A Recursive Algorithm for Low-Power Memory Partitioning

Luca Benini* Alberto Macii** Massimo Poncino**

*Universita di Bologna, Bologna, Italy 40136

**Politecnico di Torino, Torino, Italy 10129

Abstract

Memory-processor integration offers new opportunities for reducing the energy of a system. In the case of embedded systems, one solution consists of mapping the most frequently accessed addresses onto the on-chip SRAM to guarantee power and performance efficiency. This option is especially effective when memory access patterns can be profiled and studied at design time (as in typical real-time embedded systems).

In this work, we propose an algorithm for the automatic partitioning of on-chip SRAM in multiple banks that can be independently accessed. Starting from the dynamic execution profile of an embedded application running on a given processor core, we synthesize a multi-banked SRAM architecture optimally fitted to the execution profile. The algorithm provides a globally optimum solution to the problem under realistic assumptions on the power cost metrics, and with constraints on the number of memory banks.

Results, collected on a set of embedded applications for the ARM processor, have shown average energy savings around 42%.

1 Introduction

An increasingly large fraction of today's embedded System-on-Chips (SoCs) employs core processors as basic computational units. Processors are highly flexible; hence, their design time and cost can be amortized by re-using them in a wide variety of applications. Unfortunately, this decisive advantage of core processors is counter-balanced by some drawbacks. Most notably, processors are power-inefficient with respect to dedicated architectures [1]. This inefficiency is a fundamental consequence of the trade-off between flexibility and power that must be explored at every step of the design flow.

One of the key issues in the design of energy-efficient processor-based architectures for embedded systems is the power dissipated by memories. Processors are very demanding in terms of memory: They require memories for both manipulating data and fetching instructions. Several authors have pointed out that the power consumed in memories (and memory-related activities) can take a dominant fraction of the power budget of an embedded system for data-dominated applications [2]. Many power optimization approaches specifically tackle the memory power challenge in SoC design. The common denominator in most memory energy minimization techniques is to dedicate silicon real estate to memory and to integrate a memory array on the same die as the processor.

Accessing on-chip memory is much faster and power-efficient than relying exclusively on off-chip memories [3, 4].

The possibility of integrating processor and memory onto the same chip offers new opportunities for energy-efficient design. Memory size and organization can be tailored to application requirements, and application-specific memory architectures can be developed to minimize memory power for a given embedded application. Since hand-crafting application-specific components for every design can be excessively time-consuming, memory power optimization should be automated. The task of memory power optimization tools for embedded systems is to create low-power memory architectures customized for a specific core *and* a specific system functionality (i.e., an embedded application). This is in sharp contrast with general-purpose systems, where the main objective of memory architecture design is robustness and flexibility.

On-chip caches are perhaps the best known architectural optimization technique in memory design. Caches exploit the principle of locality in memory access patterns [5], and provide a flexible way to store most frequently accessed memory locations on-chip, where they can be efficiently read and written. In embedded systems, a valid alternative to caches is offered by on-chip SRAM, often called *scratch-pad* RAM. In this architecture, the most frequently accessed addresses are statically mapped onto scratch-pad RAM to guarantee power (and performance) efficiency. Scratch-pad RAMs are particularly useful in real-time embedded systems for data-intensive applications, where access patterns can be profiled and studied at design time, and where caches are known to perform suboptimally and to reduce predictability in real-time performance.

In this paper, we focus on automatic optimization of on-chip scratch-pad RAMs for embedded SoCs. We start from the dynamic execution profile of an embedded application running on a given processor core, and we synthesize a multi-banked SRAM architecture optimally fitted to such profile. The rationale in our approach is to partition scratch-pad memory in multiple banks that can be independently accessed. Power-per-access is reduced as the size of a memory bank is decreased. On the other hand, as the number of banks increases, there is an unavoidable hardware overhead caused by: (i) Duplication of addressing and control logic; (ii) Increased communication resources required to transfer information. Such an overhead manifests itself in increased power, access time and area that prevents arbitrarily fine partitioning. Hence, we need to find an optimal partition with a tight constraint on the maximum number of memory banks.

The theoretical contributions of this work are: (i) The formulation of the minimum-power partitioning problem with constrained number of memory banks; (ii) The development of an algorithm that finds the *globally optimum* solution to the problem under realistic assumptions on the power cost metrics. Experimental validation is carried out on a set of embedded appli-

cations for the ARM core processor. An average power reduction of 41.7% has been achieved.
The manuscript is organized as follows. Section 2 introduces and motivates memory partitioning, while Section 3 briefly reviews related work. The partitioning algorithm is described in detail in Section 4. Finally, Section 5 reports experimental results.

2 Memory Partitioning for Low Power

A static random access memory (SRAM) can be easily integrated onto the same chip as the processor and other ancillary logic circuits, because it does not require additional fabrication steps and dedicated technology. For this reason, embedded SRAMs are much more common in SoC designs than non-volatile memories and DRAMs, even if they are much less dense. In this work, we then focus on embedded SRAMs, although embedded DRAM technology is actively developed and commercially available, and it will probably become mainstream in the next few years [4].
SRAMs can be made available as hard macros by silicon vendors [6]. As an alternative, several EDA companies provide soft *RAM macro compilers* that can be tuned to a given technology, and are used by designers to automatically instantiate SRAM arrays with many different sizes and organizations [7, 8]. Due to their relatively large cell area, on-chip memory arrays are limited in size to a fraction of one megabyte. In $0.25\mu m$ technology, SRAM soft macros are generally smaller than 128 Kbytes [7] (hard macros are more densely packed, and they reach 256 Kbytes [6]). For the sake of explanation, we will assume the availability of a library of synchronous, single-ported SRAM memory cuts, with input/output data width of 32 bits (a four-bytes word).
Fortunately, the most frequently accessed addresses in many non-trivial embedded applications can fit into a relatively small memory space. We will assume that the designer (or a dedicated design tool [9]) specifies a range (a_{lo}, a_{hi}) of memory addresses that should be mapped onto the on-chip SRAM. A *dynamic access profile*, obtained through simulation, is also available. For each address $a_{lo} \leq i \leq a_{hi}$, it gives the number of reads $r(i)$ and writes $w(i)$ to the address during the execution of a sample run of the target embedded application. The profile can be obtained by standard instruction-level simulators available for all processor cores. In this work, we will consider a 32-bit ARM processor. In a traditional approach, all addresses in the range are mapped to a single SRAM memory array, the smallest array in the library which is large enough to contain the specified range, as shown in Figure 1(a).

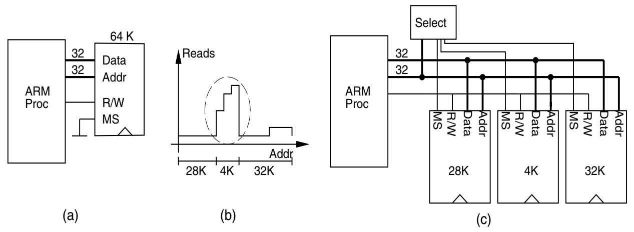

Figure 1: Memory Partitioning Example.

This solution is not optimal from the power dissipation viewpoint. Assume, for the sake of illustration, that the dynamic access profile is that shown in Figure 1(b). A small subset of the addresses in the range is very "hot" (i.e., it has large $r(i)$ values for all its addresses). A power-optimal partitioned memory organization is shown in Figure 1(c). It consists of three memories and a memory selection block. Two relatively large cuts contain the top and bottom part of the range, while the hot addresses are stored into a small memory. The average power in accessing the memory hierarchy is decreased, because a large fraction of accesses is concentrated on a small, power-efficient memory.
It is important to notice that, in our work, we are making the following assumptions:

(i) Energy per access monotonically increases with memory size;
(ii) When the processor is accessing a location whose address is outside the range contained in a memory bank, the decoder de-asserts its *memory select* (MS) input. If the MS is inactive, the bank consumes negligible power, and its data outputs are three-stated.

Also, observe that we need to account for the power consumed in the entire partitioned memory system, i.e., the address and data buses, the decoder and the control signals. These components introduce a non-negligible overhead on power consumption that must be offset by the savings given by bank partitioning. Nevertheless, we expect savings to be significant especially when the access profile is highly non-uniform and high-access addresses are clustered into small banks.
This simple example helps in clarifying why memory partitioning can be advantageous. In order to automate memory partitioning, we need to rigorously define cost metrics and search space, and to formulate an optimization algorithm that can efficiently solve realistic instances of the problem. These issues will be addressed in Section 4. In the next section, we will briefly survey related work on memory optimization.

3 Related Work

Memory partitioning for low power has been investigated by several authors in the past. Farrahi et al. [10] first studied memory partitioning to exploit sleep mode operation. This work is in the context of board-level memory optimization where memory blocks are large DRAM chips that can be powered down when they are not storing live program variables, thereby eliminating memory refresh power. Furthermore, it is assumed that activating an inactive memory incurs a significant power cost. The technique presented by Farrahi et al. tries to cluster data into memories so that memory chips are transitioned in and out of the shut-down mode as little as possible.
On-chip memory partitioning solutions have been analyzed by several authors. Su and Despain [11], Ko *et al.* [3], and Shiue and Chakrabarty [12] studied power-efficient cache organizations. They identified cache sub-banking as an effective technique to reduce cache power consumption. These works are explorative in nature and they do not provide any automatic technique to define an application-specific cache partition.
Our work is most closely related to the approach by Coumeri and Thomas [13] to embedded SRAM optimization for low power. They described a partitioned SRAM model (called *segmented configuration*), which is analogous to ours, and studied power modeling for partitioned memories in detail [14]. However, they did not propose any automatic search techniques for finding optimal partitioning: They just focused on providing design space exploration support to system designers.
Finally, a few researchers have realized that memory power can be aggressively reduced by sinergically optimizing embedded application and memory hierarchy [15, 2, 16]. Even though this

approach holds good promise, it requires a substantial amount of human intervention, and automatic optimization is restricted to very regular dataflow computations (such as array operations). In contrast, our technique can be applied to any application and data access pattern, as long as meaningful memory profiling information can be collected.

4 Optimal Recursive Partitioning

In this section, we formulate and solve the memory partitioning problem in the practical setting described in Section 2. Without loss of generality, we assume that the range of contiguous addresses mapped onto on-chip SRAM goes from 0 to $M-1$. Memory is word-addressable and the word width is 32 bits. The total memory size is then $4 \cdot M$ bytes. A hard bound, *Max*, is set on the *maximum* number of memory banks allowed in the partitioned memory architecture.

The dynamic access profile for the target embedded application is given as a pair of arrays $\mathbf{r} = [r_0, r_1, \ldots, r_{M-1}]$, $\mathbf{w} = [w_0, w_1, \ldots, w_{M-1}]$, where r_i is the number of reads to address i, and w_i is the number of writes to address i. The total energy consumed by a memory containing a given range of addresses is a technology-dependent metrics that can be expressed as a function $MemE(lo, hi, \mathbf{w}, \mathbf{r})$, where lo and hi are the maximum and minimum address in the range. Furthermore, we also define an array $\mathbf{\Delta} = [0, \delta_1, \ldots, \delta_{Max\text{-}1}]$, that expresses the energy *overhead* of adding one more bank to a partitioned memory. In other words, δ_i is the amount of additional energy that we expect to spend in selection logic and memory buses when moving from a memory organization with i banks to one with $i+1$ banks. The power savings obtained by partitioning must compensate the overhead. Clearly, the exact value of the energy overhead is not known before the memory is completely designed. Hence, $\mathbf{\Delta}$ just provides a conservative bound: It will be used to prevent partitioning when power savings are dubious.

A *memory partition* is a set of memory banks that can be independently selected. Any address $0 \leq i < M$ is stored into one and only one bank. The *total energy* consumed by a partitioned memory is the sum of the energy consumed by all its banks. Given these definitions, we are now able to formulate the memory partitioning problem:

Given $\mathbf{w}$, $\mathbf{r}$, $\mathbf{\Delta}$ *and MemE, find a partition of a M-word memory with at most Max banks that minimizes the total energy.*

Before introducing an effective solution to memory partitioning, we focus our attention on the cost metrics employed for estimating memory energy.

4.1 Cost Metrics

The cost function used to drive the partitioning process must properly evaluate the two components of *MemE*, that is, memory energy dissipation per cycle and dynamic access profile.

Memory energy dissipation per cycle requires energy models. Among those available in the literature, we have adopted the one proposed by Coumeri and Thomas [14]. Such model is empirically derived from simulation and, unlike other analytical models, it does not use technological or electrical quantities as parameters; rather, it is expressed in terms of high-level parameters such as size and bit-width.

The model consists of distinct equations for read and write operations, broken into individual expressions for the various structural components (cells, buffer, sense amps, ATD, control logic). The memory access profile for a given application can be computed using any instruction-level simulator provided with the chosen processor core. As discussed in Section 4, the distinction between read and write accesses is necessary because of the different energy cost of the two operations.

The total memory energy *MemE* is then given by the energy cost per access of a memory with given bounds, *hi, lo*, multiplied by the number of accesses to addresses within those bounds.

In formula, *MemE* is expressed as:

$$\begin{aligned} MemE(lo, hi, \mathbf{w}, \mathbf{r}) \quad = \quad & E_{\mathrm{r}}(hi - lo) \cdot \textstyle\sum_{i=lo}^{hi} \mathbf{r}[i] + \\ & E_{\mathrm{w}}(hi - lo) \cdot \textstyle\sum_{i=lo}^{hi} \mathbf{w}[i] \end{aligned}$$

where $E_{\mathrm{r}}(d)$ ($E_{\mathrm{w}}(d)$) represents the energy consumption for a read (write) access in a memory of d words.

The expression of *MemE* in Equation 4.1 is monotonically increasing with respect to the value of $(hi - lo)$, i.e., the memory size. This is because *MemE* is obtained by multiplying two functions of memory size that increase monotonically: The energy for a read/write access and the total *cumulative* read/write access counts. This monotonic behavior of the cost function is of fundamental importance for the the partitioning algorithm described in the next section.

4.2 Partitioning Algorithm

The search space of all possible memory partitions can be easily enumerated by observing that a partition is completely defined by a *cut set*, i.e., a set of addresses that identify memory bank boundaries. For a *Max*-way partition of a M-word memory, we have $Max - 1$ boundary addresses. Clearly, this number grows very rapidly with *Max*, namely, as a binomial coefficient of M over $Max - 1$. It is also easy to prove by counter-example that total energy is not a single-minimum function over the solution space: There are many local minima. These observations seem to indicate that the memory partitioning problem can be solved only with heuristic techniques, such as genetic algorithms or randomized search, that do not guarantee global optimality.

Fortunately, a careful analysis of the structure of the problem and its cost metrics reveals that it is possible to find the globally optimum solution with an algorithm that has exponential worst-case run-time (i.e., in the worst case, it exhaustively explores all possible partitions), but performs very well in practice. The algorithm finds the optimum cut by recursive bi-partitioning, and it relies on two key properties to speed-up the search:

(i) The total energy consumption of a memory bank monotonically increases with increasing memory size, if the addresses stored in a larger memory are a superset of the addresses stored in a smaller memory.

(ii) The number of memory banks, *Max*, in a partitioned architecture is much smaller than the total memory size M.

Consider the simple case of $Max = 2$ (bi-partitioning). The optimum solution can be found in $O(M)$ time by iteratively moving the lower bound, j, of the first bank from 1 to $M - 2$. The total memory energy can be computed as $TotE_2 = MemE(0, j, \mathbf{w}, \mathbf{r}) + MemE(j+1, M-1, \mathbf{w}, \mathbf{r})$. A bi-partition is considered better than the single-bank solution with energy $TotE_1$ if $TotE_2 < TotE_1 - \delta_1$.

The number of iterations can be reduced if, for a given j, we find that $MemE(0, j, \mathbf{w}, \mathbf{r}) \geq TotE_1 - \delta_1$. This early stopping condition is motivated by property (i) above: If a memory containing the range of addresses $[0, j]$ consumes more than $TotE_1 - \delta_1$, further iterations can be avoided because $MemE(0, k, \mathbf{w}, \mathbf{r}) \geq MemE(0, j, \mathbf{w}, \mathbf{r})$ for every $k > j$. The simple case of two-way partitioning indicates that property (i) can be effectively exploited to create *bounds* and prevent the exploration of search space regions that do not contain the global optimum.

4.3 Multi-Way Partitioning

The extension to multi-way partitioning leverages property (ii) of the previous section: The algorithm moves from coarse partitions to finer ones that have a larger hardware overhead. The coarse-granularity solutions are exploited to tighten the bounds on the search of fine-granularity partitions. The partitioning algorithm is therefore invoked *Max* times, to compute partitions with an increasingly larger number of blocks.

The pseudo-code of the partitioning algorithm is shown in Figure 2. Procedure `Part` receives as input the recursion index n, the current maximum depth of the recursion *Max*, the starting memory address of the current block to be partitioned *InitCut*, the current total energy *TotEnergy*, and the current energy budget *Budget*. The procedure is first invoked as `Part` $(1, 2, 0,$ *MemE(0,M,***r***,***w***)*$, 0)$, i.e., with initial budget equal to the cost of a monolithic memory of M words, and with total energy initialized to 0.

```
1  Part (n,Max,InitCut,Budget,TotEnergy) {
2    Budget -= $\delta_{n-1}$;
3    if (Budget < 0) return (0);
4    for (i = 0 to Max) {
5      CurrTotE = 0;
6      MemEnergy = MemE(InitCut,cut,r,w);
7      NewB = Budget - MemEnergy;
8      if (NewB < 0) return (0);
       else {
9        CurrTotE = TotEnergy + MemEnergy
10       if (n == Max) {
11         CurrTotE += MemE(InitCut+1,M,r,w);
12         if (CurrTotE < MinEnergy &&
13             CurrTotE < Budget) {
14           MinEnergy = CurrTotE;
15           store current solution as best solution
16           pop current last selection
           }
         } else {
17         push current selection on solution stack
18         Part(n+1,Max,InitCut+1,NewB,CurrTotE);
19         pop current last selection
         }
       }
     }
   }
```

Figure 2: Recursive Partitioning Algorithm.

The algorithm is recursive; at a given recursion depth, it computes the optimal partition (of up to *Max* blocks) of the memory portion between *Initcut* and its upper limit M.

In Line 2, the currently available power budget, *Budget*, is reduced by a factor corresponding to the energy penalty due to adding an extra memory bank. If this new budget becomes negative, no further solution can be found using n memory blocks, and execution resumes at the upper recursion level (Line 3).

If some budget is still available, we start the exploration of all possible partitions from the current cut *InitCut* to the end of the memory (Line 4). A local energy cost is initialized at each iteration (Line 5), and the cost of the partition in the generic iteration *MemEnergy* is computed using the cost function *MemE* (Line 6). The resulting cost is subtracted from the current budget, and assigned to the budget of the iteration loop *NewB* (Line 7). This "local" budget is used to restrict the search space (Line 8); the rationale here is that if the cost of the currently analyzed block ([*InitCut,i*]) exceeds the current budget, it is useless to continue with this iteration. If this is not the case, the current energy cost is added to the current total, and considered for inclusion in a solution (Line 9)

If bottom of the recursion is reached(Line 10), the current solution is completed by adding the cost of the remaining portion of memory (from the current cut to the end – Line 11). This complete solution can be stored as the new best solution if its energy cost improves the current one and it does not exceed the available budget (Lines 12 to 15). In order to continue in the iteration of Line 4, we pop the last selection from the current solution (Line 16).

The discovery of a new minimum allows us to further restrict the search space, in terms of a reduction of the total budget. This additional optimization is not shown in the pseudo-code for the sake of readability.

If the recursion can proceed, we push the current index i onto the solution stack (Line 17), and recur by adding another memory block. The current budget and current total energy of the solution built so far are forwarded to the next recursion level (Lines 18 and 19).

Although the cost function used in the algorithm is mainly an energy cost, this is not a limitation. Additional constraints (e.g., area or delay), can be easily incorporated into the algorithm; this would also help in further pruning the search space.

5 Experimental Results

The memory partitioning algorithm was tested within a design flow based on the ARM processor. The ARM core family has been specifically designed for embedded applications, and various cores are provided as intellectual property (IP) macros for integration in complex SoCs. Some cores include parts of the memory hierarchy within the bounds of the IP macro (caches and memory management units), while others just contain the basic processor, and leave to the designer the responsibility (and the freedom) of specifying the memory hierarchy. Since our purpose is to synthesize a customized partitioned memory, in our experiments we used the ARM7TDMI core that does not contain any internal memory.

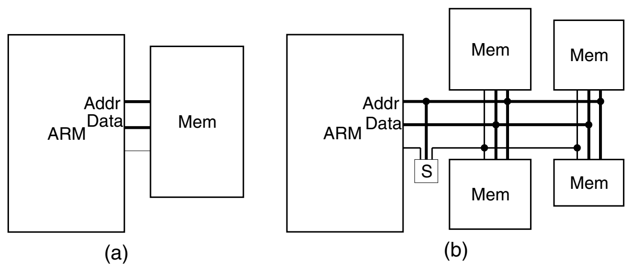

Figure 3: Floorplan Views: (a) Monolithic Memory; (b) Partitioned Memory.

The physical view (i.e., floorplan and global routing) of the processor-memory system with partitioned memory is shown in Figure 3(b): The processor is placed facing the memory system, the memory address and data buses, as well as control signals run in a channel between two sets of memory banks facing each other. Memory selection and decode logic is placed between memory and processor. The placement of unpartitioned memory and processor is shown in Figure 3(a). Notice that the length of the connections between memory and processor can be minimized in this case, because the two macro-cells can be placed one in front of the other. This simple physical model is used as a basis for specifying the values for the elements of the energy penalty array $\boldsymbol{\Delta}$, and to set the maximum number of partitions *Max*.

Benchmark	*Addr*	*Energy*			*Savings*
		Monolithic	*Best Partition*		[%]
		[nJ]	*[nJ]*	*Bank Structure*	
`AdaptFilter`	1757	2.08e-4	1.24e-4	Bank1: 109 rows x 128 columns Bank2: 166 rows x 256 columns	40.4
`Butterfly`	3820	1.01-4	6.03e-5	Bank1: 227 rows x 128 columns Bank2: 365 rows x 256 columns	40.8
`Chaos`	3696	8.18e-5	4.87e-5	Bank1: 203 rows x 128 columns Bank2: 361 rows x 256 columns	40.3
`Dft`	15584	3.94e-4	2.41e-4	Bank1: 179 rows x 128 columns Bank2: 1897 rows x 256 columns	38.8
`Dhry`	14781	2.56e-4	1.47e-4	Bank1: 454 rows x 256 columns Bank2: 1395 rows x 256 columns	42.7
`FilterBank`	8829	7.20e-4	4.12e-4	Bank1: 165 rows x 256 columns Bank2: 940 rows x 256 columns	42.6
`IirDemo`	988	1.61e-5	8.90e-6	Bank1: 89 rows x 64 columns Bank2: 32 rows x 32 columns Bank3: 195 rows x 128 columns	44.6
`Integrator`	4211	3.17e-4	1.87e-4	Bank1: 232 rows x 128 columns Bank2: 411 rows x 256 columns	40.9
`Interp`	4025	4.57e-4	2.56e-4	Bank1: 223 rows x 128 columns Bank2: 392 rows x 256 columns	43.8
`Scramble`	3873	4.50e-4	2.57e-5	Bank1: 145 rows x 128 columns Bank2: 412 rows x 256 columns	42.8
Average					41.7

Table 1: Energy Comparison of Monolithic and Partitioned Memories.

In our experiments, $Max = 4$ (i.e., we allowed a maximum of two memory banks on each side of the bus). This is a conservative bound on partitioning, which is likely to be acceptable by most designers. Array $\mathbf{\Delta}$ is set to $\mathbf{\Delta} = [0, 0.2, 0.15, 0.10]$. Element $\delta_1 = 0.2$ (i.e., 20% of the energy consumed by the monolithic memory) is the largest because we expect a sizable penalty in moving from the unpartitioned memory to the partitioned solution. We need to create the selection control logic, to place the memory banks around the bus channel, and to route the bus and control wires. $\delta_2 = 0.15$ is still fairly large because, in moving from two banks to three banks, we need to add a bus stub to the right of the first pair of banks. Finally, $\delta_3 = 0.10$ is the smallest, because the fourth bank just fills the "empty slot" below (or above) the third memory.

This simplified physical model is obviously just one of the many possible choices. The optimization algorithm is completely independent from it. Furthermore, notice that we set the values of $\mathbf{\Delta}$ in a conservative fashion. In a design flow based on state-of-the art floorplanning, global routing and placement tools, the penalties can be tightened and, possibly, more aggressive partitioning ($Max > 4$) could be considered. Again, this outlines the flexibility of our algorithm in adapting to different physical design scenarios and designer confidence levels.

We have applied the memory partitioning algorithm to a set of C benchmarks that represent typical embedded applications, and that are distributed along with `Ptolemy` [17], a simulation framework for HW/SW descriptions. `ARMulator` [18], a software emulator for core processors of the ARM family, has then been used to trace memory accesses

Results of the experiments are collected in Table 1. Column *Addr* shows the number of distinct addresses used by each application. Column *Monolithic* gives the energy cost of a single memory block that contains all the program addresses. Column *Best Partition* shows the results of the application of the partitioning algorithm. In particular, column *Energy* gives the total energy consumption of the partitioned memory, while column *Bank Structure* provides the details on how the various memory banks are organized.

Energy figures are obtained using the formula for *MemE* discussed in Section 4.1, that is, as the product of the number of memory accesses of a given program and the cost per access, as returned by the model of [14].

Finally, column *Savings* gives the percentage energy savings of the optimal solution with respect to the monolithic one.

The results are highly satisfactory, since the average energy savings is of 41.7% (maximum 44.6%) over the 10 benchmarks. Notice that the energy figures also include the wiring and logic energy overhead given by $\mathbf{\Delta}$.

The execution time is obviously proportional to the number of distinct addresses used by the program, and ranges from about two minutes for the smallest benchmarks to about three hours for the largest one (`Dft`). It is key mentioning that this relatively large execution time provides an optimal solution, and not just a heuristic local minimum. Moreover, it represents a one-time cost because, for given a memory model and access profile, the optimizer has to be run only once. Finally, the algorithm runs with the smallest possible granularity, i.e, a single memory word. In other terms, we do not restrict the sizes of the memory banks a-priori (e.g., multiples of a minimum number of words). This choice is the only one that is guaranteed to find the global optimum in the most general case. Obviously, discretizing the search space with a minimum cut size would sensibly cut run times, at the expense of the optimality.

From the table we notice that all the benchmarks but one have an optimal solution consisting of two blocks. This is indeed due to the quite conservative value of $\mathbf{\Delta}$ used in the experiments. Consider, for example, that partitioning into three memory blocks is estimated to have an overhead cost of 35% of the cost of the monolithic memory block ($\delta_1 + \delta_2$).

To quantify the impact of the $\mathbf{\Delta}$ we have run two additional experiments. In the first one, we have compared the values of Table 1, obtained with a value of $\mathbf{\Delta}$ referred to as $\mathbf{\Delta}_0$, to the values obtained with a less conservative value of $\mathbf{\Delta}$, namely $\mathbf{\Delta}_{(-0.5)} = [0.0, 0.15, 0.1, 0.05]$. Results are shown in Table 2. Column $\mathbf{\Delta}_0$ reports the savings and the number of partitions of Table 1. Column $\mathbf{\Delta}_{(-0.5)}$ shows the same quantities obtained with the new value of $\mathbf{\Delta}$. We notice how, in several cases (shown in boldface), the reduced overhead cost allows to increase the number of memory blocks; in all the benchmarks, the larger slack results in improved energy reductions.

In the second experiment, we have studied the sensitivity of the partitioning algorithm to the penalty array $\mathbf{\Delta}$ on a specific benchmark (namely, `IirDemo`). Starting from the reference value of $\mathbf{\Delta}$ ($\mathbf{\Delta}_0$, represented by the middle bar in the chart of Figure 4), we have progressively increased (bars on the right) and decreased (bars on the left) the values of $\mathbf{\Delta}$. Each bar, labeled with $\mathbf{\Delta}_{\pm X}$, refers to a $\pm X\%$ increase/decrease in $\mathbf{\Delta}$ with respect to the reference vector. For example, $\mathbf{\Delta}_{+40}$ corresponds to a penalty vector of $[0.0, 0.28, 0.21, 0.14]$, where each value δ_i has been increased by 40%.
Each bar is annotated with the energy saved by the partitioned solution, and the number of partitions found, in brackets. A hard bound set to 8 has been used in the experiment. As expected, the partitioning algorithm yields solutions of decreasing quality as the penalty factor increases.

Benchmark	$\mathbf{\Delta}_0$		$\mathbf{\Delta}_{(-0.5)}$	
	Max	*Savings* [%]	*Max*	*Savings* [%]
`AdaptFilter`	2	40.4	2	45.4
`Butterfly`	2	40.8	**3**	59.4
`Chaos`	2	40.3	**3**	46.9
`Dft`	2	38.8	2	61.3
`Dhry`	2	42.7	**3**	60.1
`FilterBank`	2	42.4	2	51.1
`IirDemo`	3	44.6	3	50.8
`Integrator`	2	40.9	**3**	59.9
`Interp`	2	43.8	**3**	60.4
`Scramble`	2	42.8	2	53.3
Average		41.7		54.8

Table 2: Energy Savings with Different $\mathbf{\Delta}$'s.

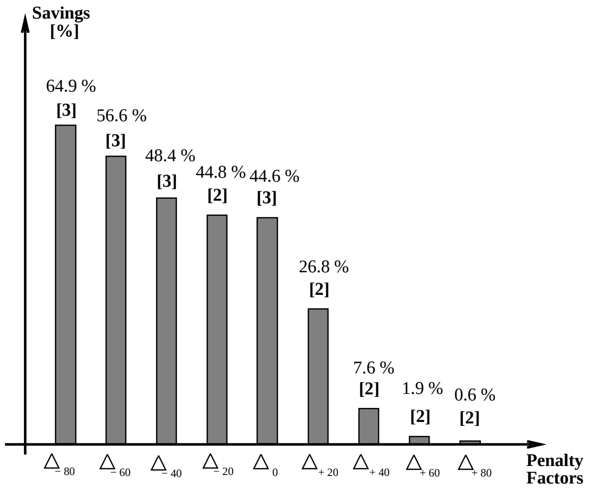

Figure 4: Impact of $\mathbf{\Delta}$ on the Achievable Savings.

6 Conclusions and Future Work

We have presented an algorithm for computing the minimum-energy partition of an on-chip memory into multiple banks that can be independently accessed. The partitioning is carried out according to the dynamic memory access profile of an embedded application. The algorithm can be constrained to the maximum number of banks, and finds the global optimum to the partitioning problem. The hardware and wiring overhead due to additional memory banks is properly taken into account as a penalty factor.
Results have been validated on a set of benchmark applications run on an ARM processor, with significant energy reduction with respect to the case of a monolithic memory.
Future work includes the addition of delay and area constraints to the algorithm, as well as a more accurate evaluation of the hardware penalty factors by analysis of actual layout data.

References

[1] J. Rabaey, M. Pedram, *Low Power Design Methodologies*, Kluwer, 1996.

[2] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, A. Vandecappelle, *Custom Memory Management Methodology Exploration for Memory Optimization for Embedded Multimedia System Design*, Kluwer, 1998.

[3] U. Ko, P. Balsara, "Energy Optimization of Multilevel Cache Architectures for RISC and CISC Processors," *IEEE Transactions on VLSI Systems*, Vol. 6, No. 2, pp. 299-308, June 1998.

[4] T. Watanabe, R. Fujita, K. Yanagisawa, "Low-Power and High-Speed Advantages of DRAM-Logic Integration for Multimedia Systems," *IEICE Transactions on Electronics*, vol. E80-C, no. 12, pp. 1523–1531, December 1997.

[5] J. Hennessy, D. Patterson, *Computer architecture, a quantitative approach*, Morgan Kaufman, 1996.

[6] UMC, *Embedded 6T Static RAM Macros Datasheet*, `http://www.umc.com`, 1999.

[7] Artisan Components, *Process-Perfect SRAM Generator Datasheet*, `http://www.artisan.com`, 1999.

[8] Virage Logic, *Custom-Touch Memory Compiler Datasheet*, `http://www.viragelogic.com`, 1999.

[9] L. Benini, A. Macii, E. Macii, M. Poncino, "Synthesis of Application-Specific Memories for Power Optimization in Embedded Systems," *DAC-37*, June 2000, To Appear.

[10] A. Farrahi, G. Tellez, M. Sarrafzadeh, "Memory Segmentation to Exploit Sleep Mode Operation," *DAC-32*, pp. 36-41, June 1995.

[11] C. Su, A. Despain, "Cache Design Tradeoffs for Power and Performance Optimization: A Case Study," *ISLPD-95*, pp. 63-68, April 1995.

[12] W. Shiue, C. Chakrabarti, "Memory Exploration for Low-Power Embedded Systems," *DAC-35*, pp. 140-145, June 1998.

[13] S. Coumeri, D. Thomas, "An environment for exploring low power memory configurations in system level design," *ICCD'99*, pp. 348-353, September 1999.

[14] S. Coumeri, D. Thomas, "Memory Modeling for System Synthesis," *ISLPED'98*, pp. 179-184, August 1998.

[15] S. Wuytack, J. Diguet, F. Catthoor, H. De Man, "Formalized Methodology for Data Reuse: Exploration for Low-Power Hierarchical Memory Mappings," *IEEE Transactions on VLSI Systems*, Vol. 6, No. 4, pp. 529-537, December 1998.

[16] P. Panda, N. Dutt, "Low-Power Memory Mapping Through Reducing Address Bus Activity," *IEEE Transactions on VLSI Systems*, Vol. 7, No. 3, pp. 309-320, September 1999.

[17] J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay and Y. Xiong, "Overview of the Ptolemy Project," ERL Technical Report UCB/ERL No. M99/37, Dept. EECS, University of California, Berkeley, July 1999.

[18] ARM Corporation, *ARM Software Development Toolkit*, Version 2.50, Reference Guide, ARM DUI 0041C, chapter 12, November 1998.