Improving the Efficiency of Memory Partitioning by Address Clustering

Alberto Macii [‡] Enrico Macii [‡] Massimo Poncino [*]

[‡] Politecnico di Torino
Torino, ITALY 10129

[*] Università di Verona
Verona, ITALY 37134

Abstract

Memory partitioning is an effective approach to memory energy optimization in embedded systems. Spatial locality of the memory address profile is the key property that partitioning exploits to determine an efficient multi-bank memory architecture.
This paper presents an approach, called address clustering*, for increasing the locality of a given memory access profile, and thus improving the efficiency of partitioning.*
Results obtained on several embedded applications running on an ARM7 core show average energy reductions of 25% (maximum 57%) w.r.t. a partitioned memory architecture synthesized without resorting to address clustering.

1 Introduction

Modern SoC platforms usually contain one or more processors; because of the increasing gap between processor and memory speed, providing sufficient memory bandwidth to sustain fast program execution is thus becoming more and more challenging. As a response, SoC architectures and technologies have evolved in an effort to enable the instantiation of various types of on-chip embedded memories providing shorter latencies and wider interfaces, in order to partially fill this gap.
Ubiquity of embedded memories makes them the largest contributor to the overall energy budget of a chip. This fact motivates the recent attention of the research community to energy-efficient memory design solutions (see [1, 2, 3, 4] for a comprehensive survey of the topic).
Factors that affect memory consumption are summarized by the following model: $E_{mem} = \sum_{i=1}^{N} Cost(i)$, where N is the number of accesses during the computation, and $Cost(i)$ lumps the effective cost of an access due to the memory organization and the cost of the physical access given by the technology. The model exposes the two quantities that can be independently tackled to reduce energy consumption. We can thus classify memory energy optimization techniques into three broad classes:

- Techniques aiming at reducing *Cost(i)*: Approaches in this class build low-energy *memory architectures.*
- Techniques aiming at reducing N: Approaches in this class modify the *memory access pattern*, typically through software optimizations.
- Techniques that concurrently reduce *Cost(i)* and N.

Techniques based on concurrent optimization of memory architecture and access patterns are the most powerful, but also the most difficult to realize. This fact is witnessed by the few solutions proposed in the literature, the most popular being the DTSE exploration framework [1, 5]. One of the biggest difficulties in concurrent optimization lies in the fact that the two dimensions of the problem are regarded as orthogonal: Architectural optimizations are viewed as a *hardware* task, while the optimization of the access patterns is viewed as a *software* task.
In this work, we relax this dichotomy, and revisit the problem from an architectural perspective: The design of an application-specific memory architecture is carried out concurrently with the optimization of the access patterns by introducing some specific hardware. In practice, the access patterns are modified on the fly, without any intervention on the software application running on the processor.
The basic architectural optimization relies on the memory partitioning technique proposed in [6], where a memory block (hosting the data or the code of the application running on the system) is partitioned into multiple, non-overlapping sub-banks that can be independently accessed. Energy efficiency is achieved by exploiting the non uniformity of the memory access profile.
The contribution of this work is a technique called *address clustering*, which consists of reorganizing (through extra hardware) the address trace fed to a memory block, in such a way that the potential for the memory partitioning engine is maximized. This is equivalent to modifying the memory access profile, yet in a totally transparent fashion to the programmer.
Experimental results show that address clustering allows to reduce the energy consumption of a partitioned memory architecture by 25% on average (maximum 57%) with respect to plain partitioning, on a set of typical embedded applications running on an ARM-based system.

2 Low-Power Memory Partitioning

This section briefly describes how an on-chip memory can be partitioned into disjoint sub-banks with the objective of reducing its energy consumption [6].
The conceptual operation of a partitioning scheme is shown in Figure 1. Conventionally, the whole address space of the application is mapped to a single SRAM memory array, the smallest one in the memory library which is large enough to contain the specified range (Figure 1-(a)).

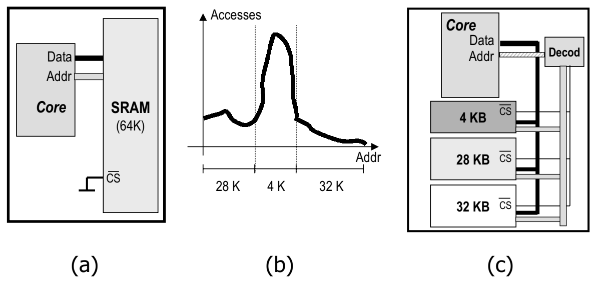

Figure 1: Memory Partitioning Example.

If we assume a dynamic access profile as in Figure 1-(b), where a small subset of the addresses in the range is very "hot" (i.e., accessed very frequently), the partitioned memory shown in Figure 1-(c) is clearly advantageous from the energy dissipation point of view. It consists of three memories and a memory selection block. Two relatively large cuts contain the top and bottom parts of the range, while the hot addresses are stored into a small memory.

Based on the fact that smaller memories have lower (area, delay, and energy) cost, the partitioned scheme allows most of the addresses to fit into a smaller block; thus, the average power in accessing the memory hierarchy is decreased, because a large fraction of accesses is concentrated to a small, power-efficient memory. Obviously, this is made possible by selectively disabling (through the chip-enable signal) the inactive memory blocks.

Notice 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 an overhead on power consumption that must be offset by the savings given by bank partitioning.

3 Address Clustering

3.1 Motivating Example

As a motivating example for the benefits of address clustering, let us consider the address profile of a MPEG decoding application, obtained for an ARM7 core. The profile refers to the instruction stream, therefore only read accesses are traced.

Figure 2 shows the occurrence frequency of each address in the trace, which consists of 31233 addresses (from 0 to 124892) fitting into a memory cut having 1952 rows × 512 columns. The plot highlights the well-known irregularity of the profile, with very few "hot" addresses. This memory, when exercised with this trace (44.4 million total read accesses), consumes 170 mJ.

The application of the memory partitioning algorithm described in [6] to this trace yields a multi-bank memory configuration consisting of three memory blocks of sizes 736 × 256, 696 × 512, and 892 × 512, consuming a total of 96 mJ (inclusive of the overhead), a 43.5% energy reduction w.r.t. the monolithic memory architecture. In particular, the middle memory cut (696 rows × 512 columns), which is also the smallest one, keeps the majority (82%) of the memory accesses (36 million out of 44.4).

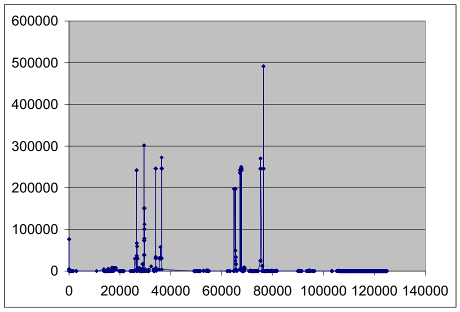

Figure 2: Address Profile of a MPEG Decoder.

Consider now the situation depicted in Figure 3, where the addresses of the MPEG trace are now *clustered* towards the low-end of the profile. Intuitively, this improves the spatial locality of the profile by keeping the most visited addresses as close as possible.

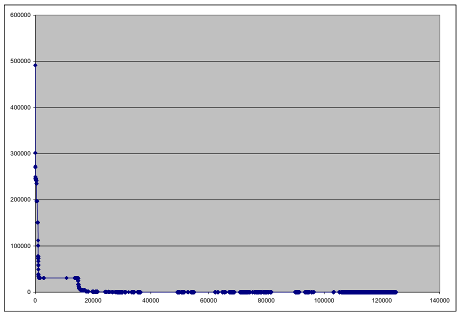

Figure 3: Clustered Address Profile of a MPEG Decoder.

The application of the memory partitioning algorithm to this trace yields now a partitioned memory consisting of two blocks of sizes 212 × 128 and 1900 × 512, consuming a total of 42 mJ, with an additional 56% of energy saved (75% over the non-partitioned case). The smallest memory block contains now 99% of the accesses (43.99 million out of 44.4 million), whereas the second, largest block is active only for a very small fraction (1%) of the time.

Clearly, dealing with the second trace is preferable. However, unless there exists a compiler that is able to explicitly enforce locality, obtaining the trace of Figure 3 implies *relocating* all the addresses of the trace. This has clearly a cost: It requires an address decoding function with unacceptable complexity, that would offset the energy savings achieved by increasing the locality of the profile. As a matter of fact, such encoder would be a 32-input, fully-specified function that builds the correspondence between original and clustered address.

On the other hand, intermediate solutions are possible, where only a (small) subset of the addresses is clustered, yet maximizing the increase in locality. In other terms, we face the usual trade-off between the achievable energy savings and the complexity of the encoder. The next two sections describe the formulation and a solution of the address clustering problem, respectively.

3.2 Problem Formulation

The address clustering problem consists of finding a relocation of a proper subset of the address space that maximizes the locality of the dynamic trace, and with the ultimate objective of minimizing the energy consumption of the memory architecture for the given trace, possibly under area and cycle time constraints.

The actual energy consumption of a partitioned memory architecture is determined by the outcome of the memory partitioning algorithm of [6]. However, running the memory partitioning engine for each candidate clustering solution may become quite computationally expensive. We thus need to devise a high-level cost function that can be used into an exploration framework to evaluate the suitability of a clustering solution.

Following the observation that the potential of the memory partitioning engine is related to the locality of the trace, the cost metric to be chosen should be related to the locality of the address profile.

3.3 Cost Metrics

The dynamic access profile **C** for the target application is given as an array $\mathbf{C} = [c_0, c_1, \ldots, c_{N-1}]$, where c_i is the total number of accesses to address i, and N is the total number of addresses. Although read and write accesses have different energy costs, at this level of abstraction we do not distinguish between them, and just consider the total number of accesses.

We want to infer, from a given trace, a single-value quantity that expresses its degree of (spatial) locality. Although a trace sorted in non-decreasing or non-increasing order has clearly maximum locality, it only represents a special case of a maximum-locality trace. Locality is affected by the fact that highly visited addresses are close in space, rather than the fact of being *sorted.*

In other words, any metric related to the "distance" of a profile from a perfectly sorted one would not be very precise, because we need a quantity that also measures the magnitude of the out-of-sequence elements. Furthermore, evaluating the shape of a profile against a sorted one would imply forcing the profile to exhibit a locality that is skewed towards the upper or lower end of the profile, which is clearly a limitation.

A second requirement is related to the basic operation realized by address clustering, that is, moving only a relatively small subset of the addresses. Therefore, the metric should also provide an indication of how many addresses are covered by a given subset of addresses W. In other terms, it should be parameterized with respect to a value that represents a set of addresses.

We define the metric that incorporates both requirements as the *density* of a profile. Given a profile **C**, its density is *the maximum value of the cumulative number of accesses for a sliding window of size W over the trace.* In formula:

$$D(\mathbf{C}, W) = \max_i(S_i) \quad i = 0, \ldots, N - W \tag{1}$$

where S_i is defined as $S_i = \sum_{j=0}^{W-1} c_{i+j}$.

Density, as defined in Equation 1, is an absolute quantity. For a more effective use of density as a cost function across different traces, we normalize the value of $D(\mathbf{C}, W)$ to the total number of memory accesses in the trace $Tot = \sum_{i=0}^{N-1} c_i$, to make it a number between 0 and 1. We denote the normalized density with d:

$$d(\mathbf{C}, W) = \frac{D(\mathbf{C}, W)}{Tot} \tag{2}$$

Density measures the degree of *spatial locality* of a trace, in terms of how highly visited addresses are kept close. In practice, $D(\mathbf{C}, W)$ measures the maximum number of total accesses of **C** covered by a window of size W.

Density is a suitable measure to compare two different address traces, for the same value of W, as shown in the following example.

Example 1

Consider the profile of Figure 2. Figure 4 shows the values of $d(\mathbf{C}, W)$ for $W = 32, 64, 128, 256$ and 512. The solid line refers to the original profile, whereas the dashed one refers to the profile obtained by relocating the $M = 256$ most visited addresses.

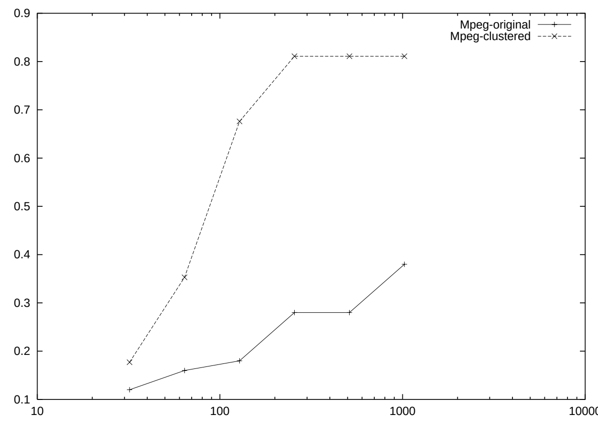

Figure 4: Density of the Original and of a Clustered Trace.

The superiority in terms of locality of the clustered trace is evident from the plot: A window of $W = 256$ addresses covers more than 80% of the total accesses, whereas it roughly covers a 20% in the original case.

The choice of the number M of addresses to be clustered is in strict relation with the value W used for computing the density of trace **C**. In other words, for a given value of W, the density of trace **C** is maximal if $M \geq W$. Also, the larger the value of W, the larger the density of the trace. The problem to be solved is then that of finding a good value of W (and thus of $M = W$) such that the density of the clustered trace **C'** (i.e., $d(\mathbf{C}', W)$) is large and W is small. The latter objective corresponds to minimizing the number of addresses that need to be moved, and thus helps in keeping under control the HW encoder that actuates address clustering.

4 Clustering Algorithm

The optimization procedure consists of a main exploration loop, that is used to find a good value for W (and thus of M). This value is then fed to the core optimization routine that generates the encoder and the clustered trace.

Figure 5 shows the high-level pseudo-code of the exploration loop. Procedure `Explore` receives, as input, an address profile $\mathbf{C}$, and a threshold T used by the algorithm to check the convergence and thus control termination of the execution.

```
1 Explore ($\mathbf{C}$,$T$) {
2   $\mathbf{C}_{sort}$ = Sort($\mathbf{C}$);
3   for ($W = 1$ to $N$) {
4     $Dens = d(\mathbf{C}_{sort}, W)$;
5     if (EvalSlope ($\mathbf{C}_{sort}$,$T$,$Dens$)) {
6       return $W$;
      }
7   $W += step$;
  }
```

Figure 5: Exploration Algorithm.

The exploration is based on a simple convergence test on the values of locality. In particular, W is chosen such that the marginal increase in locality (with respect to the previous iteration) falls below T.

The initial trace $\mathbf{C}$ is first sorted in decreasing order of accesses (Line 2). Then, a loop iterates on the size of the sliding window W (Line 3). For each W, locality of $d(\mathbf{C}_{sort}, W)$ is computed (Line 4). Procedure `EvalSlope`, based on the analysis of a history of locality values and the current one, decides whether convergence is reached (Line 5). If this is the case, it returns the current value of W (Line 6), that will be used by the clustering procedure; otherwise, the analysis is repeated with a new value of W (Line 7).

The high-level pseudo-code of the address clustering algorithm is shown in Figure 6. Procedure `Cluster` receives, as input, the original memory address trace, the number of addresses M to be clustered and the "golden" trace *GoldenTrc* (i.e., the trace in which all the addresses are sorted in non-increasing order of frequency count).

The procedure is invoked with $OrigTrc = \mathbf{C}$, $M = W$ and $GoldenTrc = \mathbf{C}_{sort}$.

The algorithm works in two phases: In the first one (Lines 2–8), each of the first M addresses of the original trace is checked to see if it is contained in the first M of the golden trace (i.e., the most visited ones) – Line 4. If so, it is left untouched and its position, both in the original and the golden traces, is marked as "used" (Line 6 and Line 8).

In the second phase (Lines 9–18), the first M addresses in the golden trace that are left unused (Line 10) replace the "unused" addresses of the original trace (Line 14). This replacement implies the update of the original trace in such a way that the original behavior is preserved (Lines 15–18). Procedure `Cluster` returns a modified trace whose first M locations contain the M most visited addresses.

```
1  Cluster ($OrigTrc$,$M$,$GoldenTrc$) {
2    for ($i = 0$ to $M$) {
3      for ($j = 0$ to $M$) {
4        if ($OrigTrc[j] == GoldenTrc[i]$) {
5          Leave the address in the original place;
6          Mark $OrigTrc[j]$ as used;
7          Mark $GoldenTrc[i]$ as used;
8          break;
         }
       }
     }
9    for ($i = 0$ to $M$) {
10     if ($GoldenTrc[i]$ is not used) {
11       for ($j = 0$ to $M$) {
12         if ($OrigTrc[j]$ is not used) {
13           $temp = OrigTrc[j]$;
14           $OrigTrc[j] = GoldenTrc[i]$;
15           for ($k = 0$ to end of $OrigTrc$) {
16             if (($OrigTrc[k] == temp$) &&
                     $OrigTrc[k]$ is not used) {
17               $OrigTrc[k]$ = temp;
18               break;
               }
             }
           }
         }
       }
     }
   }
```

Figure 6: Clustering Algorithm.

5 Experimental Results

5.1 Energy Optimization

The address clustering technique has been implemented around the memory partitioning tool of [6], and validated on a set of C programs that represent typical embedded applications. Some of the benchmarks are taken from the `Ptolemy` distribution [7], others come from the *MediaBench* suite [8].

We have used the ARM software development kit as target platform for the execution of the benchmarks. In particular, the memory access traces have been obtained through the profiling features provided by `ARMulator` [9], the instruction set simulator of the ARM tool-chain.

Table 1 shows the address clustering results. It is split into two parts: The top half concerns *data* accesses, whereas the bottom half refers to *instruction* accesses.

Column *#Addr* gives the total number of distinct addresses in the trace. Column E_{mono} gives the energy of the monolithic memory that contains all the data/instructions, while columns $E_{partitioned}$ show the total memory energy of a partitioned memory architecture. In particular, column *Original* refers to the application of the plain memory partitioning algorithm of [6], while columns $M = 256$, $M = 512$, and $M = 1024$ give results for memory partitioning combined with address clustering.

Benchmark	*#Addr*	E_{mono}	$E_{partitioned}$							
		[nJ]	*Original*		$M = 256$		$M = 512$		$M = 1024$	
			E [nJ]	*Δ [%]*	*E [nJ]*	*Δ [%]*	*E [nJ]*	*Δ [%]*	*E [nJ]*	*Δ [%]*
Chaos	9108	1.56e5	8.11e4	48.0	7.45e4	52.2	7.45e4	52.2	7.45e4	52.2
FilterBank	16659	1.04e5	5.08e4	51.1	3.94e4	62.1	3.91e4	62.4	3.91e4	62.4
iirDemo	9417	4.06e5	1.72e5	52.2	1.72e5	57.6	1.72e5	57.6	1.72e5	57.6
integrator	5253	4.48e4	2.66e4	40.6	2.53e4	43.5	2.53e4	43.5	2.53e4	43.5
scramble	2537	5.19e5	4.05e5	22.0	3.73e5	28.1	3.73e5	28.1	3.73e5	28.1
DTMFCodec	3520	2.62e7	1.77e7	32.4	1.67e7	36.3	1.67e7	36.3	1.67e7	36.3
MpegDec	184320	1.51e9	4.60e8	69.5	3.54e8	76.5	3.34e8	77.8	2.34e8	84.5
EPIC	216064	1.04e9	1.19e8	88.6	1.15e8	89.0	1.15e8	89.0	1.15e8	89.0
unEPIC	216064	6.28e7	1.74e7	72.3	1.67e7	73.4	1.67e7	73.4	1.67e7	73.4
MpegEnc	31224	3.52e10	1.65e10	53.1	1.82e10	48.3	1.24e10	64.8	1.06e10	69.9
MpegDec	31224	2.05e9	1.17e9	42.9	7.47e8	63.6	5.21e8	74.6	5.01e8	75.6
DTMFCodec	19982	3.23e8	1.32e8	59.1	1.48e8	54.2	1.16e8	64.1	1.06e8	67.2
EPIC	56460	2.73e9	9.23e8	66.1	6.89e8	74.7	5.12e8	78.2	5.13e8	81.2
unEPIC	52388	1.99e8	6.41e7	67.8	6.89e7	65.4	4.77e7	76.0	3.87e7	80.6
RawEnc	10610	2.22e7	1.21e7	45.6	9.73e6	56.2	9.84e6	55.8	9.84e6	55.8
RawDec	10610	3.18e7	1.85e7	41.9	1.38e7	56.6	1.45e7	54.5	1.45e7	54.5
Average				**53.3**		**58.6**		**61.9**		**63.4**

Table 1: Comparison of Energy Savings: Before and After Clustering.

All the memory data are provided by the memory partitioning tool, which uses memory models derived from a commercial memory generator by STMicroelectronics. In particular, the data refer to a library of static SRAM memory cuts.

We first observe that, in order to give the reader the feeling about how clustering impacts energy savings, in the table we have reported data for different values of M, namely $M = 256$, $M = 512$ and $M = 1024$, instead of simply providing the results obtained for the best value of M, as determined by the exploration procedure.

The interesting information we can extract from the data is that, for several examples, energy savings (which are recalculated in Table 2 by taking plain memory as the term of comparison for the sake of readability) are not monotonically increasing with the value of M. In these cases, a small value of M is sufficient, as further increasing it does not provide further advantages while complicating unnecessarily the implementation of the decoder. This is due to the fact that memory traces have very irregular locality and this clearly justifies the need of using a threshold-controlled exploration loop.

We notice also that all the energy figures are inclusive of the hardware overhead caused by partitioning, as discussed in Section 2.

Benchmark	*Saving [%]*		
	$M = 256$	$M = 512$	$M = 1024$
Chaos	8.1	8.1	8.1
FilterBank	22.4	23.0	23.0
iirDemo	11.3	11.3	11.3
integrator	4.9	4.9	4.9
scramble	7.9	7.9	6.2
DTMFCodec	5.6	5.6	5.6
MpegDec	23.1	27.3	49.1
EPIC	3.5	3.5	3.5
unEPIC	4.0	4.0	4.0
MpegEnc	-10.3	24.8	35.8
MpegDec	36.2	55.5	57.2
DTMFCodec	-12.1	12.1	19.7
EPIC	25.4	35.6	44.4
unEPIC	-7.5	25.6	39.6
RawEnc	19.6	19.6	19.6
RawDec	25.3	25.3	25.3
Average	**10.5**	**18.4**	**22.5**

Table 2: Energy Savings w.r.t Non Clustered Traces.

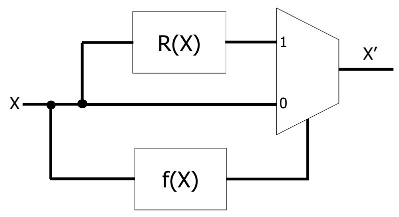

Figure 7: Conceptual Architecture of the Address Encoder.

5.2 Encoder Overhead Analysis

The clustering of the addresses is implemented through a hardware encoder, which translates the original addresses into their modified values. Since the clustering is based on the swap of address pairs, there are actually $2M$ clustered addresses.

Figure 7 shows a conceptual block diagram of the encoder. X denotes the original address as issued by the core, and X' the clustered address. The encoder has the typical structure of a conditional encoder. $f(X)$ represents a function that evaluates to 1 if the address X belongs to the set of the $2M$ clustered addresses; in that case, the output consists of the clustered address $X' = R(X)$, where R is the function that maps the clustered addresses to actual ones. When $f(X) = 0$, $X' \equiv X$.

The encoder is implemented as a fully combinational network that performs the mapping from X to X'. Given the relatively small values of M (compared to the total number of addresses), the complexity of such mapping network is acceptable. In fact, it corresponds to a Boolean function consisting of $2M$ 32-input minterms. Since the whole input space of 2^{32} minterms largely exceeds the on-set of this function, the huge don't care set results in quite small and energy-efficient encoders.

Figure 8 shows the energy consumption of the encoder versus some typical values of M. The values represent the average energy over all the benchmarks of Table 1.

The various encoders have been synthesized with `Synopsys DesignCompiler` on a $0.25\mu m$ technology by STMicroelectronics, under delay constraints. Performance is in fact more critical than power, as discussed later. Power figures are obtained with `Synopsys PowerCompiler`, under application of the corresponding address traces at a frequency of 150MHz.

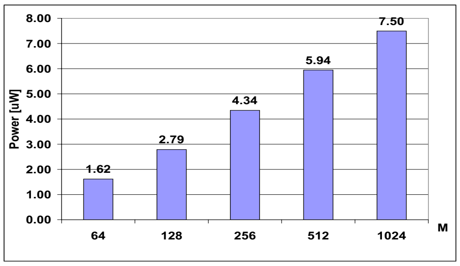

Figure 8: Encoder Energy Overhead vs. M.

The plot shows that energy increases sub-linearly with respect to M; furthermore, power figures are negligible with respect to the contribution of the memory block. For instance, the encoder consumes approximately $7mW$ for $M = 1024$, a marginal (about 2%) overhead compared to a 16K memory, which dissipates about $375mW$.
The variance of the energy figures over the various applications is relatively small, because (i) the complexity of the decoder is basically independent of the set of addresses that are clustered, and (ii) the switching activity of the address lines is very similar for all benchmarks (because of the well-known spatial locality of address traces).
Delay overhead is also important, as address translation affects the timing of a memory access. However, as discussed in [6], in the ARM-based architectural model assumed in this paper, the protocol requires a delay of one clock cycle between the issuing of the address and the reading of the data-bus [10]. It is thus sufficient that the time needed for the memory to retrieve the information, plus the additional delays in the wiring and the encoder, remains smaller than the clock cycle, in the worst case, to ensure a correct behavior with no performance degradation. We have assumed an operating frequency of 150 MHz (towards the high end of the ARM low-power core performance), corresponding to a 6.66ns cycle time. To allow a reasonable safety margin, we have imposed a $5ns$ delay constraint in the synthesis of the decoder.
The energy, delay, and area overheads caused by the presence of this encoder are automatically taken into account by the memory partitioning engine, in the form of pre-characterized penalty factors used to prevent arbitrary fine partitioning. Such overheads are pre-characterized using layout-related technological information captured with the flow described in [6]. With respect to the conventional partitioning, these overheads have been properly modified according to the analysis carried out in this section.

6 Conclusions

The energy reduction achievable by memory partitioning techniques can be improved sensibly by increasing the locality of the trace. Rather than realizing it on the application at the software level, we proposed an architectural solution, called *address clustering*, which consists of the relocation of a small subset of the address space that maximizes the potential energy savings, by keeping the hardware overhead at a minimum.
Experimental results on a set of typical embedded applications running on an ARM-based system show that address clustering is able to reduce the energy consumption of a partitioned memory architecture by 25% on average (maximum 57%) with respect to the partitioning driven by the original trace.

References

[1] 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.

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

[3] L. Benini, G. De Micheli, "System-Level Power Optimization: Techniques and Tools," *ACM Transactions on Design Automation of Electronic Systems*, Vol. 5, No. 2, pp. 115-192, April 2000.

[4] A. Macii, L. Benini, M. Poncino, *Memory Design Techniques for Low-Energy Embedded Systems*, Kluwer, 2002.

[5] 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.

[6] L. Macchiarulo, A. Macii, L. Benini, M. Poncino, "Layout-Driven Memory Synthesis for Embedded Systems-on-Chip," *IEEE Transactions on Very Large Scale Integration (VLSI)*, Vol. 10, No. 2, pp. 96-105, April 2002.

[7] J. Davis II et al., "Overview of the Ptolemy Project," ERL Technical Report UCB/ERL No. M99/37, Dept. EECS, University of California, Berkeley, July 1999.

[8] C. Lee, M. Potkonjak, W. H. Mangione-Smith, "MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *30th IEEE/ACM International Symposium on Microarchitecture,* pp. 330-335, Research Triangle Park, NC, December, 1997.

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

[10] S. Segars, "The ARM9 Family - High Performance Microprocessors for Embedded Applications," *ICCD-98: IEEE International Conference on Computer Design*, pp. 230-235, Austin, TX, October 1998.