# From Architecture to Layout: Partitioned Memory Synthesis for Embedded Systems-on-Chip

L. Benini * L. Macchiarulo ‡ A. Macii ‡ E. Macii ‡ M. Poncino ‡

‡ Politecnico di Torino
DAI
Torino, ITALY 10129

* Università di Bologna
DEIS
Bologna, ITALY 40136

**Abstract**

*We propose an integrated front-end/back-end flow for the automatic generation of a multi-bank memory architecture for embedded systems. The flow is based on an algorithm for the automatic partitioning of on-chip SRAM. 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 partitioning algorithm is integrated with the physical design phase into a complete flow that allows the back-annotation of layout information to drive the partitioning process. Results, collected on a set of embedded applications for the ARM processor, have shown average energy savings around 34%.*

## 1 Introduction

A key challenge in low power design for Systems-on-Chip (SoC) containing embedded cores and memories is to reduce the power consumed in accessing memory [1, 2, 3, 4]. Data memory accesses are critical for power in data-dominated embedded applications (e.g., MPEG decoding, speech processing, etc.), and they are harder to deal with than instruction accesses, because they tend to be more irregular. Nevertheless, the critical relevance of data memory energy reduction has been stressed by many authors [2, 3], and a number of system-level data memory optimization techniques have been proposed.

One of the most effective solutions is memory partitioning. The rationale in this approach is to sub-divide a large memory into many smaller memories that can be accessed independently. Energy-per-access is reduced because on every access only a single small memory consumes power, while all the others remain quiescent.

Memory designers and computer architects have long ago acknowledged the importance of partitioning, and many circuit design techniques [5, 6, 7], as well as architectural optimizations [8, 9, 10, 11, 12] have been proposed to partition (i.e., sub-bank) memories and improve energy efficiency.

The full potential of memory partitioning can be exploited by focusing jointly on how to partition the memory space and how to distribute memory accesses in the partitioned memory architecture.

For embedded systems, it is possible to tailor a memory partition to a given target application. This objective can be achieved by analyzing data accesses with either data flow analysis [2, 3], or execution profiling [13].

A key point in application-specific memory partitioning techniques is how to take into account the overhead that comes with a partitioned memory: (i) Power and area of the control logic for memory selection and addressing; (ii) Area growth caused by the instantiation of multiple memory banks as opposed to a monolithic memory; (iii) Timing bottlenecks and additional power caused by bus multiplexing. Most of the overhead becomes apparent only at the physical design level, when a partitioned memory architecture is placed and routed. Current automatic memory partitioning techniques [2, 3, 13] do not fully address this issue.

This paper proposes an integrated optimization approach that starts from an embedded application targeted for a processor family, and outputs a complete layout, fully placed and routed, of a power-optimal partitioned memory, tailored for the chosen embedded application. The layout includes memory macros, core, addressing and memory selection logic, control signals and system buses. Our optimization algorithm takes layout-related overhead into account both during top-down optimization and bottom-up validation. The first objective is achieved thanks to a preliminary analysis, that can be run once and for all for a given technology, and that leads to a precise pre-characterization of the expected partitioning overhead to be used by the memory partitioning algorithm. Overhead estimation is exploited by the partitioning algorithm to speed-up its search for a power-optimal partition. The second objective is achieved by developing a seamless flow that integrates high-level memory partitioning, logic synthesis, placement and routing, parasitic extraction and back annotation, timing, area and power analysis.

Results show that the architecture-to-layout approach provides: (i) Dramatic power reductions, since it allows us to tighten the safety margins during optimization; (ii) Precise control of side constraints (timing, area) during power optimization; (iii) One-pass optimization, without the need of iterations in the time-consuming physical design step. The partitioned memory architectures obtained with the proposed approach result in energy savings of 34% on average (54% maximum), estimated on actual layouts.

## 2 Memory Partitioning Algorithm

In [13], an algorithm for the automatic partitioning of the memory space of an embedded application into multiple memory banks was proposed. We have enhanced the search engine developed in that work to allow a tighter integration in the archi-

tecture-to-layout flow. For a better understanding of the modifications required by the integration with the back-end, we will briefly outline its main features.

The algorithm is formulated as an optimization problem consisting of the computation of a set of memory cuts, up-to a maximum of *Max* blocks, on an abstract view (an array of $M$ locations) of the data memory to be partitioned. The optimum cut set is computed using a recursive, branch-and-bound algorithm driven by a cost function that models the overall memory energy used by the target application during its execution.

The recursive formulation allows to express the multi-way partitioning problem as a set of bi-partitioning instances. At a given recursion depth, $l$, it is assumed that the first $l-1$ cuts have been already computed; the last cut is obtained by bi-partitioning the remaining block of memory. An efficient bi-partitioning algorithm is thus central to the overall partitioning process.

Because of the size of the overall search space, effective bounds are needed to make the exploration feasible.

The algorithm uses two bounds to prune the search space. The first one is imposed by the physical interpretation of the partitioning process; that is, the feasibility of a partition is subject to the fact that the energy overhead of an extra bank is properly amortized. This fact is modeled as an array $\mathbf{\Delta} = [0, \delta_1, \ldots, \delta_{Max\text{-}1}]$; $\delta_i$ expresses the energy overhead imposed for moving from $i-1$ to $i$ memory partitions ($\delta_0$ is referred to the case of the monolithic memory). This implies that a bi-partition at a given recursion depth $l$ is rejected if this does not improve the current minimum of at least $\delta_l$.

The second bound exploits a peculiar property of the energy cost function used to drive the partitioning process. This cost function is obtained by multiplying two monotonically increasing functions of memory size: The energy for a read/write memory access and the total *cumulative* read/write access counts. Therefore, by construction, the energy cost function is also monotonically increasing with respect to memory size. This implies that, as soon as the $\Delta$ constraint is violated, say, at a given iteration $i$, further iterations on values $k > i$ can be avoided because the energy consumed in those memory blocks will be larger.

The algorithm of [13] solves the partitioning problem from an abstract point of view. However, practical application of the algorithm to a realistic memory partitioning framework must take into account several additional factors, possibly trading off optimality for flexibility. The following are important issues that were not considered in the original algorithm:

- *Accurate estimation of the energy partitioning overhead:* In the original algorithm the $\delta$s were estimated in a conservative way, under reasonable assumptions on the placement of the memory blocks. The energy partitioning overheads should be computed using information back-annotated from actual layouts; this requires the integration of the memory partitioning program in a complete design flow.

  Accurate evaluation of hardware overhead is key for an effective use of the partitioning tool, because it allows the exploration of partitioning alternatives at the application level, without the need of interacting with the back-end framework for each partitioning solution.

- *Cycle time and area constraints*: Although energy optimization is the primary target, the optimum partitioning solution should not violate possible user-specified cycle time and area constraints. From the algorithmic point of view, constraints are easily taken into account by specifying a set of relative penalties, similarly to the energy overhead $\delta$s.

  Concerning cycle time, the address decoder that activates the various memory banks increases the cycle time of the system; on the other hand, the cycle time imposed by the memories can be reduced because smaller memories are used. Concerning area, partitioning into multiple banks intuitively increases the total area. However, the actual overhead can be much smaller than expected because smaller blocks may be placed more easily.

- *Design flow integration issues*: The implementation of the algorithm within a realistic design flow requires the adoption of some approximations to trade-off execution times of the partitioning algorithm with the optimality of the produced solution. For example, although the algorithm is able to generate partitions of any size, their actual size is bound to the rules of a real-life automatic memory generator.

Addressing all the issues listed above is key for enabling a complete design path, from architecture to layout. Among the above issues, the accurate estimation of the partitioning overhead is definitely the most critical. The solution we propose is based on the idea of linking together the partitioning tool (the front-end) with physical-design tools (i.e., back-end) and thus of supplying the partitioning algorithm with realistic overhead data calculated after layout. Details on the front-end/back-end integration flow are provided in the next section.

## 3 Physical Design of Partitioned Memory

Accurately taking into account the partitioning overhead (added logic to compute the real addresses, added loads due to wiring, placement constraints) during optimization and validation is essential for calculating a realistic memory partition.

In this section, we discuss the physical design flow that we have developed. The input of the flow is the list of partitions produced by the tool described in Section 2, and its output is a legal layout of placed and routed blocks, that is, the core and memory system, including address decoder and memory selector. A back-end flow manager automatically takes care of all the phases of the physical design, all the way down to detailed routing. The following are the main steps of the flow, that are described in detail in the following subsections:

- Decoder generation;
- Memory generation;
- Block placement;
- Block routing;
- Power/ Delay/Area estimation.

The target technology is a 0.25$\mu$m process from ST Microelectronics, with six levels of metal (only the first two levels were used for signal routing between blocks). Memory blocks are synthesized using ST's embedded SRAM generators, that provide accurate timing, area and power information for the various memory cuts. The generators allow the user to specify fine details of the internal memory organization (such as buffer parametric sizing, various degrees of output multiplexing, etc.). Since we target power minimization, the internal structure of the memory banks has been specified so as to minimize energy-per-access during read and write cycles.

The processor that interfaces with the customized memory system is an ARM9 core [14]. The control and addressing logic are synthesized in standard cell style onto the 0.25$\mu$m low-power HCMOS library from ST Microelectronics.

## 3.1 Decoder Generation

The knowledge of the cut points for the addresses is used to generate a synthesizable Verilog description of a block (the decoder, hereafter) that interfaces with the CPU to translate its addresses and control signals into the multiple control and address signals needed to drive the various memory banks. The decoder takes the address lines of the core as inputs, and produces two output signals:

- *Memory select*: According to the interval of the virtual address issued by the core, it selects and activates the memory block that physically maps that address.
- *Physical address*: The virtual address has to be re-scaled to the address w.r.t. the selected memory bank. For example, if the second bank maps the virtual addresses starting from the virtual (CPU) address of *1FFF* to address *2FFF*, when the issued address is between those two limits, the physical address has to be equal to *virtual_address–1FFF*

The Verilog description is synthesized using Synopsys Design Compiler, that maps it on the standard cell library. The synthesis is timing- driven to ensure that the final implementation will not suffer from performance degradation.

The technology mapped decoder is then passed to a commercial place and route tool (Cadence Silicon Ensemble), together with the description of the standard cell library, to obtain a standard cell implementation. The result of this phase is an independent block which has to be placed and routed together with all the other blocks of the design. This "hierarchical" solution has been chosen, instead of routing the standard cell netlist together with the memory blocks, for two reasons: power distribution uses metal-2 lines inside the standard celle structure, while power distribution in the overall system uses levels 3 and 4 of metal; this would cause conflicts with standard cell power routing. This approach is consistent with common practice P&R tools that usually deal separately with block and cell routing, by cutting out an area explicitly for block design out of the standard cell area. Second, the area of the decoder with respect to the CPU core and memories is very small. Therefore, having the decoder enclosed in a small atomic block helps the task of the global placer and router, because it will tend to place the decoder close to the address pins of the core representing its primary inputs.

## 3.2 Memory Generation

The partitioning tool generates memory cuts that are consistent with the rules of the memory generator, that is, they are automatically translated into actual valid memory blocks. Clearly, a memory generator is required, in order to obtain the physical views required by the back-end flow.

The proprietary tool by ST Microelectronics used in our flow yields multiple views of a memory bank: A data-sheet description, a functional and a timing view (both in Verilog), a frame view with blockage information for floorplanning and a physical view for placement and routing.

The parameters used to generate memory cuts are the number of words, the word width, and the number of output MUXes. These three quantities are obviously not independent. In our case, the word width is fixed to 32 bits; however, for a given number of words, different memory configurations can be generated, corresponding to different memories with different features in terms of shapes of the memory (different aspect ratios), delays and power dissipation. As the main focus of this work is power reduction, we always choose the least power expensive cut, that turns out to be the memory with the least number of columns. This choice, however, could be suboptimal in terms of delay and/or area.

Two features of the generated memories are particularly attractive for the proposed flow. First, the functional signals (control, data and address buses) are to be accessed on the same side of the memory: This allows easier floorplanning (as shown in next section) and simplifies thus the physical design phase. Second, the memories can be turned off rapidly, so that it is possible to activate and deactivate a memory block at every clock cycle without impairing the performance of the whole system.

## 3.3 Block Placement

After the memory cuts are generated and the decoder synthesized, it is possible to place them on the die. We explored two different choices for the placement phase: (i) A fully automated strategy, with no insight on the functionality of the blocks, that relies on the block-placer contained in Silicon Ensemble to enforce minimal wire length; (ii) A directed floorplanning strategy, which uses the knowledge of the position of the pins and the functionality of the blocks to ease the routing phase. Out of the many choices for regular placement, we explored that of a bus-channel arrangement, in which the blocks are placed as in Figure 1.

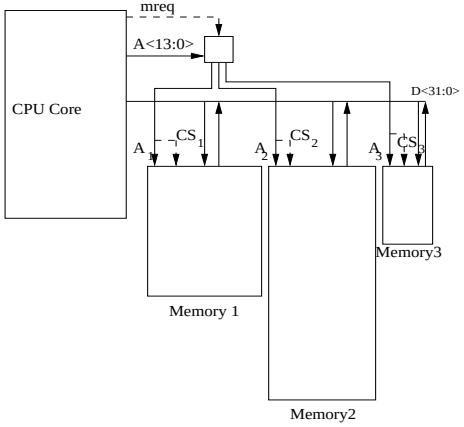


Figure 1: Floorplan for Non-Automatic Placement.

In both routing styles the system has to be described in a Verilog file, and the physical view of the blocks is given in a LEF file format. Those files are obtained from the previous two phases for the decoder and the memories, while the processor core is a fixed block of the design. After the entire design description is input, the placement operation can proceed.

In the case of automated placement, Silicon Ensemble is invoked to perform a legal placement of the blocks. In order to ease routability of the design, block halos are imposed, and no timing driven constraint added. Therefore the P&R suite tries to minimize the basic cost function of total wire length, which directly impacts power consumption; assuming equal switching activities on the buses, wiring power is basically dependent on the total length. Automatic placement typically tries to perform a strict bin-packing of the various blocks; this typically reduces the overall area, but it might complicate the routing phase.

In the case of directed placement, all memories are placed in such a way that all their pins lay on the same straight line, so that the decoder can easily feed them from above. This arrangement is made possible by the positioning of the pins of the generated memory cuts. This arrangement is typically not very efficient from the point of view of area, especially in the case

of highly asymmetrical partitions; on the other hand, ease of routing and predictability are a plus of this method.

### 3.4 Routing

The routing phase is completely automated; wire length is the cost metric used to drive routing. The model used for the power dissipated by the wires is basically proportional to their length, besides switching activity (which is not related to the physical design). Also, the power dissipated in the decoder is largely dependent on its output load, which is primarily given by memory address bus capacitance. Therefore, a standard routing tool, using total wire length as cost function, is well suited for our purposes. Only the two lowest metal layers are used to perform routing. This typically guarantees a good solution in terms of wiring delays and homogeneity of routing, at the possible expense of some minor area loss. The choice is also dictated by the decision to leave the topmost levels free for global power distribution nets.

Even if routing options for the automatic and fixed case are exactly the same, the different placement style influences this phase as well. In particular, the automatic placement can make it difficult for the tool to route specific nets, therefore increasing the probability of introducing geometric violations. Even when this is not the case, automatically placed blocks have a bigger spread in their wire lengths, that can adversely influence timing performance of the system. In both cases routing is followed by an automatic search and repair phase that solves all violation problems, leaving a legal placed and routed design whose power, area and timing performances can be accurately estimated.

### 3.5 Power Estimation

After the routing phase, we have a complete physical design of the interconnect, together with accurate information about memories, core and decoder placements; it is now possible to evaluate the power consumption of the resulting architecture. Dissipated power can be divided into three contributions:

*Memory power dissipation:* This is the most important contribution to the total power. It is obtained as the power per access (determined from the datasheet of the memory cuts), weighted by the relative activity of the accessed addresses w.r.t. the total addressed space (extracted from the memory trace):

$$\sum_{i=1}^{M}(a_i R_i + b_i W_i)$$

where the $a_i$ and $b_i$ are the number of read and write accesses on the $i$-th memory, and $R_i$ ($W_i$) denotes the energy for a read (write) operations in the $i$-th memory. We assume that there are $M$ memory blocks.

The above formula computes memory dissipation as if at least one memory is accessed per clock cycle, either with a read or a write operation. Even if this assumption is somewhat ideal, it gives correct figures of energy, when applied to the entire execution trace.

*Interconnect power dissipation:* This is the second contribution in order of importance, that is primarily responsible for the power overhead due to memory partitioning. It can be computed as:

$$\sum_{i=1}^{\#addr} SW_{addr_i} C_{addr_i} + \sum_{i=1}^{\#data} SW_{data_i} C_{data_i}$$

where $C_{addr_i}$ and $C_{data_i}$ represent the capacitances of the address and data bus lines, respectively, and the $SW_{addr_i}$ and $SW_{data_i}$ the relative switching activities.

The contribution of data lines is larger than that of address lines, because there are more data wires than address, they are longer, and they generally have higher switching activity (data tend to exhibit random behavior).

*Decoder power dissipation:* This component is the hardest to estimate before the end of the flow; fortunately, it has a marginal impact on the overall power consumption. Decoder power roughly consists of two contributions: The power dissipated inside the decoder, and the power dissipated in memory addresses and select wires. The former is a function of the complexity of the decoder logic, and is positively correlated with the number of partitions. The latter, by far the most important, is still related to the wire length.

Power estimation of the decoder has been obtained by using the traced switching activity on the CPU addresses as input switching activity, and running power simulation on the mapped netlist, loaded by the real capacitances of the wires.

### 3.6 Delay and Area Estimation

Partitioning also affects other design parameters of the system, namely, delay and area. Concerning timing, determining whether, and how much, partitioning affects timing or not depends on whether the decoder and the extra wiring are on the critical path. This, in turn, is dependent on the details of the communication protocol between the core and the memory.

In our case, where an ARM core is used, the protocol requires a delay of one clock cycle between the issuing of the address and the reading of the data-bus [14]. It is thus sufficient that the time needed for the memory to retrieve the information, plus the additional delays in the wiring and the decoder, remains smaller than the clock cycle, in the worst case, to ensure a correct behavior with no performance degradation. If this is the case, the partitioning program does not need to take into account any delay penalty, and power savings can be obtained at no performance loss.

In order to evaluate the system timing, we start from a maximum operating frequency of 150 MHz (high enough for most embedded applications, and towards the high end of the ARM low-power core performance), and assign a delay budget to the decoder plus the wiring; if the delay is smaller than this allowed budget, the system can work at its maximum cycle time. The evaluation of accurate timing figures needs an extraction of the parasitics of the interconnect. These parameters are fed back to the decoder's synthesized netlist, so that the real decoder loads are taken into account. The RC parameters allow estimation of the interconnect delay itself. The sums of these two contributions are the delays for each address and selection signals, whose maximum value is compared to the delay budget to check for proper behavior at maximum frequency.

Concerning area, our figures do not include the size of the core, that is considered as a fixed block. Area is computed as the difference between the area of the overall system and the core area. This choice is acceptable for the controlled placemente, it may be a bit inaccurate for the automatic placement, because the actual size of the core may affect the routing area.

## 4 Experimental Results

### 4.1 Partitioning Overhead Characterization

The most intuitive way of characterizing the energy overhead introduced by the partitioned memory is to take a set of sample runs of the partitioning algorithm (obtained with some initial reasonable overhead values) and to apply the flow of Section 3 to each partition. The overhead values evaluated on this sample

are back-annotated into the partitioning tool, and the process is iterated until convergence on the overhead values has been reached.
An alternative solution consists of performing statistical characterization, based on a set of synthetic partitions (i.e., artificially generated) from which the overhead values are extracted. To this purpose, we first need to determine which parameters define the dimension of the exploration space.
The values of the $\delta$s are used to model only the additional energy caused by extra wiring and the decoder, while memory energy is already taken into account in the partitioning algorithm. We might thus expect that the value of the $\delta$s only depend on the number of blocks of the partition. In general, however, also the actual sizes of the blocks of the partition may affect the $\delta$s. For instance, a quasi-balanced bi-partition (e.g., $(50\%, 50\%)$) may have a different overhead than a very unbalanced one (e.g., $(10\%, 90\%)$) because of the different loads seen at the bus outputs.
In our experiments, we have characterized the values of the $\delta$s for the cases of two, three, and four partitions. For each case, we have chosen a number of different combinations of memory block sizes, and averaged the values of $\delta$ obtained for the different configurations.

| *Partitions* | *Benchmark* | | *Synthetic* | |
|---|---|---|---|---|
| | *Manual* | *Automatic* | *Manual* | *Automatic* |
| **2** | 44.93 | 57.34 | 42.25 | 43.03 |
| **3** | 63.45 | 77.04 | 62.42 | 85.61 |
| **4** | – | – | 88.54 | 102.98 |

Table 1: Synthetic vs. Application-Dependent Overheads.

Table 1 reports the data of the comparison between the energy overhead values obtained with the two characterization strategies discussed above. Column *Benchmarks* refers to the values of $\delta$ obtained by the partitioning algorithm on a set of embedded applications; Column *Synthetic* refers to values of $\delta$ obtained from synthetic partitions. For each characterization type, two values are reported: Column *Automatic* refers to a fully automatic run of the place and route tool, whereas column *Manual* refers to a partially manual layout, where some blocks have been pre-placed. The values in the table are expressed in $\mu W/MHz$.
For the case of manual layout, the results show very good match between the overhead values obtained from both synthetic and application-dependent partitions. This fact confirms that the $\delta$s are basically insensitive to the actual sizes of the memory blocks of the partition, and mainly depend on the *number* of memory blocks. This has an important consequence, because the application-independent characterization of the $\delta$s allows a designer to use the memory partitioning algorithm as a memory optimization tool in the context of system-level design exploration. In other words, the partitioner can be used to accurately estimate the energy savings of different memory architectures without the need of a complete layout of each solution.
For the automatic case, the overhead values obtained from synthetic and application-dependent partitions are less correlated than in the case of manual layuot; this is due to the inherent approximation of the place and route tool, that is typically more effective for cell-based designs than for macro-based ones.
Notice also that none of the selected benchmarks resulted in a 4-way partition; this implies that an application-dependent characterization may sometime span only a limited region of the exploration space. In this case, it is then mandatory to resort to synthetic partitions to set up a meaningful sample set for characterization.

## 4.2 Energy Optimization

The memory partitioning tool that includes the back-end flow of Section 3 has been validated on a set of C programs that represent typical embedded applications, and that are distributed along with `Ptolemy` [15], a simulation framework for HW/SW descriptions. `ARMulator` [16], a software emulator for core processors of the ARM family, has then been used to trace data memory accesses.

| *Benchmark* | *Address Space* | *Partitions* | *Savings [%]* | |
|---|---|---|---|---|
| | | | *Automatic* | *Manual* |
| `Adaptfilter` | 3085 | [1232 1584 288] | 22.02 | 27.55 |
| `Butterfly` | 2095 | [1216 896] | 9.80 | 11.38 |
| `Chaos` | 4473 | [1584 2736 160] | 39.50 | 38.69 |
| `Dft` | 6584 | [3200 144 3264] | 48.79 | 44.55 |
| `iirDemo` | 3838 | [1376 2208 272] | 29.03 | 33.34 |
| `integrator` | 4153 | [1504 2464 192] | 37.26 | 36.41 |
| `interp` | 4120 | [1440 2464 224] | 35.92 | 35.56 |
| `loop` | 8196 | [2400 5632 192] | 53.50 | 53.88 |
| `scramble` | 1814 | [ 976 848] | 7.43 | 8.58 |
| `upsample` | 8239 | [2448 5632 176] | 53.23 | 54.00 |
| **Avg.** | | | 33.65 | 34.39 |

Table 2: Energy Savings.

The algorithm of [13], fed with the energy overheads determined with "synthetic" characterization, has been used to generate the partitions.
The execution traces have then been profiled to extract the relevant information required by the back-end flow:

- Switching activity values on the individual bits of both data and address buses.
- Switching activity at the output of the decoder function; this values also depends on the set of resulting memory cuts, which determines the toggle count of the decoder's outputs.
- Total number of read/write accesses for each block of the partition.

The partitions obtained from the partitioning tool are then used to drive the physical design flow, as described in Section 3.

| *Benchmark* | *Savings [%]* | | | |
|---|---|---|---|---|
| | *Pre-Layout* | | *Post-Layout* | |
| | *Automatic* | *Manual* | *Automatic* | *Manual* |
| `Adaptfilter` | 28.62 | 29.89 | 22.02 | 27.55 |
| `Butterfly` | 13.90 | 13.94 | 9.80 | 11.38 |
| `Chaos` | 39.26 | 40.53 | 39.50 | 38.69 |
| `Dft` | 41.87 | 48.32 | 48.79 | 44.55 |
| `iirDemo` | 34.03 | 35.30 | 29.03 | 33.34 |
| `integrator` | 36.90 | 38.17 | 37.26 | 36.41 |
| `interp` | 36.25 | 37.52 | 35.92 | 35.56 |
| `loop` | 53.03 | 54.30 | 53.50 | 53.88 |
| `scramble` | 11.50 | 11.54 | 7.43 | 8.58 |
| `upsample` | 53.09 | 54.48 | 53.23 | 54.00 |
| **Avg.** | 34.84 | 36.39 | 33.65 | 34.39 |

Table 3: Energy Savings Before and After Layout.

Table 2 shows the power results for the benchmark applications. Column *Address Space* reports the number of distinct data addresses in the trace; column *Partitions* gives the list of the memory cuts as returned by the partitioning program. Column *Savings* shows the energy savings with respect to the case of monolithic memory, as estimated on the actual layouts. The two values shown regard automatic (column *Automatic*) and manual (column *Manual*) placement strategies.
Savings range from 7% to 54%, with an average of 33.6% for the automatic placement, and 34.4% for the manual placement.
We observe that the system is assumed to be clocked at 150MHz. This cycle time ($6.66ns$) is used as a cycle time constraint during the physical design phase. From the analysis of the cycle operations of the ARM core, the memory blocks and the delays of

| *Benchmark* | *Automatic [%]* | | | *Manual [%]* | | | *Area Ratio* | |
|---|---|---|---|---|---|---|---|---|
| | *Memory* | *Wiring* | *Decoder* | *Memory* | *Wiring* | *Decoder* | *Automatic* | *Manual* |
| `Adaptfilter` | 85.0 | 13.7 | 1.3 | 91.0 | 7.5 | 1.5 | 1.75 | 1.95 |
| `Butterfly` | 93.0 | 6.4 | 0.6 | 94.0 | 5.2 | 0.8 | 1.49 | 1.57 |
| `Chaos` | 92.0 | 6.4 | 1.6 | 91.0 | 7.6 | 1.4 | 1.41 | 2.28 |
| `Dft` | 86.0 | 13.0 | 1.0 | 91.0 | 7.8 | 1.2 | 1.56 | 1.83 |
| `iirDemo` | 93.0 | 6.3 | 0.7 | 91.0 | 7.9 | 1.1 | 1.38 | 1.75 |
| `integrator` | 93.0 | 6.1 | 0.9 | 94.0 | 5.1 | 0.9 | 2.12 | 1.97 |
| `interp` | 86.0 | 12.6 | 1.4 | 91.0 | 7.8 | 1.2 | 2.15 | 1.91 |
| `loop` | 92.0 | 6.9 | 1.1 | 91.0 | 7.5 | 1.5 | 1.62 | 1.74 |
| `scramble` | 92.0 | 6.6 | 1.4 | 91.0 | 8.0 | 1.0 | 1.49 | 1.59 |
| `upsample` | 91.0 | 7.2 | 1.8 | 90.0 | 8.6 | 1.4 | 1.72 | 1.75 |
| **Avg.** | 90.3 | 8.5 | 1.2 | 91.5 | 7.3 | 1.2 | 1.67 | 1.83 |

Table 4: Energy Breakdown and Area Overhead of the Partitioned Memories.

the wires and the decoder, we have observed that all the designs end up with a cycle time shorter than the initial constraint. For this reason, delay figures are not reported in the table.

In Table 3, we compare the estimated energy savings of Table 2 (i.e., estimates after layout), to those provided by the evaluation of the cost function used by the partitioning algorithm (estimates before layout). This experiment is very important to demonstrate the effectiveness of the memory partitioning tool as a high-level exploration tool.

The difference between pre- and post-layout estimates are very small, especially for the case of the manual floorplan; this was somehow expected from the results in Table 1.

The experimental data suite is completed in Table 4 with information about the breakdown of the energy consumption of the various components of the system. For all the examples, the memory accounts for the largest fraction of the energy consumption (around 90%), while the decoder has a marginal impact on the total energy (around 1%). The energy breakdown is roughly the same for both the automatic and manual layout cases. The table also includes (columns *Area Ratio*) results on area overheads, as the ratio of the partitioned memory area over the monolithic memory area. The values are taken as the area of the bounding box of the entire layout.

## 5 Conclusions

We have proposed an integrated memory energy optimization approach that, starting from an embedded application executed on a core processor, builds a power-optimal memory partition that is directly placed and routed on a target technology, together with its addressing and memory selection logic, control signals and system buses.

The key feature of the proposed approach is the use of layout-related information during the memory partitioning phase, in the form of pre-characterized energy overheads used to speed up the the search for the optimal partition, as well as during the validation phase, thanks to a flow that links memory partitioning to physical design.

The accurate characterization of the energy overheads translates high-level optimization margins into real energy savings; therefore, the memory partitioning tool can be used as a high-level memory optimization tool for the fast exploration of partitioning alternatives.

Results show that the architecture-to-layout approach provides significant power reductions (34% on average, on the actual layout), and accurate control of side constraints (timing, area) during power optimization.

## 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] P. Panda, N. Dutt, *Memory Issues in Embedded Systems-on-Chip Optimization and Exploration*, Kluwer, 1999.

[4] S. Coumeri, *Modeling Memory Organizations for the Synthesis of Low Power System.* Doctoral Dissertation, EE and CS Dept. Carnegie Mellon University, May 1999.

[5] K. Itoh, K. Sasaki, Y. Nakagome, "Trends in Low-Power RAM Circuit Technologies," *Proceedings of the IEEE* Vol. 83, No. 4 , pp. 524-543, April 1995.

[6] B. Amrutur, M. Horowitz, "Speed and power scaling of SRAM's," *Journal of Solid-State Circuits*, Vol. 35, No. 2, pp. 175-185, February 2000.

[7] N. Kavabe, K. Usami, "Low-Power Technique for On-Chip Memory using Biased Partitioning and Access Concentration," *CICC-00*, pp. 275-278, May 2000.

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

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

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

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

[12] S. Coumeri, *Modeling Memory for System Synthesis*, *IEEE Trans. on VLSI*, Vol. 8, No. 3, pp. 327-334, June 2000.

[13] L. Benini, A. Macii, M. Poncino, "A Recursive Algorithm for Low-Power Memory Partitioning," *ISLPED-00*, pp. 78–83, July 2000.

[14] S. Segars, "The ARM9 Family - High Performance Microprocessors for Embedded Applications," *ICCD'98*, pp. 230-235, October 1998.

[15] 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.

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