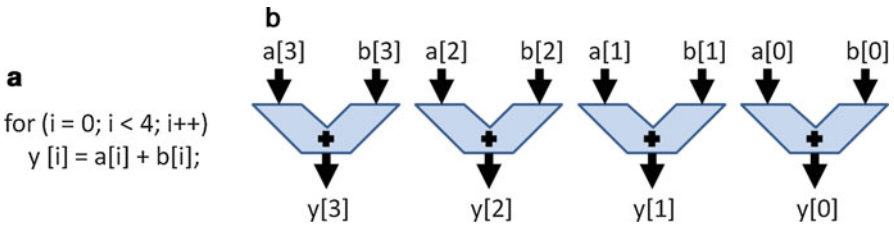


Fig. 13.1 (**a**) Code with loop and array (**b**) Hardware implementation

Arithmetic-Logic Units (ALUs). Such a specification may represent combinational logic without sequential/memory elements, as it does not involve the storage of the operands or result. Thus, in its simplest form, system implementation need not involve any memory. Note that an equivalent software implementation could consist of a sequence of addition and branch instructions, whose execution does involve registers and memory.

13.1.1 Discrete Registers

Memory elements are inferenced if we slightly modify the implementation scenario. Let us assume that there is a resource constraint of only two ALUs for implementing the specification of Fig. 13.1a. Now, we need to sequentialize the ALU computations over time so that the ALUs can be reused. This leads to registers being required in the implementation, with the computation being spread out over multiple clock cycles: $a[0]+b[0]$ and $a[2]+b[2]$ are performed in the first cycle, and $a[1]+b[1]$ and $a[3]+b[3]$ are performed in the second cycle. Since the ALU outputs have different values at different times, a more appropriate interface consists of registers connected to the ALU outputs. The select signals of the multiplexers, and load signals of the registers, would have to be generated by a small controller/FSM that asserts the right values in each cycle.

13.1.2 Organizing Registers into Register Files

The example of Fig. 13.1 was a simple instance of a specification requiring memory elements in its hardware translation. Discrete registers were sufficient for the small example. However, such an implementation does not scale well as we deal with larger amounts of data in applications. The interconnections become large and unstructured, with hard to predict area and delay behavior. For simplicity of implementation, the discrete registers are usually grouped into *register files* (RFs).

Figure 13.3 shows an alternative hardware arrangement with registers grouped into one common structure. The multiplexers, select lines, and load lines of Fig. 13.2 are now replaced with an addressing mechanism consisting of an address buses, data

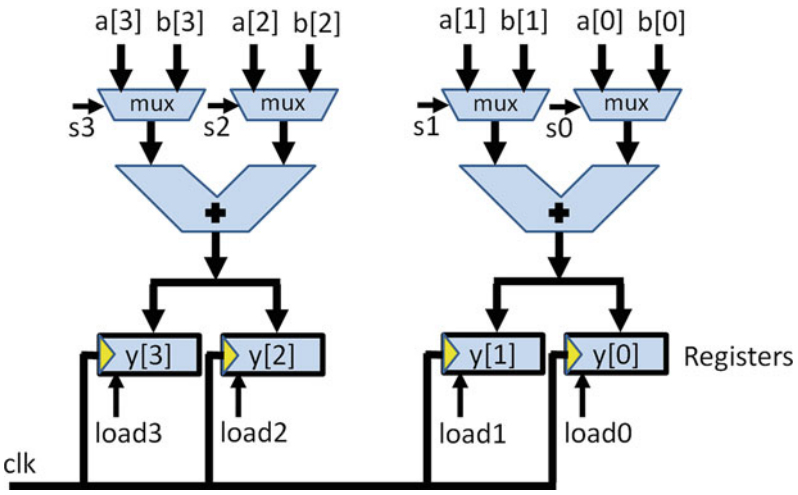
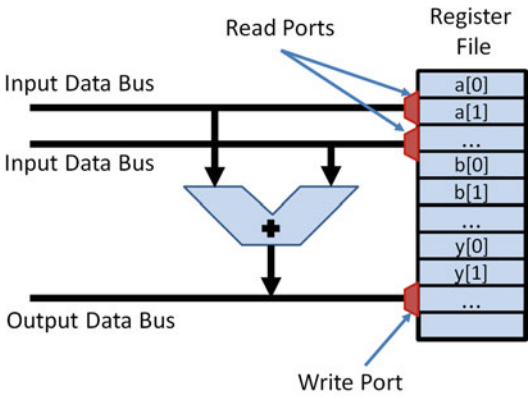


Fig. 13.2 Discrete registers in hardware implementations. Load lines are *load0* to *load3*. Mux select lines are *s0* to *s3*

Fig. 13.3 Architecture with registers grouped into register files



buses, and an internal address decoder structure that connects the data bus to the appropriate internal register. The ALU is now connected to the data buses instead of being directly connected to the registers [12]. The multiplexer and decoder structures highlighted as ports in Fig. 13.3 represent the peripheral hardware that is necessary to make the register file structure work. In other words, the register file does not consist of merely the storage cells; the multiplexing and decoding circuits do represent an area and power overhead here also, but the structure is more regular than in the discrete register case.

This abstraction of individual registers into a larger register file structure represents a fundamental trade-off involving memories in system design. The aggregation into register files is necessary for handling the complexity involving the storage and retrieval of large amounts of data in applications. One drawback of arranging the

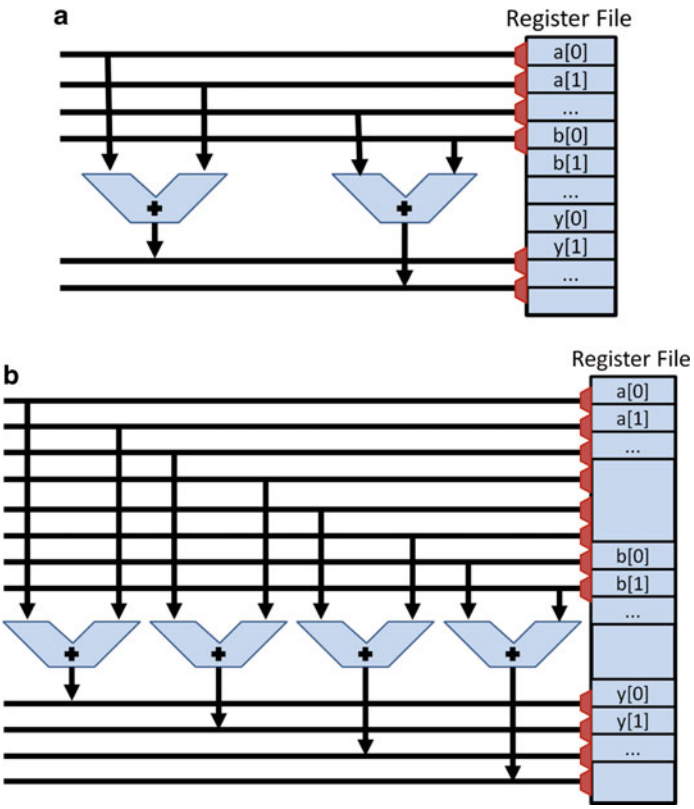


Fig. 13.4 Parallel access to register files (**a**) 6-port register file (**b**) 12-port register file

data in this way is that we have lost the ability to simultaneously access all the data, since the data bus can carry only one piece of data at any time. This leads to sequentialization of data accesses, which could impact performance severely.

To work around the sequential access problem, register files are organized to have multiple *ports*, each port consisting of an independent set of address and data buses, and control signals to indicate the operation (read, write, etc.). Figure 13.4a shows an architecture with two ALUs and a register file. In order to keep the ALUs busy, we should be able to read four operands simultaneously and also write two results back to the register file. This imposes a requirement for a total of six register file ports.

Extending the architecture to permit four simultaneous ALU operations, we observe from Fig. 13.4b that twelve ports are needed in the register file. Larger number of ports has the associated overhead of larger area, access times, and power dissipation in the register file. The peripheral multiplexing and decoding circuit increases correspondingly with the increased ports, leading to larger area and power overheads. Since the increased delays affect all register file accesses, the architecture

should be carefully chosen to reflect the requirements of the application. Alternative architectures that could be considered include splitting the register file into multiple banks, with each bank supporting a smaller number of ports. This has the advantage of faster and lower power access from the individual register file banks, while compromising on connectivity – all ALUs can no longer directly access data from all storage locations. Such trade-offs have been investigated in the context of clustered VLIW processors in general-purpose computing and also influence on-chip memory architecture choices in System-on-Chip (SoC) design. Simultaneous memory access through multiple ports also raises the possibility of access conflicts: a write request to a location through one port may be issued simultaneously with a write or read request to the same location through a different port. Such conflicts need to be resolved externally through an appropriate scheduling of access requests.

13.1.3 Packing Data into On-Chip SRAM

Registers and register files are the closest memory elements to computation hardware, making them the most quickly accessible storage. Fast access also implies an inherent constraint on their sizes – typical register files can store 16, 32, or 64 data words, which can sometimes extend to a few hundreds. When we need to process larger amounts of data, we have to incorporate other structures such as Static Random-Access Memory (SRAM). Register files and SRAMs are usually distinguished by the relative sizes and number of ports. SRAMs can accommodate hundreds of kilobytes of on-chip storage. Large SRAMs also have correspondingly fewer ports because the basic cell circuit for providing connectivity to a large number of ports does not scale well for large sizes, and the memory would incur large area, performance, and power overheads.

Figure 13.5 shows possible configurations where SRAM is integrated into SoC. In Fig. 13.5a the data bus of the SRAM is directly connected to ALUs, while in Fig. 13.5b the ALU is connected only to the register file, with the register file serving as the interface to the SRAM; data is first transferred from the SRAM to the register file before being operated upon by the ALUs. It is possible to consider discrete registers also here, instead of register files. Being denser, the SRAMs can store more data per unit area than the register files. However, the larger SRAMs also exhibit longer access times, which leads to a memory hierarchy as a natural architectural choice.

13.1.4 Denser Memories: Main Memory and Disk

The need for larger capacities in data storage leads to the incorporation of other memory structures as part of the hierarchy. Main memory and disks are the next architectural components that complete the hierarchy, with higher capacity and correspondingly higher access times. Modern main memories are usually implemented using Dynamic Random-Access Memory (DRAM), although other

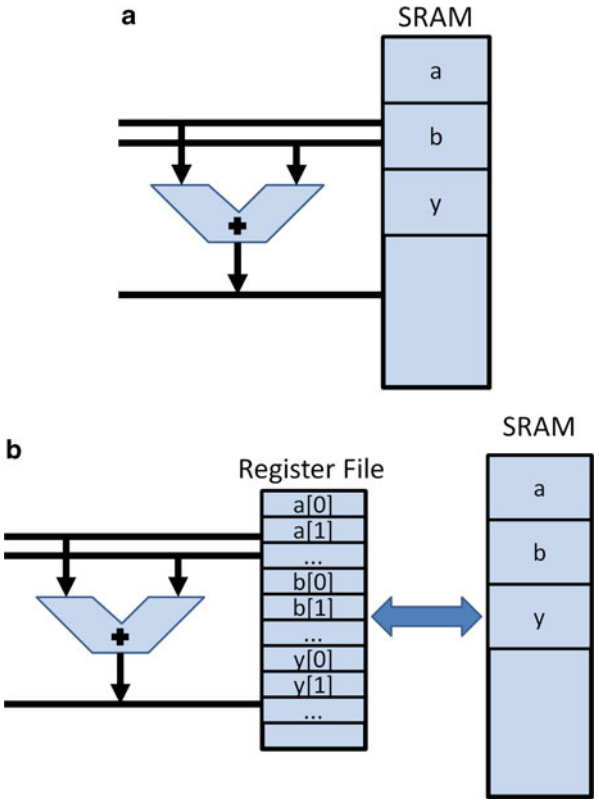


Fig. 13.5 (**a**) Data in SRAM (**b**) Hierarchically arranged register file and SRAM

memory technologies such as Phase Change Memory (PCM) and Spin-Transfer Torque Random-Access Memory (STT-RAM) have appeared on the horizon recently [14, 19, 21, 36].

The essential difference between register files and SRAM on one hand, and DRAM on the other, is that in the former, data is stored as long as the cells are powered on, whereas in DRAM the data, which is stored in the form of charge on capacitors, is lost over a period of time due to leakage of charge from the capacitors. To ensure storage for longer periods, the DRAM cells need to be refreshed at intervals. Note that *nonvolatile* memory technologies such as PCM and STT-RAM, referred to above, do not need to be refreshed in this way. However, they have other associated issues, which are discussed in more detail in ▸ Chap. 14, "Emerging and Nonvolatile Memory".

Figure 13.6 shows a simplified DRAM architecture highlighting some of the major features. The address is divided into a row address consisting of the higher-order bits and a column address consisting of the lower-order bits. A row decoder uses the row address to select a *page* from the core storage array and copy it to a buffer. A column decoder uses the column address to select the word at the right offset within the buffer and send it to the output data bus. If subsequent

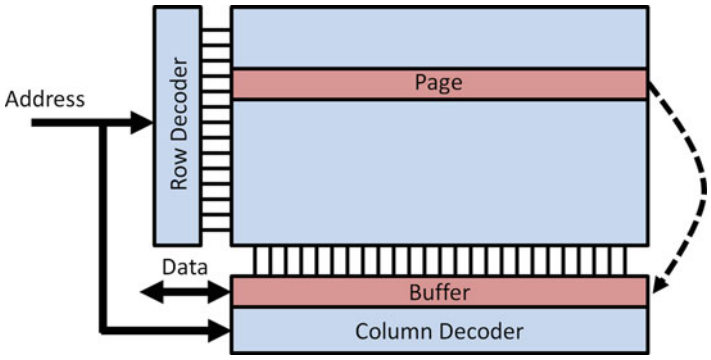


Fig. 13.6 DRAM architecture

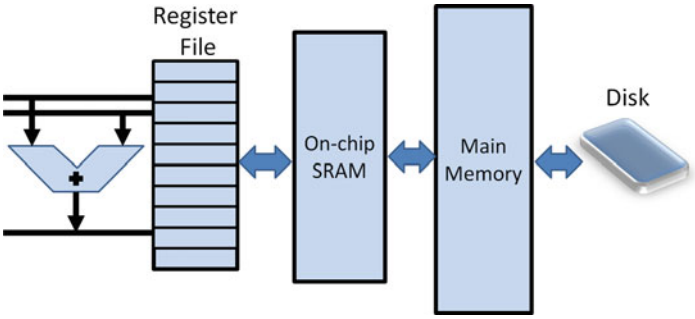


Fig. 13.7 Memory hierarchy may include disk

accesses occur to data within the same page, then we could skip the row decode stage and retrieve data directly from the buffer using only the column decode stage [30], employing what is referred to as an *open-page policy*. DRAMs have been architected around this central principle over the decades, although a large number of other structural details and management policies (including a *close-page policy* where the page is closed right after an access – a strategy that is useful when locality is weak) have been added. DRAM is generally incorporated as an off-chip component, although sometimes integrated on-chip with logic.

The final memory level in SoCs could include some form of nonvolatile storage such as solid-state disk (SSD, shown in Fig. 13.7). Occurring at a level beyond main memory, disk storage is often necessary when larger amounts of data need to be stored, for longer periods of time. The underlying technology is often flash-memory-based SSD.

13.1.5 Memory Hierarchy

System designers have to make a choice between the different types of memories discussed above, with the general trend being that smaller memories are faster,

whereas larger memories are slower. The common solution is to architect the memory system as a hierarchy of memories with increasing capacities, with the smallest memory (registers and register files) located closest to the processing units and the largest memory (DRAM and disk) lying farthest. This way, the processing units fetch the data from the closest memory very fast. There is also the requirement that the performance should not be overwhelmed by excessive accesses to the large memories.

Fortunately, the important concept of *locality of reference*, an important property exhibited by normal computing algorithms, plays an important role in this decision. *Spatial locality* refers to the observation that if a memory location is accessed, locations nearby are likely to be also accessed. This derives, for example, from (non-branch) instructions being executed as sequences, located in consecutive memory locations. Similarly, arrays accessed in loops also exhibit this property. *Temporal locality* refers to the observation that if a memory location is accessed, it is likely to be accessed again in the near future. This property can be related to instruction sequences executed multiple times in a loop and also data variables such as loop indices referenced multiple times within a short span of time, once in each iteration.

The locality property provides compelling motivation to organize the memory system as a hierarchy. If frequently accessed data and instructions can be stored/-found in levels of the memory located closer to the processor, then the average memory access times would decrease, which improves performance (and also power and energy). Within this general philosophy, a large number of combinations and configurations exist, making the overall memory architecture decision a complex and challenging process in hardware/software codesign.

13.2 Memory Architectures in SoCs

In Sect. 13.1 we reviewed the major technologies used in memory subsystems. These components are used to evolve different architectural blocks that are commonly used, depending on the requirements of the application. On-chip memories are dominated by different configurations of *caches* and *scratchpad memories*.

13.2.1 Cache Memory

Cache memory is a standard architectural block in the on-chip memory hierarchy. Designed to exploit the temporal and spatial locality properties exhibited by programs and data in typical applications, caches are SRAM-based structures that attempt to retain a copy of recently accessed information so that when it is required, the data is delivered from the cache instead of further levels of the hierarchy, thereby saving time and energy. Spatial locality is exploited by prefetching a *block* or *line*

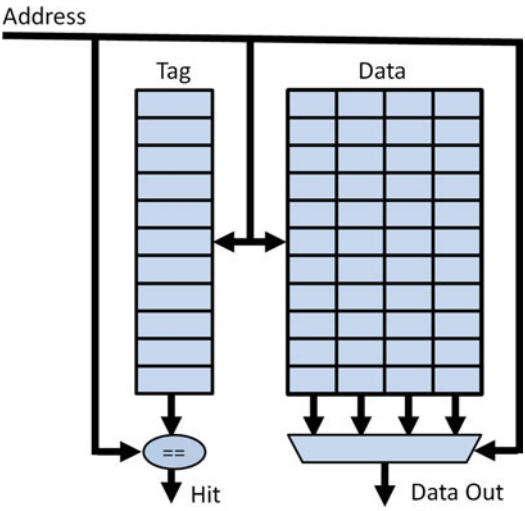


Fig. 13.8 Read operation in direct-mapped cache

of data when a single word is accessed, so that when adjacent words are accessed, they can be found in the cache. Temporal locality is exploited by implementing an appropriate *replacement policy* that attempts to retain relatively recently accessed data in the cache [12]. Caches are usually implemented using SRAM technology, but other technologies such as embedded DRAM and STT-RAM have also been explored for implementing caches [17].

Figure 13.8 shows a high-level block diagram of a *direct-mapped* cache. In this design, each address of the next memory level is mapped to one location in the cache. Since each cache level is smaller than the next level, a simple mapping function consisting of a subset of the address bits is used for determining the cache location from a given memory location. Consequently, several memory locations could map to the same cache location. When memory data is accessed, a cache line (consisting of four words from the Data memory shown in Fig. 13.8) is fetched and stored in the cache location to which the memory address maps. The higher-order address bits are also stored in the *tag* field of the cache to identify where the line came from. When a new address is presented to the cache, the higher-order address bits are compared with those stored at the corresponding location in the tag memory, and if there is a match (causing a cache *hit*), the data is delivered from the cache line itself. If there is no match (causing a cache *miss*), then the data has to be fetched from the next memory level.

The cache structure can be generalized to permit the same data to possibly reside in one of several cache locations. This permits some flexibility in the mapping and helps overcome limitations of the direct-mapped cache arising out of multiple memory addresses conflicting at the same cache location. The standard architecture for implementing this is a *set-associative* cache, with each line mapping to any of a set of locations. Figure 13.9 outlines the block diagram of a four-way set-associative cache, in which a cache line fetched from the memory can reside in one out of four

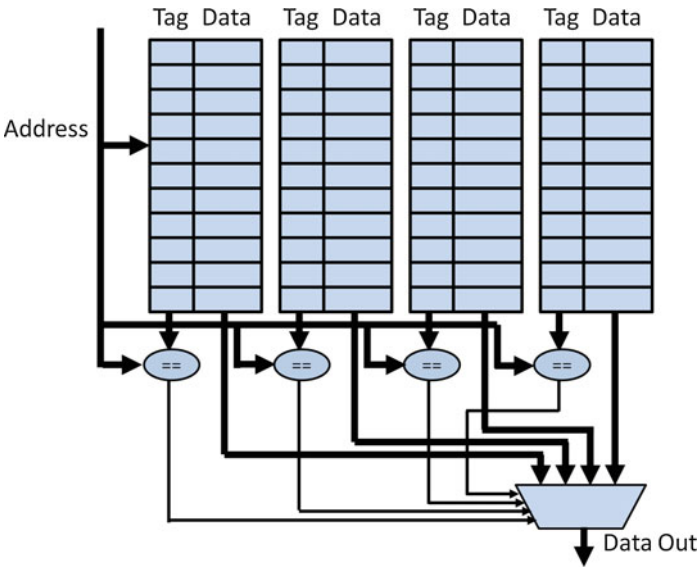


Fig. 13.9 Read operation in four-way set-associative cache

cache *ways*. The higher-order address bits are compared simultaneously with four tags, and if there is a match, a word from the corresponding way is delivered by the cache. If none of the tags match, then we have a cache miss, and a line is fetched from the next memory level. The decision of which line should be replaced in the set is taken by the replacement policy that might usually favor replacing the line that was accessed furthest in the past.

13.2.2 Scratchpad Memory

Scratchpad Memory (SPM) refers to simple on-chip memory, usually implemented with SRAM, that is directly addressable and where decisions of transfer to and from the next memory level are explicitly taken in software instead of implicitly through hardware replacement policies, as in caches [29]. Figure 13.10 shows a logical picture of the memory address map involving both SPM and cache. The caches are not visible in the address map because the cache storage decisions are not explicitly made in software. The SPM physically resides at roughly the same level as the cache, and its data is not accessed through the cache [28,32]. Although traditionally implemented in SRAM technology, other technologies such as STT-RAM are being considered for denser SPM implementation [37].

Scratchpad memory is actually simpler than caches because there is no need to include tags, comparators, implementation of replacement policies, and other control information. This makes it smaller, lower power, and more predictable,

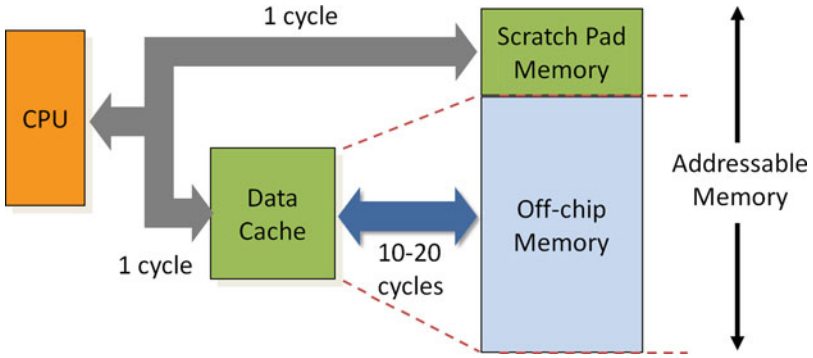


Fig. 13.10 Scratchpad memory address map

which makes it suitable for use in real-time and low-power systems. However, it does increase the work done in software (either by the developer, if done manually, or the compiler, if automated) because data transfers have to be explicitly performed in software.

Scratchpad memory could be integrated into system designs in a variety of ways, either independently or in conjunction with other memory structures. Figure 13.11 illustrates some such configurations. In the architecture of Fig. 13.11a, the local first-level memory consists of only SPM, which holds both instructions and data. The concept of a cache could still be useful, however, and a cache could be emulated within the SPM (Sect. 13.2.3). A Direct Memory Access (DMA) engine acts as the interface between the SPM and external memory. The DMA is responsible for transferring a range of memory data between different memories – in this case, between SPM and the next level. In Fig. 13.11b, the architecture supports both local SPM and hardware cache. In such systems, decisions have to be made for mapping code and data to either SPM or cache. Other variations are possible – for example, in Fig. 13.11c the local memory could be dynamically partitioned between SPM and cache.

13.2.3 Software Cache

Software Cache (SWC) refers to cache functionality being emulated in software using an SPM with no hardware cache support as the underlying structure. The tag structures discussed in Sect. 13.2.1 are still conceptually present, so they need to be separately stored in the same memory, and comparisons have to be implemented in software.

The working of a software cache is illustrated in Fig. 13.12. The SWC implementation consists of a cache line data storage and tag storage area.

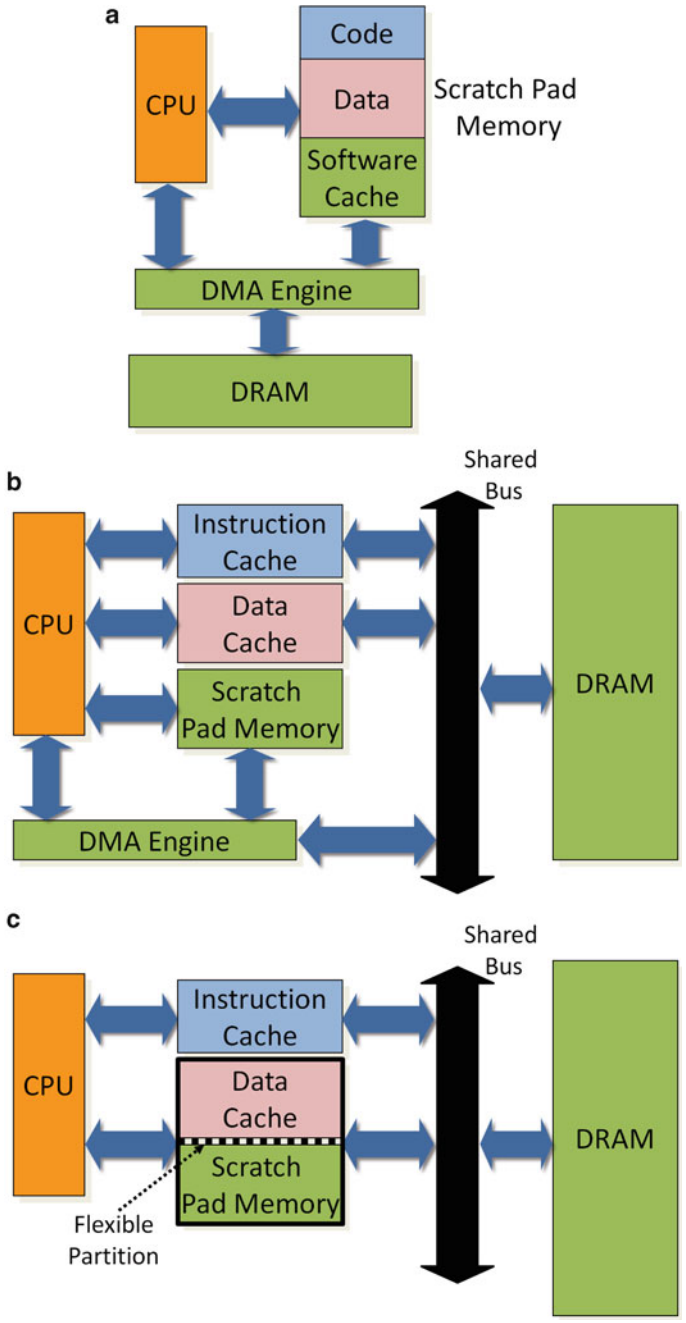


Fig. 13.11 Different architectural configurations for scratchpad memory (**a**) Local memory consists of only SPM. No hardware cache. (**b**) Local memory with both hardware cache and SPM (**c**) Dynamically configurable partition between local cache and SPM

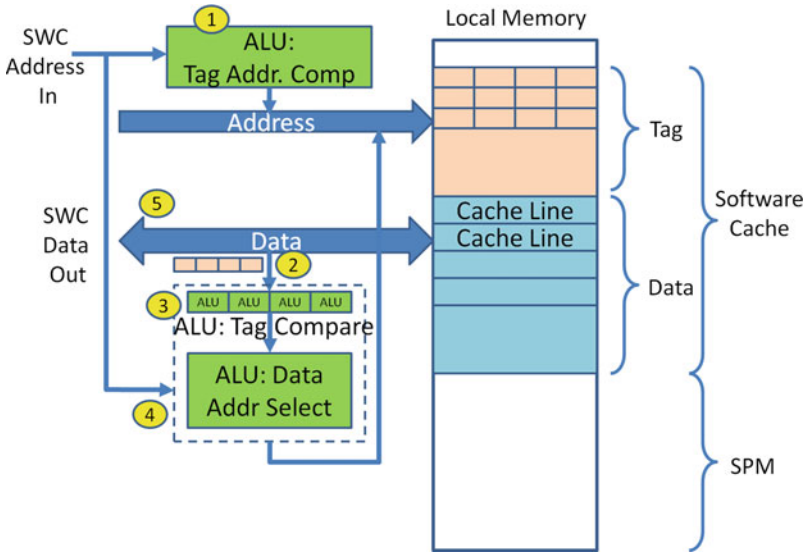


Fig. 13.12 Software caches: emulating caches in software

1. In Step 1, the index bits are extracted from the address to generate the address of the relevant tag within the SPM.
2. In Step 2, the tag is from the memory, with the four words corresponding to the four tag fields of the four-way set-associative cache.
3. In Step 3, the tags are compared against the tag bits of the address to determine a cache hit.
4. If a hit results, then the SPM address of the cache line is computed in Step 4.
5. As the final step, the data is fetched and delivered.

The emulation is relatively slow and energy inefficient compared to the hardware cache and, hence, should be judiciously used, for those data for which it is difficult to perform the static analysis required for SPM mapping [5, 7]. Efficient implementation of the software cache routines – such as a cache read resulting in a hit – used in the CELL library cause a roughly 5X performance overhead (six cycles for SPM access vs. 32 cycles for software cache access) (Sect. 13.3.2).

13.2.4 Memory in CGRA Architectures

We examine some of the on-chip memory architectures in some commercial and research SoCs and processors. Some of the designs are application specific or domain specific, while others are designed for broader applicability but still under restricted thread organization structures that do not apply to general-purpose software applications with random control structures. Such

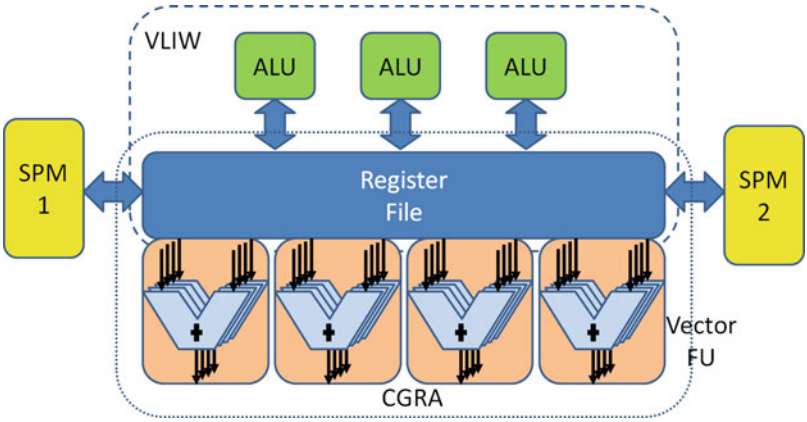


Fig. 13.13 Multiple SPMs in CGRA architecture

applications are more relevant to systems-on-a-chip rather than general-purpose computing.

The ADRES architecture [1] is a CGRA of processor cores and memories where the number of cores and memories is customizable for a specific application domain. The architecture is vector oriented, with a register file being connected to both the configurable array of vector processors (four in Fig. 13.13) and a VLIW processor that handles the control flow. In the instance of Fig. 13.13, there are two SPM blocks with wide vector interfaces to the register file. A significant responsibility lies on the system designer for utilizing such systems efficiently, which translates to challenging new problems for the compiler. The number of Central Processing Unit (CPU) cores and SPM instances has to be decided after a careful evaluation of the system throughput requirements and energy constraints. The compiler support also needs to accommodate the specialized architectures so that the application can benefit from it.

13.2.5 Hierarchical SPM

Figure 13.14 shows another instance of a processor system where a large number of independent SPMs are organized hierarchically into different memory levels [3]. Each individual core contains, apart from an ALU and register file, a level-1 SPM as well as hardware cache. Each block contains several such cores, along with a control processor with a level-2 SPM. Several such blocks exchange data through a shared memory in the form of a level-3 SPM. Finally, the entire chip is connected to a global shared memory implemented as a level-4 SPM. Such memory organizations have the ability to tremendously simplify multiprocessor architecture by not requiring complex cache coherence protocols but also need to rely on extensive support of sophisticated parallel programming environments and compiler analysis.

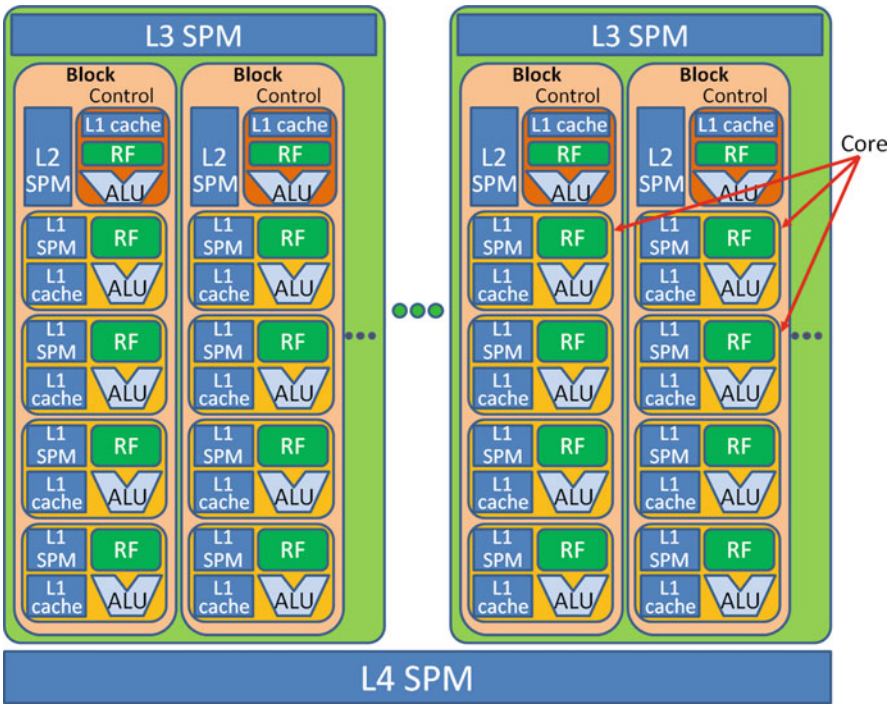


Fig. 13.14 Hierarchically organized SPM

13.3 Commercial SPM-Based Architectures

In this section we examine a few commercial architectures with interesting on-chip memory structures including scratchpad memory, in addition to conventional caches. These architectures have been used in a wide variety of applications with diverse requirements and constraints, ranging from low-power embedded systems to high-end gaming platforms and high-performance machines. We exclude general-purpose processors with conventional cache hierarchies.

13.3.1 ARM-11 Memory System

The Advanced Risc Machines (ARM) processor architecture family has been extensively used in embedded systems-on-chip. Figure 13.15 shows the local memory architecture of the ARM-11 processor, consisting of both hardware cache and SPM [2]. Data delivered from the memory subsystem could be routed from one of several sources: one of four memory banks corresponding to the four-way cache,

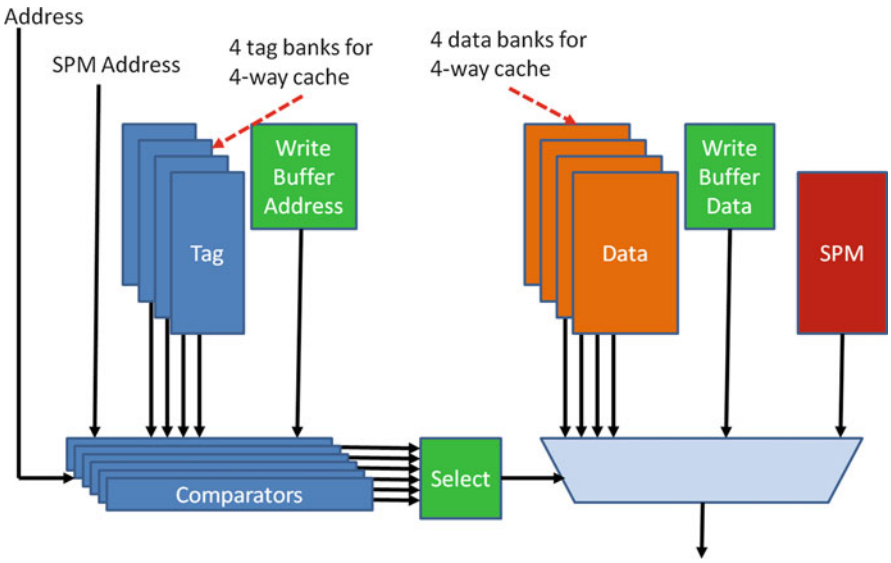


Fig. 13.15 The ARM-11 local memory architecture

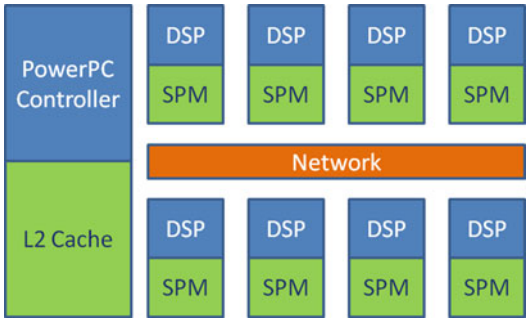


Fig. 13.16 Memory organization in the CELL processor

the SPM, and the write buffer associated with the cache. The selection of the data source is done by examining the address range, the write buffer locations, and the cache tags.

13.3.2 Local SPMs in CELL

The CELL processor [20], shown in Fig. 13.16, is a multiprocessor system consisting of eight digital signal processing engines connected over a ring network, each with a large local SPM storage intended for both instructions and data. There is no local hardware cache, and there is library support for software caches. At a higher level, a PowerPC [22] processor is used for control functions, with a regular level-2 hardware cache, which, in turn, interfaces with external DRAM.

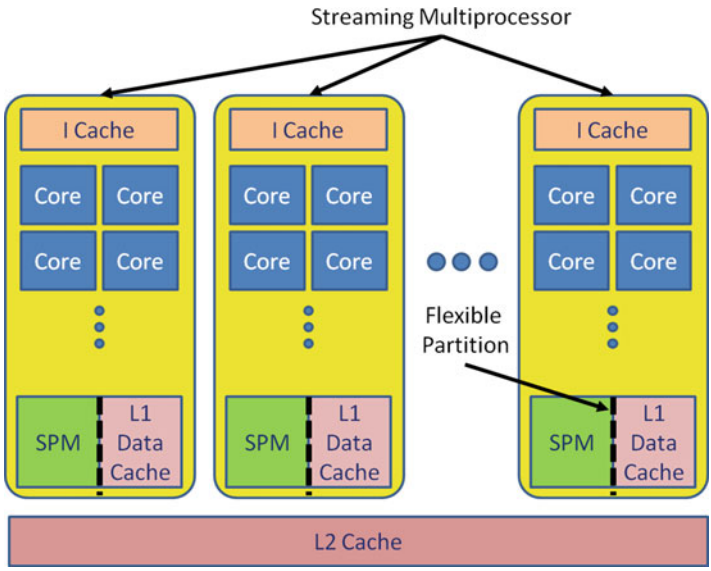


Fig. 13.17 Memory in the Fermi processor

13.3.3 Programmable First-Level Memory in Fermi

In the Fermi architecture [25] which represents the Graphics Processing Unit (GPU) class of designs (Fig. 13.17), each streaming multiprocessor consists of several processing cores with a shared data memory consisting of both SPM and hardware cache. The second-level cache is common to all the processors and interfaces with external DRAM. The architecture also supports the dynamic reconfiguration of the local memory into partitions of different SPM vs. cache sizes, depending on the requirements of the application. Although originally targeted at the graphics rendering application, architectures in this family have also been used for other applications with similar data-parallel properties, known as General-Purpose computing on Graphics Processing Units (GPGPU) applications [26].

13.4 Data Mapping and Run-Time Memory Management

The presence of advanced memory-related features requires a corresponding automated analysis functionality for exploiting the features efficiently. Conventional cache-oriented compiler analysis is sometimes applicable for these scenarios, but new mechanisms are typically necessary targeting the specialized memory structures. When data transfers are no longer automatically managed in hardware, the most fundamental problem that arises is the decision of where to map the

data from among all the memory structure choices available. Techniques have been developed to intelligently map data and instructions in SPM-based systems [6, 9, 15, 16, 38].

A major advantage of caches, from a methodology point of view, is the simplicity of their usage. Application executables that are compiled for one architecture with a given cache hierarchy also perform well for an architecture with a different cache structure. In principle, applications do not need to be recompiled and reanalyzed when the cache configuration changes in a future processor generation, although several cache-oriented analyses indeed rely on the knowledge of cache parameters. However, the other memory structures such as SPM would need a careful reanalysis of the application when the configuration is modified.

13.4.1 Tiling/Blocking

The conventional *tiling* or *blocking* optimization refers to the loop transformation where the standard iteration space covering array data is rearranged into tiles or blocks, to improve cache performance – this usually results in better temporal locality [8, 18, 33].

Figure 13.18 illustrates the tiling concept on a simple one-dimensional array, where the data is stored in main memory and needs to be fetched first into SPM for processing. Assuming no cache, the fetching and writing back of data have to be explicitly managed in software. For the code in Fig. 13.18a, the tiled version shown in Fig. 13.18b divides the array into tiles of size 100 and copies the tiles into array AA located in SPM (using the *MoveToSPM* routine). The inner loop now operates on the data in the SPM. If the data were modified, the tile would also need to be written back before proceeding to the next tile. The process is illustrated in Fig. 13.18c.

A generalization of the simple tiling concept is illustrated in Fig. 13.19a. Here, a two-dimensional array is divided into 2D tiles. In the classical tiling optimization,

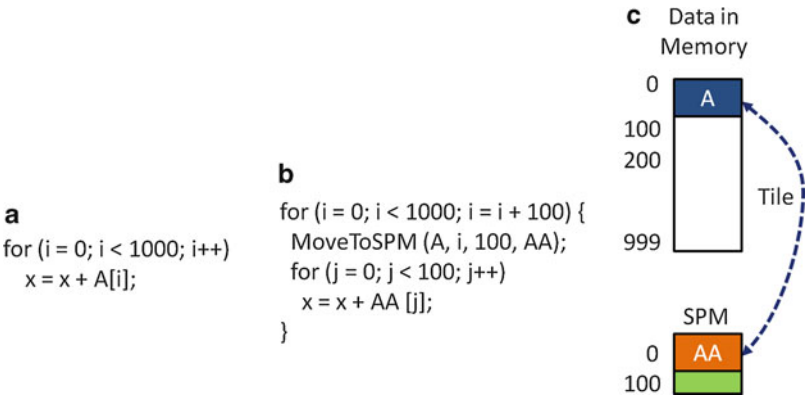


Fig. 13.18 The tiling/blocking transformation (**a**) original code (**b**) tiled code (**c**) tiling and SPM

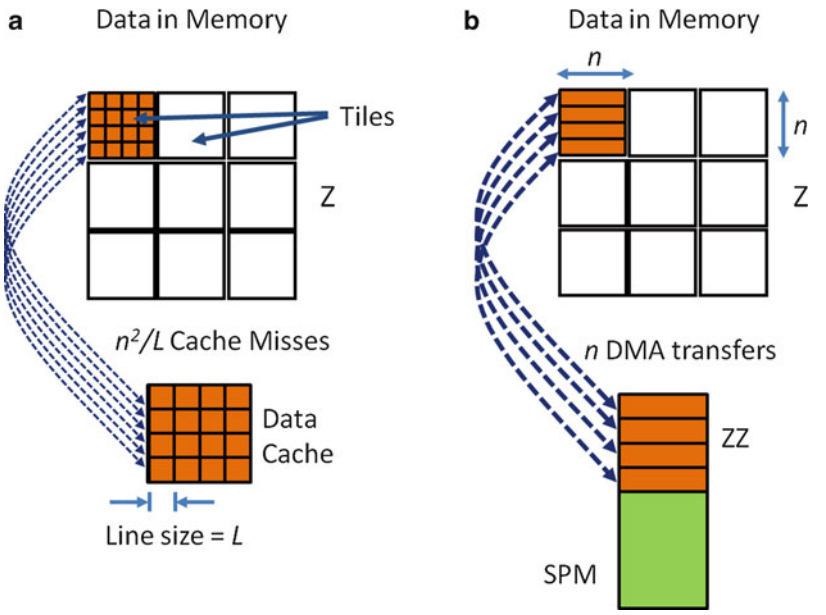


Fig. 13.19 Tiling multidimensional data (**a**) in data caches and (**b**) in scratchpad memory

the loop iteration space is modified so that the computation is first performed in one tile before proceeding to another. The tile height and width are carefully chosen such that the tile size is less than the data cache size, and also, elements within a tile exhibit minimal cache conflicts between themselves.

The tiling idea can be extended to SPM, where each tile is first fetched into the SPM for processing (Fig. 13.19b). All processing then takes place on SPM data, leading to a lower power solution because each access to the SPM is more energy efficient than the corresponding access to a hardware cache of similar capacity. Figure 13.20 shows an example of a tiled matrix multiplication targeted at SPM storage. Tiles of data are moved into the SPM using the *READ_TILE* routine before being processed. The iteration space is divided into a six-deep nested loop. This general principle is followed in SPM-based storage, where data is first fetched into the relatively small SPM, and actual processing is then performed on SPM data. The overhead of fetching data into the SPM is usually overcome by the energy-efficient accesses to the SPM.

13.4.2 Reducing Conflicts

A slightly different example of data mapping and partitioning is shown in Fig. 13.21. The array access pattern for the code shown in Fig. 13.21 is illustrated in Fig. 13.21b for the first two iterations of the j-loop. We observe that the *mask* array is small and

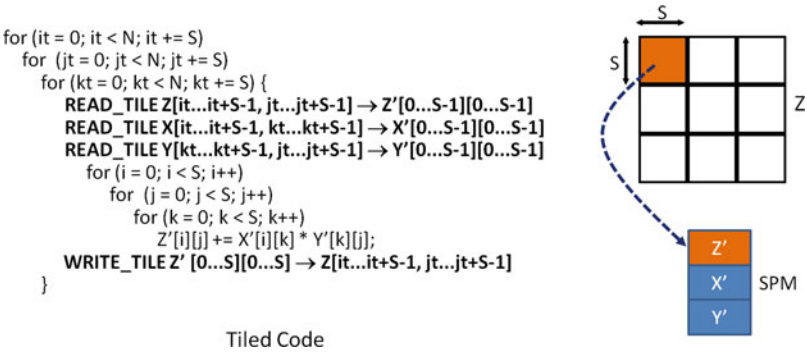


Fig. 13.20 Tiled matrix multiplication

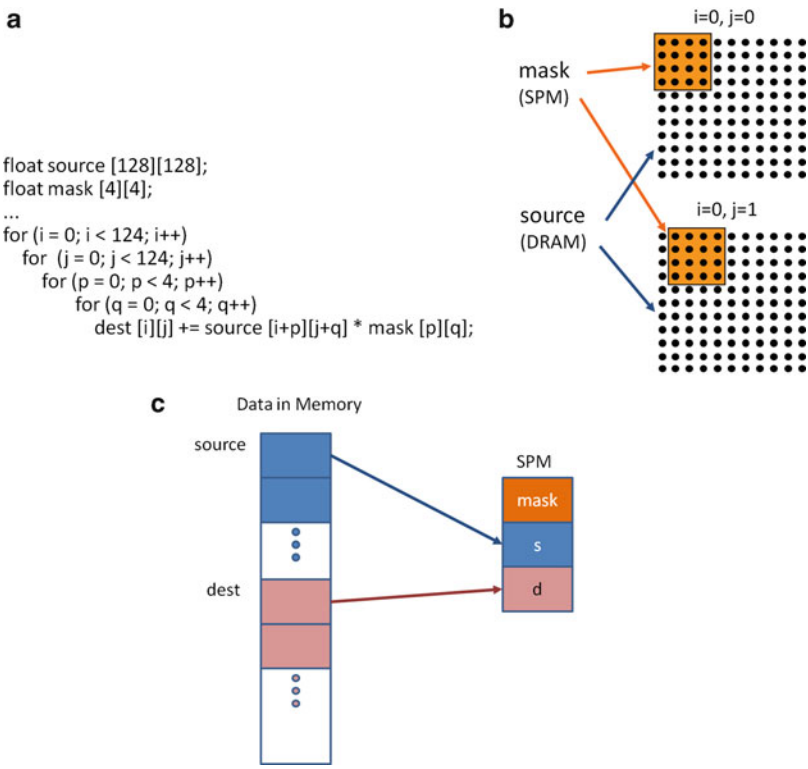


Fig. 13.21 (**a**) Convolution code (**b**) Data accesses in the first two iterations (**c**) SPM mapping

is accessed a large number of times. In comparison, the *source* array is large with the accesses exhibiting good spatial locality. It is possible that the two arrays would conflict in the cache if the cache is small. A good data mapping decision would be

to store the *mask* array in SPM and access *source* through the cache [31]. Another strategy illustrated in Fig. 13.21c is to fetch tiles from *source* and *dest* arrays into SPM for processing.

13.5 Comparing Cache and Scratchpad Memory

The quality of memory mapping decisions is closely related to the appropriate modeling of the cache and SPM parameters. Suitable high-level abstractions are necessary so that the impact of mapping decisions can be quickly evaluated. In this section we present a comparison of caches and SPM with respect to the area and energy parameters. There is no explicit access time comparison because these are similar for both; cache access times are dominated by the data array access time, which is also present in SPM.

13.5.1 Area Comparison

Caches are associated with an area overhead compared to SPM because of the tag array that is used for managing the cache. Figure 13.22 shows a comparison of the areas of SPM and a direct-mapped cache, for different memory capacities ranging from 8 to 128 KB, assuming a 32-byte line size and 32 nm process technology using the popular CACTI cache modeling tool [23]. For the same capacity, we observe a 10–32% additional area for the cache, compared to SPM.

13.5.2 Energy Comparison

We compare the energy dissipation of caches and SPM by first outlining a simple energy model of the memories expressed in terms of the different components. Since

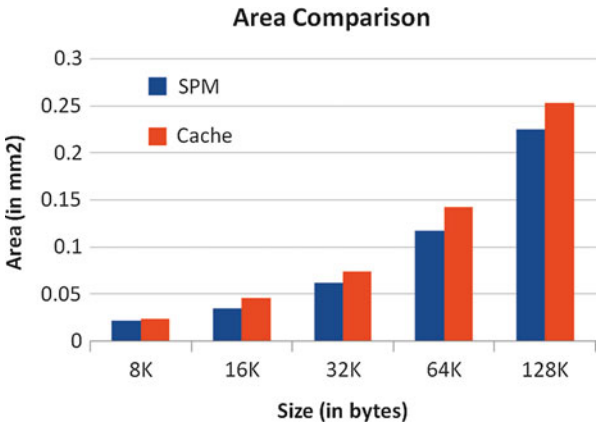


Fig. 13.22 Comparison of SPM and cache areas for different memory capacities

the mapping process may also result in conflict misses in the cache, we also study the variation of the memory energy with the conflict miss ratio, in addition to standard parameters such as memory capacity.

13.5.2.1 Energy Model for Tiled Execution

Table 13.1 shows a simplified model of the dynamic energy dissipation components of cache-based and SPM-based systems, when an $O(n^3)$ algorithm such as matrix multiplication is executed with an $n \times n$ tile. Column 1 lists the energy components in the memories.

- The computation remains identical in both cases, so the associated energy E_{Comp} is assumed to be equal.
- The dynamic energy dissipated during each memory accesses is higher in caches because of the additional energy $E_{Dyn-Tag}$ dissipated in the tag array, apart from that in the data array ($E_{Dyn-Data}$). The SPM's dynamic memory energy is limited to the data array energy $E_{Dyn-Data}$. To generate the total dynamic memory energy, the per-access energy values are multiplied by n^3, which is assumed to be the number of memory accesses required for the tile's processing.
- When the memories are idle, leakage energy is dissipated. The common data array causes a leakage energy $E_{Leak-Data}$ in both SPM and cache. The cache dissipates an additional $E_{Leak-Tag}$ due to the tag array.
- Finally, the data transfer overheads need to be carefully considered; they are sensitive to specific processor and memory architectures. The simple energy expressions for these overheads given in Table 13.1 highlight an important consideration: it is usually more energy efficient to fetch larger chunks of consecutive data from the main memory, instead of smaller chunks (the chunk size is also called *burst length* for main memory accesses). For the SPM, we assume a DMA architecture in which the $n \times n$ tile is fetched into the SPM by n DMA transfers of length n each. Each DMA transfer, fetching n elements, dissipates $E_{DMA}(n)$. In contrast, the data transfers triggered by cache misses are of size L, the cache line size, which is expected to be much smaller than n.

Table 13.1 Data cache vs. scratchpad memory energy comparison for computing an $O(n^3)$ algorithm on an $n \times n$ tile. Cache line size = L. f is the conflict miss ratio

Energy component	**SPM**	Data cache	Description
Computation	E_{Comp}	E_{Comp}	Computation stays fixed
Dynamic energy	$E_{Dyn-Data} \times n^3$	$(E_{Dyn-Data} + E_{Dyn-Tag}) \times n^3$	Tag access causes extra dynamic energy in cache
Leakage energy	$E_{Leak-Data}$	$E_{Leak-Data} + E_{Leak-Tag}$	Tag array causes extra leakage energy in cache when idle
Data transfer overheads	$E_{DMA}(n) \times n$	$E_{Miss}(L) \times (\frac{n^2}{L} + n^3 f)$	Longer burst in DMA is more energy efficient. Conflicts cause extra cache misses

Since all the tile data has to be fetched to the cache, there would be an estimated $\frac{n^2}{L}$ compulsory misses, each leading to an energy dissipation of $E_{Miss}(L)$. $E_{DMA}(n)$ for transferring a tile row of n elements is expected to be smaller than $E_{Miss}(L) \times \frac{n}{L}$, the corresponding cache energy.

Capacity misses in the cache (occurring due to insufficient cache size) are avoided by choosing an appropriate tile size, but conflict misses (occurring due to limitations of the mapping function, in spite of sufficient space) may not be completely avoided and lead to an additional $n^3 f$ misses, where f is the conflict miss ratio (defined as the number of conflict misses per access).

13.5.2.2 Sensitivity to Memory Capacity

Figure 13.23 shows a comparison of the per-access read energy for SPM and a direct-mapped cache, for different memory capacities ranging from 8 to 128 KB, assuming a 32-byte line size and 32nm process technology [23]. For the same capacity, we observe a 5–10% additional access energy for the cache, compared to SPM.

13.5.2.3 Sensitivity to Conflict Misses

Let us study the impact of one of the parameters identified in Table 13.1 – the conflict misses in tiling. As observed in Table 13.1, tiling may cause overheads in the cache if the memory accesses are subject to conflict misses. What is the extent of this overhead?

Figure 13.24 plots a comparison between the dynamic energy of the SPM and data cache for different miss ratios between 0 and 7% in a 50 × 50 tile (i.e., $n = 50$). We have ignored leakage and computation energy values and have assumed a 10% extra energy due to the tag array ($E_{Dyn-Tag} = 0.1 \times E_{Dyn-Data}$); a 2X and 4X overhead for DMA (per-word) and cache miss (per word), respectively. That is, $E_{DMA}(n) = 2n \times E_{Dyn-Data}$ and $E_{Miss}(L) = 4L \times E_{Dyn-Data}$. We notice that the memory energy overhead rises significantly as the miss ratio increases and amounts to an increase of 40% over the SPM energy for a 7% conflict miss ratio.

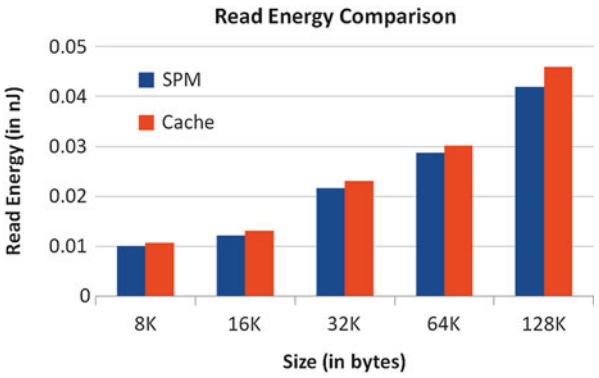


Fig. 13.23 Comparison of SPM and cache dynamic energy for different memory capacities

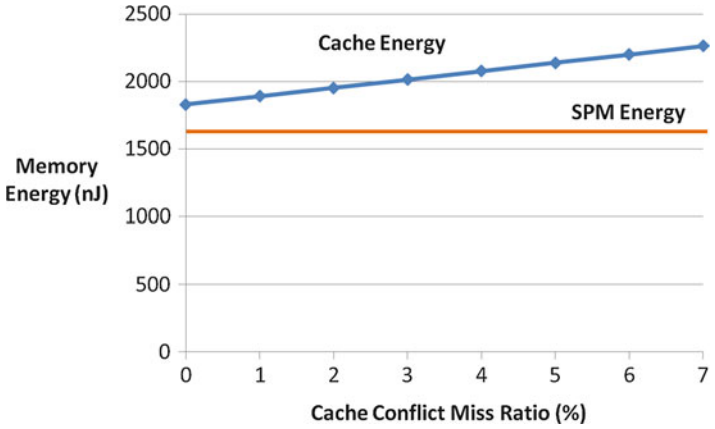


Fig. 13.24 Comparison of SPM and cache dynamic energy for different conflict miss ratios for 50×50 tile

This illustrates the importance of the appropriate mapping decision, which should ideally be implemented through a systematic procedure that estimates the benefits and cost overheads for different data mapping possibilities.

13.6 Memory Customization and Exploration

In addition to the diverse memory structures in SoC architectures, there is often the opportunity to customize the architecture for a single application or domain of applications. For example, the size and number of caches, register files, and SPMs could be customized. This process requires an exploration phase that involves iterating between architectural possibilities and compiler analysis to extract the best performance and power from each architectural instance [4, 11, 32, 35, 39]. Fast estimators are necessary to help converge on the final architecture.

13.6.1 Register File Partitioning

The register file could be an early candidate for application specific customization. The RF size could be determined based on application requirements. Further, from the application behavior, we could determine that a small set of registers need to be frequently accessed, whereas other data in the register file might not be accessed as frequently. This knowledge could be exploited by dividing the RF into two physical partitions, one smaller than the other (Fig. 13.25). If most accesses are routed to the smaller RF partition, the overall energy consumption could be smaller than the standard RF architecture where a larger RF is accessed for every register data access [24].

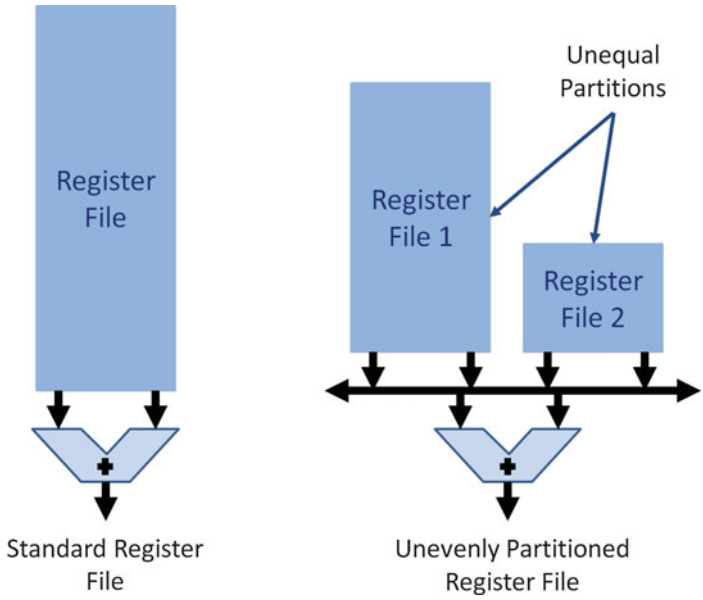


Fig. 13.25 Partitioning the register file. Uneven partitioning can lead to energy efficiency if the majority of accesses are to the smaller partition

13.6.2 Inferring Custom Memory Structures

Generalizing the register imbalance observed above, we could have a small range of memory addresses being relatively heavily accessed, leading to an analogous situation where the small range could be mapped to a small physical memory, which could lead to overall energy efficiency. Such a situation could be detected either by a compiler analysis or an execution profile of the application. This is illustrated in Fig. 13.26. The graph in Fig. 13.26a shows that a small range dominates the memory accesses. This could lead to an architectural possibility indicated in Fig. 13.26b, where this range of addresses is stored in a separate small memory. Memory accesses lying in this range could be more energy efficient because the access is made to a much smaller physical memory module [27]. The overhead of the range detection needs to be factored here. Custom memory structures could also be inferred by a data locality compiler analysis in loop nests, leading to relatively heavily accessed arrays being mapped to separate memory structures so as to improve overall energy efficiency [28].

13.6.3 Cache Customization and Reconfiguration

The presence of caches sometimes leads to opportunities for configuring the local memory in several ways. Caches themselves have a large number of parameters that

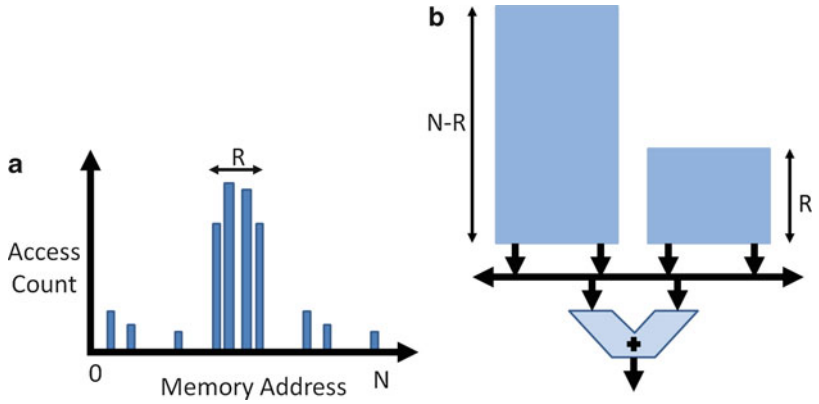


Fig. 13.26 (**a**) Profile of memory accesses (**b**) Memory architecture derived from profile

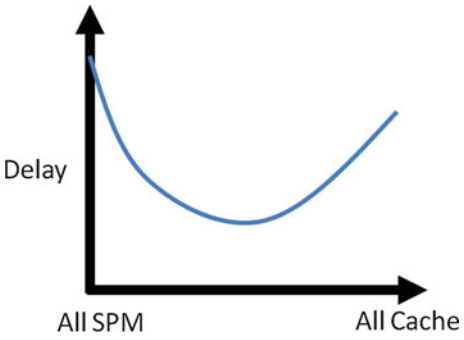


Fig. 13.27 Partitioning local memory between SPM and cache

could be tuned to the requirements of an application. Further, the coexistence of caches with other structures such as SPM expands the scope for such customization.

One exploration problem that comes up in this context is to divide a given amount of local memory space into cache and SPM. The best partitioning would be application dependent, and a compiler analysis of the application behavior would help determine the best partition. As Fig. 13.27 indicates [31], both extremes of all cache and all SPM may not be the best because different application data access patterns are suitable for different memory types. The best partition may lie somewhere in between [5, 31].

With processors such as Fermi permitting dynamic local memory reconfiguration (Sect. 13.3.3), the partitioning between cache and SPM could also be performed during the application execution, with different partitions effected during different application phases. Apart from size, possibilities also exist for dynamically reconfiguring other cache parameters such as associativity and management policy. Dynamic adjustment of cache associativity may help identify program phases where some ways can be turned off to save power. In Fig. 13.28a, the four-way cache has all four banks active at time $t1$, but two ways are turned off at $t2$ [40]. When a

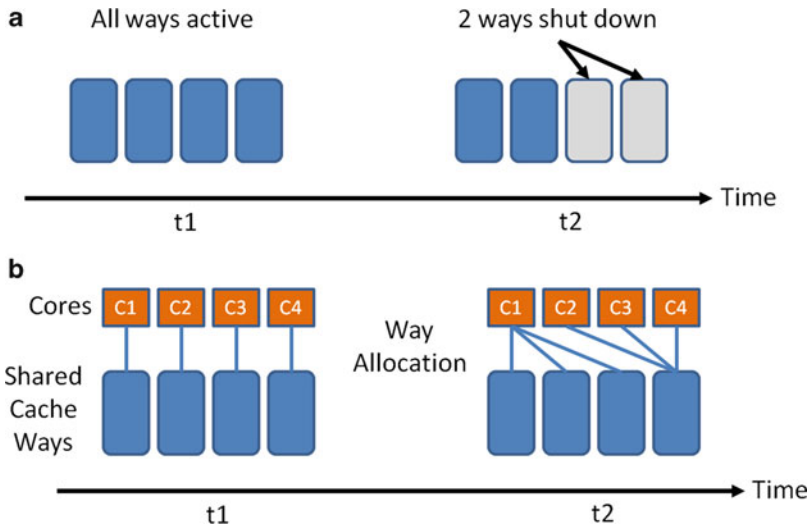


Fig. 13.28 Cache way configuration. (**a**) Ways shut down to save power (**b**) Dynamic allocation of ways to cores

cache is shared among several processor cores, an active management policy could exclusively allocate different sets of ways to different cores, with the objective of maximizing overall throughput. In Fig. 13.28b each core is allocated one exclusive way at time $t1$, but at $t2$, three ways are allocated to core $c1$, while the other way is shared among the remaining three cores [13, 34]. Such a decision could result from an analysis of the loads presented to the shared cache by the four cores. Application-specific analysis could also reveal possibilities for improving the cache mapping function [10].

13.7 Conclusions

In this chapter we reviewed some of the basics of memory architectures used in hardware/software codesign. While the principles of memory hierarchies used in general-purpose processors are relatively well defined, systems-on-chip tend to use a wide variety of different memory organizations according to the requirements of the application. Nevertheless, the architectures can be classified into a few conceptual classes such as registers, register files, caches, and scratchpad memories, instanced and networked in various ways. Codesign environments give rise to the possibility of integrating both the memory architecture as well as the application data mapping into the memory, leading to exciting technical challenges that require a fast exploration of the large number of configurations and mapping possibilities. As memory technologies continue to evolve, the problem of selecting and exploiting memory architectures continues to be relevant and expands in scope because of the

different trade-offs associated with memories with very different properties. As the technology marches forward, the integration of nonvolatile memories into system design poses very interesting new and exciting problems for the designer. This topic is discussed further in ▶ Chap. 14, "Emerging and Nonvolatile Memory".

Acknowledgments The author acknowledges Lokesh Siddhu for generating the data for memory configurations reported in Figs. 13.23 and 13.24.

References

1. Aa TV, Palkovic M, Hartmann M, Raghavan P, Dejonghe A, der Perre LV (2011) A multi-threaded coarse-grained array processor for wireless baseband. In: IEEE 9th symposium on application specific processors SASP, San Diego, 5–6 June 2011, pp 102–107
2. ARM Advanced RISC Machines Ltd (2006) ARM1136JF-S and ARM1136J-S, Technical Reference Manual, r1p3 edn
3. Carter NP, Agrawal A, Borkar S, Cledat R, David H, Dunning D, Fryman JB, Ganev I, Golliver RA, Knauerhase RC, Lethin R, Meister B, Mishra AK, Pinfold WR, Teller J, Torrellas J, Vasilache N, Venkatesh G, Xu J (2013) Runnemede: an architecture for ubiquitous high-performance computing. In: 19th IEEE international symposium on high performance computer architecture HPCA, Shenzhen, 23–27 Feb 2013, pp 198–209
4. Catthoor F, Wuytack S, De Greef E, Balasa F, Nachtergaele L, Vandecappelle A (1998) Custom memory management methodology: exploration of memory organisation for embedded multimedia system design. Kluwer Academic Publishers, Norwell, USA
5. Chakraborty P, Panda PR (2012) Integrating software caches with scratch pad memory. In: Proceedings of the 15th international conference on compilers, architecture, and synthesis for embedded systems, pp 201–210
6. Chen G, Ozturk O, Kandemir MT, Karaköy M (2006) Dynamic scratch-pad memory management for irregular array access patterns. In: Proceedings of the conference on design, automation and test in Europe DATE, Munich, 6–10 Mar 2006, pp 931–936
7. Chen T, Lin H, Zhang T (2008) Orchestrating data transfer for the CELL/BE processor. In: Proceedings of the 22nd annual international conference on supercomputing, ICS '08, pp 289–298
8. Coleman S, McKinley KS (1995) Tile size selection using cache organization and data layout. In: Proceedings of the ACM SIGPLAN'95 conference on programming language design and implementation (PLDI), pp 279–290
9. Francesco P, Marchal P, Atienza D, Benini L, Catthoor F, Mendias, JM (2004) An integrated hardware/software approach for run-time scratchpad management. In: Proceedings of the 41st annual design automation conference, DAC'04, pp 238–243
10. Givargis T (2003) Improved indexing for cache miss reduction in embedded systems. In: Proceedings of the 40th design automation conference, pp 875–880
11. Grun P, Dutt N, Nicolau A (2003) Memory architecture exploration for programmable embedded systems. Kluwer Academic Publishers, Boston
12. Hennessy JL, Patterson DA (2003) Computer architecture: a quantitative approach, 3rd edn. Morgan Kaufmann Publishers Inc., San Francisco
13. Jain R, Panda PR, Subramoney S (2016) Machine learned machines: adaptive co-optimization of caches, cores, and on-chip network. In: 2016 design, automation & test in Europe, pp 253–256
14. Jog A, Mishra AK, Xu C, Xie Y, Narayanan V, Iyer R, Das CR (2012) Cache revive: architecting volatile STT-RAM caches for enhanced performance in CMPs. In: Design automation conference (DAC), pp 243–252. doi: 10.1145/2228360.2228406

15. Kandemir MT, Ramanujam J, Irwin MJ, Vijaykrishnan N, Kadayif I, Parikh A (2001) Dynamic management of scratch-pad memory space. In: Proceedings of the 38th design automation conference, pp 690–695
16. Kandemir MT, Ramanujam J, Irwin MJ, Vijaykrishnan N, Kadayif I, Parikh A (2004) A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. IEEE Trans CAD Integr Circuits Syst 23(2):243–260
17. Komalan MP, Tenllado C, Perez JIG, Fernández FT, Catthoor F (2015) System level exploration of a STT-MRAM based level 1 data-cache. In: Proceedings of the 2015 design, automation & test in Europe conference & exhibition DATE, Grenoble, 9–13 Mar 2015, pp 1311–1316
18. Lam MS, Rothberg EE, Wolf ME (1991) The cache performance and optimizations of blocked algorithms. In: ASPLOS-IV proceedings - fourth international conference on architectural support for programming languages and operating systems, pp 63–74
19. Li H, Chen Y (2009) An overview of non-volatile memory technology and the implication for tools and architectures. In: Design, automation test in Europe conference exhibition (DATE), pp 731–736
20. Liu T, Lin H, Chen T, O'Brien JK, Shao L (2009) Dbdb: optimizing DMA transfer for the CELL BE architecture. In: Proceedings of the 23rd international conference on supercomputing, pp 36–45
21. Liu Y, Yang H, Wang Y, Wang C, Sheng X, Li S, Zhang D, Sun Y (2014) Ferroelectric nonvolatile processor design, optimization, and application. In: Xie Y (ed) Emerging memory technologies. Springer, New York, pp 289–322. doi: 10.1007/978-1-4419-9551-3_11
22. May C, Silha E, Simpson R, Warren H (1994) The PowerPC architecture: a specification for a new family of RISC processors, 2 edn. Morgan Kaufmann, San Francisco, USA
23. Muralimanohar N, Balasubramonian R, Jouppi NP (2009) CACTI6.0: A tool to model large caches. Technical Report HPL-2009-85, HP Laboratories
24. Nalluri R, Garg R, Panda PR (2007) Customization of register file banking architecture for low power. In: 20th international conference on VLSI design, pp 239–244
25. NVDIA Corporation (2009) NVIDIA's Next Generation CUDA Compute Architecture: Fermi
26. Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn A, Purcell TJ (2007) A survey of general-purpose computation on graphics hardware. Comput Graphics Forum 26(1):80–113
27. Panda PR, Silpa B, Shrivastava A, Gummidipudi K (2010) Power-efficient system design. Springer, US
28. Panda PR, Catthoor F, Dutt ND, Danckaert K, Brockmeyer E, Kulkarni C, Vandecappelle A, Kjeldsberg PG (2001) Data and memory optimization techniques for embedded systems. ACM Trans Design Autom Electr Syst 6(2):149–206
29. Panda PR, Dutt ND, Nicolau A (1997) Efficient utilization of scratch-pad memory in embedded processor applications. In: European design and test conference, ED&TC '97, Paris, 17–20 Mar 1997, pp 7–11
30. Panda PR, Dutt ND, Nicolau A (1998) Incorporating DRAM access modes into high-level synthesis. IEEE Trans CAD Integr Circuits Syst 17(2):96–109
31. Panda PR, Dutt ND, Nicolau A (1999) Local memory exploration and optimization in embedded systems. IEEE Trans CAD Integr Circuits Syst 18(1):3–13
32. Panda PR, Dutt ND, Nicolau A (1999) Memory issues in embedded systems-on-chip. Kluwer Academic Publishers, Boston
33. Panda PR, Nakamura H, Dutt ND, Nicolau A (1999) Augmenting loop tiling with data alignment for improved cache performance. IEEE Trans Comput 48(2):142–149
34. Qureshi MK, Patt YN (2006) Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches. In: 39th annual IEEE/ACM international symposium on microarchitecture, pp 423–432
35. Ramo EP, Resano J, Mozos D, Catthoor F (2006) A configuration memory hierarchy for fast reconfiguration with reduced energy consumption overhead. In: 20th international parallel and distributed processing symposium IPDPS

36. Raoux S, Burr G, Breitwisch M, Rettner C, Chen Y, Shelby R, Salinga M, Krebs D, Chen SH, Lung H, Lam C (2008) Phase-change random access memory: a scalable technology. IBM J Res Dev 52(4.5):465–479. doi: 10.1147/rd.524.0465
37. Rodríguez G, Touriño J, Kandemir MT (2014) Volatile STT-RAM scratchpad design and data allocation for low energy. ACM Trans Archit Code Optim (TACO) 11(4):38:1–38:26
38. Steinke S, Wehmeyer L, Lee B, Marwedel P (2002) Assigning program and data objects to scratchpad for energy reduction. In: Design, automation and test in Europe, pp 409–415
39. Wuytack S, Diguet JP, Catthoor F, Man HJD (1998) Formalized methodology for data reuse: exploration for low-power hierarchical memory mappings. IEEE Trans Very Larg Scale Integr Syst 6(4):529–537
40. Zhang C, Vahid F, Yang J, Najjar W (2005) A way-halting cache for low-energy high-performance systems. ACM Trans Archit Code Optim 2(1):34–54