Memory Architectures for Embedded Systems-On-Chip

Preeti Ranjan Panda[1] and Nikil D. Dutt[2]

[1] Dept. of Computer Science and Engineering, Indian Institute of Technology, Delhi
Hauz Khas, New Delhi - 110 016, India
panda@cse.iitd.ernet.in

[2] Center for Embedded Computer Systems, University of California, Irvine
Irvine, CA 92697-3425, USA
dutt@uci.edu

Abstract. Embedded systems are typically designed for one or a few target applications, allowing for customization of the system architecture for the desired system goals such as performance, power and cost. The memory subsystem will continue to present significant bottlenecks in the design of future embedded systems-on-chip. Using advance knowledge of the application's instruction and data behavior, it is possible to customize the memory architecture to meet varying system goals. On one hand, different applications exhibit varying memory behavior. On the other hand, a large variety of memory modules allow design implementations with a wide range of cost, performance and power profiles. The embedded system architect can thus explore and select custom memory architectures to fit the constraints of target applications and design goals. In this paper we present an overview of recent research in the area of memory architecture customization for embedded systems.

1 Introduction

The application-specific nature of embedded systems has caused a fresh look at architectural issues in recent times. Embedded systems implement a fixed application or set of related applications; consequently, the system architecture can be customized to suit the needs of the given application. This results in an architectural optimization strategy that is fundamentally different from that employed for general purpose processors. In the case of general purpose computer systems, the actual use to which the system will be put is not known, so the processors are designed for good average performance over a set of typical benchmark programs which cover a wide range of application with different behaviors. However, in the case of embedded systems, the features of the given application can be used to determine the architectural parameters. This becomes very important in modern embedded systems where power consumption is a crucial factor. For example, if an application does not use floating point arithmetic, then the floating point unit can be removed from the processor, thereby saving area and power in the implementation.

© Springer-Verlag Berlin Heidelberg 2002

The memory subsystem is an important and interesting component of system designs that can benefit from customization. Unlike a general purpose processor where a standard cache hierarchy is employed, the memory hierarchy of embedded systems can be tailored in various ways. The memory can be selectively cached; the cache line size can be determined by the application; the designer can opt to discard the cache completely and choose specialized memory configurations such as FIFOs and stream buffers; and so on. The exploration space of different possible memory architectures is vast, and there have been attempts to automate or semi-automate this exploration process [1].

Memory issues have been separately addressed by disparate research groups: computer architects, compiler writers, and the CAD/embedded systems community.

Memory architectures have been studied extensively by computer architects. Memory hierarchy, implemented with cache structures, has received considerable attention from researchers. Cache parameters such as line size, associativity, and write policy, and their impact on typical applications have been studied in detail [2]. Recent studies have also quantified the impact of dynamic memory (DRAM) architectures [3].

Since architectures are closely associated with compilation issues, compiler researchers have addressed the problem of generating efficient code for a given memory architecture by appropriately transforming the program and data. Compiler transformations such as blocking/tiling are examples of such optimizations [4, 5].

Finally, researchers in the area of CAD/embedded systems have typically employed memory structures such as register files, static memory (SRAMs), and DRAMs in generating application specific designs.

While the optimizations identified by the architecture and compiler community are still applicable in embedded system design, the architectural flexibility available in the new context adds a new exploration dimension. To be really effective, these optimizations need to be integrated into the embedded system design process as well as enhanced with new optimization and estimation techniques.

We first present an overview of different memory architectures used in embedded systems, and then survey some of the ways in which these architectures have been customized.

2 Embedded Memory Architectures

In this section we outline some common architectures used in embedded memories, with particular emphasis on the more unconventional ones.

2.1 Caches

As application-specific systems became large enough to use a processor core as a building block, the natural extension in terms of memory architecture was the addition of instruction and data caches. Since the organization of typical caches is well known [2] we will omit the explanation. Caches have many parameters which can be customized for a given application. Some of these customizations are described in Section 3.

2.2 Scratch-Pad Memory

An embedded system designer is not restricted to using only a traditional cached memory architecture. Since the design needs to execute only a single application, we can use unconventional architectural variations that suit the specific application under consideration. One such design alternative is *Scratch Pad memory* [6, 1].

Scratch-Pad memory refers to data memory residing on-chip, that is mapped into an address space disjoint from the off-chip memory, but connected to the same address and data buses. Both the cache and Scratch-Pad memory (usually SRAM) allow fast access to their residing data, whereas an access to the off-chip memory requires relatively longer access times. The main difference between the Scratch-Pad SRAM and data cache is that the SRAM guarantees a single-cycle access time, whereas an access to the cache is subject to cache misses. The concept of Scratch Pad memory is an important architectural consideration in modern embedded systems, where advances in embedded DRAM technology have made it possible to combine DRAM and logic on the same chip. Since data stored in embedded DRAM can be accessed much faster and in a more power-efficient manner than that in off-chip DRAM, a related optimization problem that arises in this context is how to identify critical data in an application, for storage in on-chip memory.

Figure 1 shows the architectural block diagram of an application employing a typical embedded core processor [1], where the parts enclosed in the dotted rectangle are implemented in one chip, interfacing with an off-chip memory, usually realized with DRAM. The address and data buses from the *CPU core* connect to the *Data Cache, Scratch-Pad memory*, and the *External Memory Interface* (EMI) blocks. On a memory access request from the CPU, the data cache indicates a cache hit to the EMI block through the **C_HIT** signal. Similarly, if the SRAM interface circuitry in the Scratch-Pad memory determines that the referenced memory address maps into the on-chip SRAM, it assumes control of the data bus and indicates this status to the EMI through signal **S_HIT**. If both the cache and SRAM report misses, the EMI transfers a block of data of the appropriate size (equal to the *cache line size*) between the cache and the DRAM.

One possible data address space mapping for this memory configuration is shown in Figure 2, for a sample addressable memory of size N data words. Memory addresses $0 \dots (P-1)$ map into the on-chip scratch pad memory, and have a single processor cycle access time. Memory addresses $P \dots (N-1)$ map into the off-chip DRAM, and are accessed by the CPU through the data cache. A cache hit for an address in the range $P \dots N-1$ results in a single-cycle delay, whereas a cache miss, which leads to a block transfer between off-chip and cache

[1] For example, the LSI Logic CW33000 RISC Microprocessor core [7].

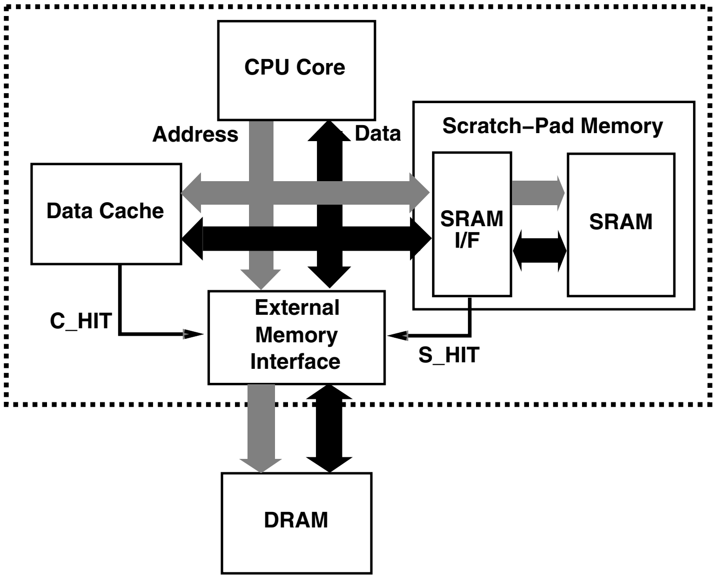


Fig. 1. Block Diagram of Typical Embedded Processor with Scratch-Pad Memory

memory, may result in a delay of say 10-20 processor cycles. We illustrate the use of this Scratch-Pad memory with the following example.

Example 1. A small (4×4) matrix of coefficients, *mask*, slides over the input image, *source*, covering a different 4×4 region in each iteration of y, as shown in Figure 3. In each iteration, the coefficients of the mask are combined with the region of the image currently covered, to obtain a weighted average, and the result, *acc*, is assigned to the pixel of the output array, *dest*, in the center of the covered region. If the two arrays *source* and *mask* were to be accessed through the data cache, the performance would be affected by cache conflicts. This problem can be solved by storing the small *mask* array in the Scratch-pad memory. This assignment eliminates all conflicts in the data cache – the data cache is now used for memory accesses to *source*, which are very regular. Storing *mask* on-chip, ensures that frequently accessed data is never ejected off-chip, thereby significantly improving the memory performance and energy dissipation.

The memory assignment described in [6] exploits this architecture by first determining a *Total Conflict Factor* (TCF) for each array based on the access frequency and possibility of conflict with other arrays, and then considering the arrays for assignment to scratch pad memory in the order of TCF/(array size), giving priority to high-conflict/small-size arrays.

Dynamic Data Transfers In the above formulation, the data stored in the Scratch-Pad Memory was statically determined. This idea can be extended to the

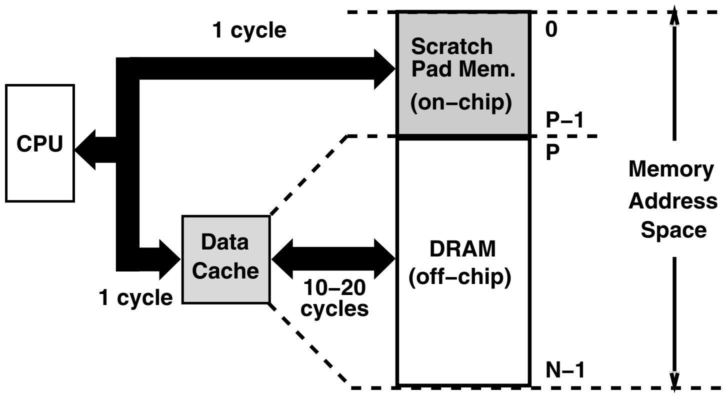


Fig. 2. Division of Data Address Space between Scratch Pad memory and off-chip memory

case of dynamic data storage. However, since there is no automatic hardware-controlled mechanism to transfer data between the scratch pad and the main memory, such transfers have to be explicitly managed by the compiler. In the technique proposed in [8], the tiling compiler optimization is modified by first moving the data tiles into scratch pad memory and moving it back to main memory after the computation is complete.

Storing Instructions in Scratch-Pad Memory A scratch-pad memory storing a small amount of frequently accessed data on-chip, has an equivalent in the instruction cache. The idea of using a small buffer to store blocks of frequently used instructions was first introduced in [9]. Recent extensions of this strategy are the Decoded Instruction Buffer [10] and the L-cache [11].

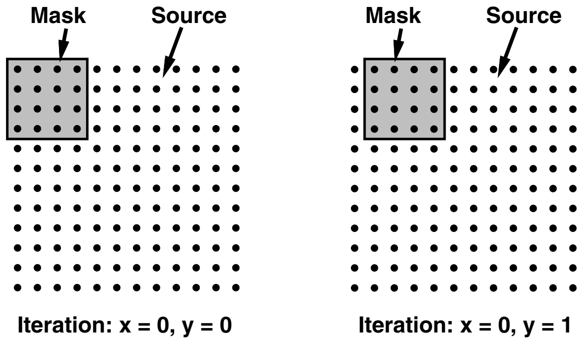
Researchers have also examined the possibility of storing both instructions and data in the scratch pad memory. In the formulation proposed in [12], the frequency of access of both data and program blocks are analyzed and the most frequently occurring among them are assigned to the scratch pad memory.

2.3 DRAM

DRAMs have been used in a processor-based environment for quite some time, but the context of its use in embedded systems – both from a hardware synthesis viewpoint, as well as from an embedded compiler viewpoint – have been investigated relatively recently. DRAMs offer better memory performance through the use of specialized access modes that exploit the internal structure and steering/buffering/banking of data within these memories. Explicit modeling of these specialized access modes allows the incorporation of such high-performance access modes into synthesis and compilation frameworks. New synthesis and compilation techniques have been developed that employ detailed knowledge of the

```
# define N 128
# define M 4
# define NORM 16
int source[N][N], dest [N][N];
int mask [M][M];
int acc, i, j, x, y;
⋮
for (x = 0; x < N − M; x++)
    for (y = 0; y < N − M; y++) {
        acc = 0;
        for (i = 0; i < M; i++)
            for (j = 0; j < M; j++)
                acc = acc + source[x+i][y+j] * mask[i][j];
        dest[x+M/2][y+M/2] = acc/NORM;
    }
```

(a)



(b)

Fig. 3. (a)Procedure CONV (b) Memory access pattern in CONV

DRAM access modes and exploit advance knowledge of an embedded system's application to better improve system performance and power.

A typical DRAM memory address is internally split into a *row address* consisting of the most significant bits and a *column address* consisting of the least significant bits. The row address selects a page from the core storage and the column address selects an offset within the page to arrive at the desired word. When an address is presented to the memory during a READ operation, the entire page addressed by the row address is read into the page buffer, in anticipation of spatial locality. If future accesses are to the same page, then there is no need to access the main storage area since it can just be read off the page buffer, which acts like a cache. Thus, subsequent accesses to the same page are very fast.

A scheme for modeling the various memory access modes and using them to perform useful optimizations in the context of a Behavioral Synthesis environment is described in [13]. The main observation is that the input behavior's memory access patterns can potentially exploit the page mode (or other specialized access mode) features of the DRAM. The key idea is the representation of these specialized access modes as graph primitives that model individual DRAM access modes such as row decode, column decode, precharge, etc.; each DRAM family's specialized access modes are then represented using a composition of these graph primitives to fit the desired access mode protocol. These composite graphs can then be scheduled together with the rest of the application behavior, both in the context of synthesis, as well as for code compilation. For instance, some additional DRAM-specific optimizations discussed in [13] are:

Read-Modify-Write (R-M-W) Optimization that takes advantage of the R-M-W mode in modern DRAMs which provides support for a more efficient realization of the common case where a specific address is read, the data is involved in some computation, and then the output is written back to the same location.

Hoisting where the row-decode node is scheduled ahead of a conditional node if the first memory access in both branches are on the same page.

Unrolling optimization in the context of supporting the page mode accesses.

A good overview of the performance implications of the architectural features of modern DRAMs is found in [3].

Synchronous DRAM As DRAM architectures evolve, new challenges are presented to the automatic synthesis of embedded systems based on these memories. *Synchronous DRAM* represents an architectural advance that presents another optimization opportunity: multiple memory banks. The core memory storage is divided into multiple banks, each with its own independent page buffer, so that two separate memory pages can be simultaneously active in the multiple page buffers.

The problem of modeling the access modes of synchronous DRAMs is addressed in [14]. The modes include:

Burst mode read/write – fast successive accesses to data in the same page.

Interleaved row read/write modes – alternating burst accesses between banks.

Interleaved Column Access – alternating burst accesses between two chosen rows in different banks.

Memory bank assignment is performed by creating an interference graph between arrays and partitioning it into subgraphs so that data in each part is assigned to a different memory bank. The bank assignment algorithm is related to techniques such as [15] that address memory assignment for DSP processors such as the Motorola 56000 which has a dual-bank internal memory/register file [16, 17]. The bank assignment problem in [15] is targeted at scalar variables, and is solved in conjunction with register allocation by building a constraint graph that models the data transfer possibilities between registers and memories followed by a simulated annealing step.

[18] approached the SDRAM bank assignment problem by first constructing an *array distance table*. This table stores the *distance* in the DFG between each pair of arrays in the specification. A short distance indicates a strong correlation, possibly indicating that they might be, for instance, two inputs of the same operation, and hence, would benefit from being assigned to separate banks. The bank assignment is finally performed by considering array pairs in increasing order of their array distance information.

Whereas the previous discussion has focused primarily in the context of hardware synthesis, similar ideas have been employed to aggressively exploit the memory access protocols for compilers [19, 20]. In the traditional approach of compiler/architecture codesign, the memory subsystem was separated from the microarchitecture; the compiler typically dealt with memory operations using the abstractions of memory loads and stores, with the architecture (e.g., the memory controller) providing the interface to the (typically yet-unknown) family of DRAMs and other memory devices that would deliver the desired data. However in an embedded system, the system architect has advance knowledge of the specific memories (e.g., DRAMs) used; thus we can employ *memory-aware* compilation techniques [19] that exploit the specific access modes in the DRAM protocol to perform better code scheduling. In a similar manner, it is possible for the code scheduler to employ global scheduling techniques to hide potential memory latencies using knowledge of the memory access protocols, and in effect, improve the ability of the memory controller to boost system performance [20].

2.4 Special Purpose Memories

In addition to the general memories such as caches, and memories specific to embedded systems, such as scratch-pad, there exist various other types of custom memories that implement specific access protocols. Such memories include:

- LIFO (memory implementing Last-In-First-Out protocol).
- FIFO (memory implementing queue or First-In-First-Out protocol).
- CAM (content addressable memory).

3 Customization of Memory Architectures

We now survey some recent research efforts that address the exploration space involving on-chip memories. A number of distinct memory architectures could be devised to exploit different application specific memory access patterns efficiently. Even if we restrict the scope of the architecture to those involving on-chip memory only, the exploration space of different possible configurations is too large, making it infeasible to exhaustively simulate the performance and energy characteristics of the application for each configuration. Thus, exploration tools are necessary for rapidly evaluating the impact of several candidate architectures. Such tools can be of great utility to a system designer by giving fast initial feedback on a wide range of memory architectures [1].

3.1 Caches

Two of the most important aspects of data caches that can be customized for an application are: (1) the cache line size; and (2) the cache size. The customization of cache line size for an application is performed in [21] using an estimation technique for predicting the memory access performance – that is, the total number of processor cycles required for all the memory accesses in the application.

There is a tradeoff in sizing the cache line. If the memory accesses are very regular and consecutive, i.e, exhibit spatial locality, a longer cache line is desirable, since it minimizes 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. The maximum size of a cache line is the DRAM page size.

The estimation technique uses data reuse analysis to predict the total number of cache hits and misses inside loop nests so that spatial locality is incorporated into the estimation. An estimate of the impact of conflict misses is also incorporated. The estimation is carried out for the different candidate line sizes and the best line size is selected for the cache.

The customization of the total cache size is integrated into the scratch pad memory customization described in the next section.

3.2 Scratch-Pad Memory

MemExplore [21], an exploration framework for optimizing the on-chip data memory organization addresses the following problem: given a certain amount of on-chip memory space, partition this into data cache and scratch pad memory so that the total access time and energy dissipation is minimized, i.e., the number of accesses to off-chip memory is minimized. In this formulation, an on-chip memory architecture is defined as a combination of the total size of on-chip memory used for data storage; the partitioning of this on-chip memory into: scratch memory, characterized by its size; and data cache, characterized by the cache size; and the cache line size. For each candidate on-chip memory size T, the technique considers different divisions of T into cache (size C) and scratch pad memory (size $S = T - C$), selecting only powers of 2 for C. The procedure described in Section 2.2 is used to identify the right data for storage in scratch pad memory. Among the data assigned to be stored in off-chip memory (and hence accessed through the cache), an estimation of the memory access performance is performed by combining an analysis of the array access patterns in

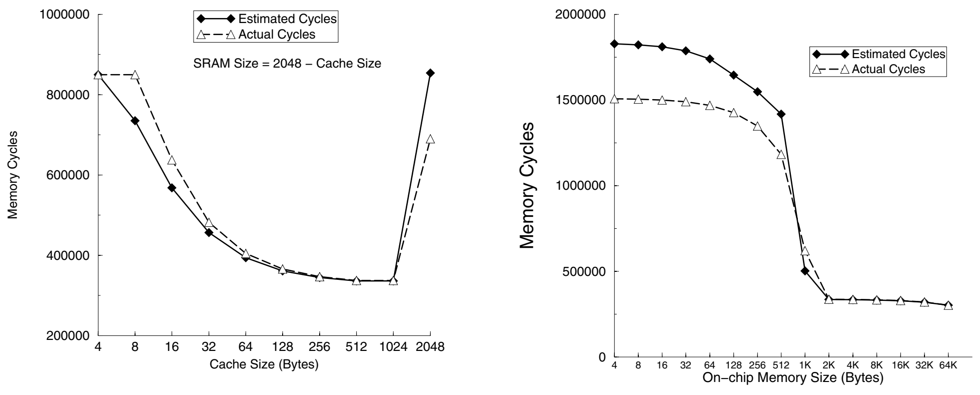


Fig. 4. *Histogram* Example (a) Variation of memory performance with different mixes of cache and Scratch-pad memory, for total on-chip memory of 2 KB (b) Variation of memory performance with total on-chip memory space

the application and an approximate model of the cache behavior. The result of the estimation is the expected number of processor cycles required for all the memory accesses in the application. For each T, the (C, L) pair that is estimated to maximize performance is selected.

Example 2. Typical exploration curves of the MemExplore algorithm are shown in Figure 4. Figure 4(a) shows that the ideal division of a 2K on-chip space is 1K scratch pad memory and 1K data cache. Figure 4(b) shows that very little performance improvement is observed beyond a total on-chip memory size of 2KB.

The exploration curves of Figure 4 are generated from fast analytical estimates, which are three orders of magnitude faster than actual simulations, and are independent of data size. This estimation capability is important in the initial stages of system design, where the number of possible architectures is large, and a simulation of each architecture is prohibitively expensive.

3.3 DRAM

The presence of embedded DRAMs adds several new dimensions to traditional architecture exploration. One interesting aspect of DRAM architecture that can be customized for an application is the banking structure.

Figure 5(a) illustrates a common problem with the single-bank DRAM architecture. If we have a loop that accesses in succession data from three large arrays A, B, and C, each of which is much larger than a page, then each memory access leads to a fresh page being read from the storage, effectively cancelling the benefits of the page buffer. This page buffer interference problem cannot be avoided if a fixed architecture DRAM is used. However, an elegant solution

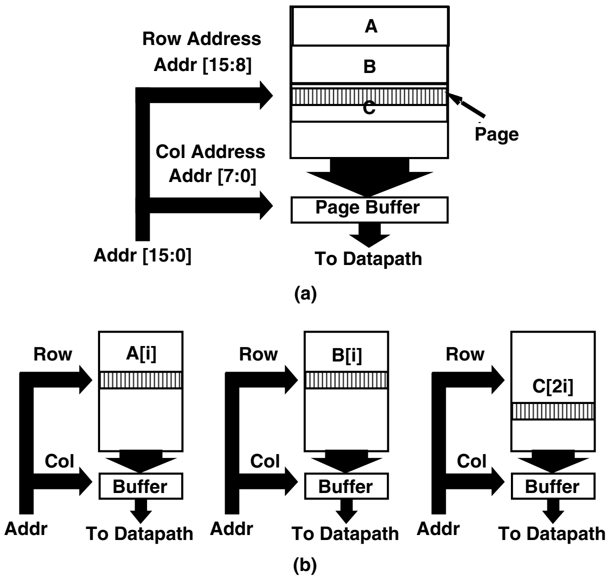


Fig. 5. (a) Arrays mapped to a single-bank memory (b) 3-bank memory architecture

to the problem is available if the banking configuration of the DRAM can be customized for the application [22]. Thus, in the example of Figure 5, the arrays can be assigned to separate banks as shown in Figure 5(b). Since each bank has its own private page buffer, there is no interference between the arrays, and the memory accesses do not represent a bottleneck.

In order to customize the banking structure for an application, we need to solve the memory bank assignment problem – determine an optimal banking structure (number of banks) and determine the assignment of each array variable into the banks such that the number of page misses is minimized. This objective optimizes both the performance as well as the energy dissipation of the memory subsystem. The memory bank customization problem is solved in [22] by modeling the assignment as a partitioning problem – partition a given set of nodes into a given number of groups such that a given criterion (bank misses in this case) is optimized. The partitioning proceeds by associating a cost of assigning two arrays into the same bank, determined by the number of accesses to the arrays and the loop count. If the arrays are accessed in the same loop, then the cost is high, thereby discouraging the partitioning algorithm from assigning them to the same bank. On the other hand, if two arrays are never accessed in the same loop, then they are candidates for assignment into the same bank. This pairing is associated with a low cost, guiding the partitioner to assign the arrays together.

3.4 Multiple SRAMs

In a custom memory architecture, the designer can choose memory parameters such as the number of memories, and the size, and number of ports on each memory.

The number of memory modules used in a design has a significant impact on the access times and power consumption. A single large monolithic memory to hold all the data is expensive in terms of both access time and energy dissipation than multiple memories of smaller size. However, the other extreme, where all array data is stored in distinct memory modules, is also expensive, and the optimal allocation lies somewhere in between.

The memory allocation problem is closely linked to the problem of assigning array data to the individual memory modules. Arrays need to be clustered into memories based on their accesses [23]. The clustering can be vertical (different arrays occupy different memory words) or horizontal (different arrays occupy different bit positions within the same word) [24]. Parameters such as bit-width, word count, and number of ports can be included in this analysis [25]. The required memory bandwidth (number of ports allowing simultaneous access) can be formally determined by first building a conflict graph of the array accesses and storing in the same memory module the arrays that do not conflict [26].

3.5 Special Purpose Memories

Special purpose memories such as stacks (LIFO), queues (FIFO), frame buffers, streaming buffers, etc. can be utilized when customizing the memory architecture for an application. Indeed, analysis of many large applications shows that a significant number of the memory references in data-intensive applications are made by a surprisingly small number of lines of code. Thus it is possible to customize the memory subsystem by tuning the memories for these segments of code, with the goal of improving performance, and also for reducing the power dissipation. In the approach described in [27], the application is first analyzed and different access patterns identified. Data for the most *critical* access patterns are assigned to memory modules that best fit the access pattern profiles. The system designer can then evaluate different cost/performance/power profiles for different realizations of the memory subsystem.

3.6 Processor-Memory Coexploration

Datapath Width and Memory Size The CPU's bit-width is an additional parameter that can be tuned during architectural exploration of customizable processors. [28] studied the relationship between the width of the processor data path and the memory subsystem. This relationship is important when different data types with different sizes are used in the application. The key observation made is that as datapath width is decreased, the data memory size decreases because of less wasted space. For example, storing 3-bit data in a 4-bit word instead of 8-bit word), but the instruction memory might increase. For example, storing 7-bit data in an 8-bit word requires only one instruction to access it, but requires two instructions if a 4-bit datapath is used. The authors use a RAM and ROM cost model to evaluate the cost of candidate bit-widths in a combined CPU-memory exploration.

Architectural Description Language (ADL) Driven Coexploration Processor Architecture Description Languages (ADLs) have been developed to allow for a language-driven exploration and software toolkit generation approach [29, 30]. Currently most ADLs assume an implicit/default memory organization, or are limited to specifying the characteristics of a traditional memory hierarchy. Since embedded systems may contain non-traditional memory organizations, there is a great need to model explicitly the memory subsystem for an ADL-driven exploration approach. A recent approach [31] describes the use of the EXPRESSION ADL [32] to drive Memory Architecture Exploration. The EXPRESSION ADL description of the processor-memory architecture is used to explicitly capture the memory architecture, including the characteristics of the memory modules (such as caches, DRAMs, SRAMs DMAs), the parallelism and pipelining present in the memory architecture (e.g., resources used, timings, access modes). Each such explicit memory architecture description is then used to automatically generate the information needed by the compiler [19, 20] to efficiently utilize the features in the memory architecture, and to generate a memory simulator, allowing feedback to the designer on the match between the application, the compiler and the memory architecture.

3.7 Split Spatial and Temporal Caches

Various specialized memory structures proposed over the years could be candidates for embedded systems. One such concept is split spatial/temporal caches.

Variables in real life applications present a wide variety of access patterns and locality types (for instance scalars, such as indexes, present usually high temporal and moderate spatial locality, while vectors with small stride present high spatial locality, and vectors with large stride present low spatial locality, and may or may not have temporal locality). Several approaches including [33] have proposed splitting a cache into a spatial cache and a temporal cache that store data structures with high temporal and high spatial locality respectively. These approaches rely on a dynamic prediction mechanism to route the data to either the spatial or the temporal caches, based on a history buffer. In an embedded system context, the approach of [34] uses a similar split-cache architecture, but allocates the variables statically to the different local memory modules, avoiding the power and area overhead of the dynamic prediction mechanism. Thus by targeting the specific locality types of the different variables, better utilization of the main memory bandwidth is achieved. The useless fetches due to locality mismatch are thus avoided. For instance, if a variable with low spatial locality is serviced by a cache with a large line size, a large number of the values read from the main memory will never be used. The approach in [34] shows that the memory bandwidth and memory power consumption can be reduced significantly.

4 Conclusion

The significant interest in embedded systems in recent times has caused researchers to study architectural optimizations from a new angle. With the full advance knowledge of the applications being implemented by the system, many design parameters can be customized. This is especially true of the memory subsystem where a vast array of different organizations can be employed for application specific systems and the designer is not restricted to the traditional cache hierarchy. The optimal memory architecture for an application specific system can be significantly different from the typical cache hierarchy of processors. We outlined different memory architectures relevant to embedded systems and strategies to customize them for a given application. While some of the analytical techniques are automated, a lot of work still remains to be performed in the coming years before a completely push-button methodology evolves for application-specific customization of the memory organization in embedded systems.

References

[1] Panda, P. R., Dutt, N. D., Nicolau, A.: Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration. Kluwer Academic Publishers, Norwell, MA (1999) 648, 649, 655
[2] Hennessy, J. L., Patterson, D. A.: Computer Architecture – A Quantitative Approach. Morgan Kaufman, San Francisco, CA (1994) 648, 649
[3] Cuppu, V., Jacob, B. L., Davis, B., Mudge, T. N.: A performance comparison of contemporary dram architectures. In: International Symposium on Computer Architecture, Atlanta, GA (1999) 222–233 648, 653
[4] Lam, M., Rothberg, E., Wolf, M. E.: The cache performance and optimizations of blocked algorithms. In: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. (1991) 63–74 648
[5] Panda, P. R., Nakamura, H., Dutt, N. D., Nicolau, A.: Augmenting loop tiling with data alignment for improved cache performance. IEEE Transactions on Computers **48** (1999) 142–149 648
[6] 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** (2000) 682–704 649, 650
[7] LSI Logic Corporation Milpitas, CA: CW33000 MIPS Embedded Processor User's Manual. (1992) 649
[8] Kandemir, M. T., Ramanujam, J., Irwin, M. J., Vijaykrishnan, N., Kadayif, I., Parikh, A.: Dynamic management of scratch-pad memory space. In: Design Automation Conference, Las Vegas, NV (2001) 690–695 651
[9] Jouppi, N. P.: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In: International Symposium on Computer Architecture, Seattle, WA (1990) 364–373 651

[10] 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** (1997) 417–424 651

[11] 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** (2000) 317–326 651

[12] Steinke, S., Wehmeyer, L., Lee, B. S., Marwedel, P.: Assigning program and data objects to scratchpad for energy reduction. In: Design, Automation & Test in Europe, Paris, France (2002) 651

[13] Panda, P. R., Dutt, N. D., Nicolau, A.: Incorporating DRAM access modes into high-level synthesis. IEEE Transactions on Computer Aided Design **17** (1998) 96–109 653

[14] Khare, A., Panda, P. R., Dutt, N. D., Nicolau, A.: High-level synthesis with SDRAMs and RAMBUS DRAMs. IEICE Transactions on fundamentals of electronics, communications and computer sciences **E82-A** (1999) 2347–2355 653

[15] Sudarsanam, A., Malik, S.: Simultaneous reference allocation in code generation for dual data memory bank asips. ACM Transactions on Design Automation of Electronic Systems **5** (2000) 242–264 653, 654

[16] Saghir, M. A. R., Chow, P., Lee, C. G.: Exploiting dual data-memory banks in digital signal processors. In: International conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA (1996) 234–243 654

[17] Cruz, J. L., Gonzalez, A., Valero, M., Topham, N.: Multiple-banked register file architectures. In: International Symposium on Computer Architecture, Vancouver, Canada (2000) 315–325 654

[18] Chang, H. K., Lin, Y. L.: Array allocation taking into account SDRAM characteristics. In: Asia and South Pacific Design Automation Conference, Yokohama (2000) 497–502 654

[19] Grun, P., Dutt, N., Nicolau, A.: Memory aware compilation through accurate timing extraction. In: Design Automation Conference, Los Angeles, CA (2000) 316–321 654, 659

[20] Grun, P., Dutt, N., Nicolau, A.: MIST: An algorithm for memory miss traffic management. In: IEEE/ACM International Conference on Computer Aided Design, San Jose, CA (2000) 431–437 654, 659

[21] Panda, P. R., Dutt, N. D., Nicolau, A.: Local memory exploration and optimization in embedded systems. IEEE Transactions on Computer Aided Design **18** (1999) 3–13 655

[22] Panda, P. R.: Memory bank customization and assignment in behavioral synthesis. In: Proceedings of the IEEE/ACM International Conference on Computer Aided Design, San Jose, CA (1999) 477–481 657

[23] Ramachandran, L., Gajski, D., Chaiyakul, V.: An algorithm for array variable clustering. In: European Design and Test Conference, Paris (1994) 262–266 658

[24] Schmit, H., Thomas, D. E.: Synthesis of application-specific memory designs. IEEE Transactions on VLSI Systems **5** (1997) 101–111 658

[25] Jha, P. K., Dutt, N.: Library mapping for memories. In: European Design and Test Conference, Paris, France (1997) 288–292 658

[26] Wuytack, S., Catthoor, F., Jong, G. D., Man, H. D.: Minimizing the required memory bandwidth in vlsi system realizations. IEEE Transactions on VLSI Systems **7** (1999) 433–441 658

[27] Grun, P., Dutt, N., Nicolau, A.: Apex: Access patter based memory architecture customization. In: Proceedings International Symposium on System Synthesis, Montreal, Canada (2001) 25–32 658

[28] Shackleford, B., Yasuda, M., Okushi, E., Koizumi, H., Tomiyama, H., Yasuura, H.: Memory-cpu size optimization for embedded system designs. In: Design Automation Conference. (1997) 658

[29] Tomiyama, H., Halambi, A., Grun, P., Dutt, N., Nicolau, A.: Architecture description languages for systems-on-chip design. In: Proceedings 6th Asia Pacific Conference on Chip Design Languages, Fukuoka (1999) 109–116 659

[30] Halambi, A., Grun, P., Tomiyama, H., Dutt, N., Nicolau, A.: Automatic software toolkit generation for embedded systems-on-chip. In: Proceedings ICVC'99, Korea (1999) 659

[31] Mishra, P., Grun, P., Dutt, N., Nicolau, A.: Processor-memory co-exploration driven by a memory-aware architecture description language. In: VLSIDesign, Bangalore (2001) 659

[32] Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., Nicolau, A.: Expression: A language for architecture exploration through compiler/simulator retargetability. In: Proceedings DATE'99, Munich, Germany (1999) 659

[33] Gonzales, A., Aliagas, C., Valero, M.: A data cache with multiple caching strategies tuned to different types of locality. In: International Conference on Supercomputing, Barcelona, Spain (1995) 338–347 659

[34] Grun, P., Dutt, N., Nicolau, A.: Access pattern based local memory customization for low power embedded systems. In: Design, Automation, and Test in Europe, Munich (2001) 659