

Synthesis of Application-Specific Memories for Power Optimization in Embedded Systems

Luca Benini * Alberto Macii † Enrico Macii † Massimo Poncino †

* Università di Bologna
Bologna, ITALY 40136

† Politecnico di Torino
Torino, ITALY 10129

Abstract

This paper presents a novel approach to memory power optimization for embedded systems based on the exploitation of data locality. Locations with highest access frequency are mapped onto a small, low-power application-specific memory which is placed close to the processor. Although, in principle, a cache may be used to implement such a memory, more efficient solutions may be adopted. We propose an architecture that outperforms (power-wise) different types of cache memories at no penalty in performance. Power savings (averaged over a number of embedded applications running on ARM processors) range from 12% to 68%.

1 Introduction

The focus of this paper is on memory synthesis for low-power embedded systems. We assume the availability of a processor core and of the embedded application that it must execute, and we generate a memory architecture that is *power-efficient* and *application-specific*. More in detail, we create a hierarchical memory architecture that exploits a known principle of locality [1] which states that accesses to (instruction and data) memory are highly non-uniform over the complete address space.

The rationale in our approach is to map all most frequently accessed locations onto a small memory which can be placed very close to the processor. The power cost in accessing this memory is much smaller than that required for fetching and storing information into the main memory. Clearly, a cache is the most obvious implementation of this concept. Our work moves from the observation that cache memories are *not* the most power-efficient architecture for a “local” memory. Information storage/retrieval into/from a cache is much more power-consuming than accessing a memory containing the same amount of data. Caches have been developed to provide access locality in general-purpose systems, and they are very flexible. Such flexibility has a major power cost. When accessing a cache line we need to access and compare its tag to the incoming address. These operations consume non-negligible power.

Cache tags can be seen as a general-purpose adaptive decoding logic that maps non-consecutive memory addresses onto a small local memory. Our approach eliminates cache tag overhead by creating *application-specific decoding logic*, which is automatically synthesized at design time based on profiling information obtained by running the embedded application on the core processor. The purpose of such logic is to map the small set of most frequently accessed addresses into a set of consecutive addresses of a small local memory. This memory is similar to a scratch-pad buffer with the major difference that it can be distributed in many non-contiguous locations of the address space.

The proposed solution shares some of the advantages of cache memories (i.e., the capability of “localizing” non-contiguous locations in memory), with the advantages of memory buffers (i.e., low power consumption per access). We can expect our approach to provide some performance improvements as well, because accesses to a simple memory are faster than accesses to a cache of comparable size. However, the latter may not be as large, since tag lookups in cache memories are performed in parallel with data accesses.

It is important to notice that our architecture trades off flexibility for decreased power consumption per memory access. Differently from tags in cache memories, our decoding logic does not dynamically adapt to non-stationary program working sets. In other words, caches can provide good locality of reference even for working sets that change over time, because they are dynamically loaded with new addresses in response to misses. Even though this characteristics can be very helpful, it can also be harmful. *Cache pollution* is a well-known mis-behavior of caches, where rarely-accessed lines evict highly-accessed lines just to be replaced again after a short time. Furthermore, cache updates on misses consume additional power. In general, cache memories are the only viable choice for general-purpose systems, where working sets cannot be pre-characterized in advance. However, the application-specific nature of our technique makes it a viable choice for embedded systems, as demonstrated by our experimental results. Power savings (averaged over a number of embedded applications running on ARM core processors) are, in fact, around 68%, when the comparison is done against caches with minimum line size, degree of associativity and write-through replacement. Smaller, yet noticeable reductions (around 12%) occur with respect to direct-mapped caches with the largest allowable line size and write-back policy.

2 Related Work

Several approaches to memory hierarchy optimization for low power do exist in the literature [2, 3, 4, 5, 6]. As in our case, they specify a core processor and an application (or an application mix), and they explore the memory hierarchy design space to find the organization that best matches processor and application. Usually, the design space is parameterized and discretized, to allow exhaustive or near-exhaustive search. Most research efforts in this area postulate a memory hierarchy with one or more levels of caching. A finite number of cache sizes and cache organization options are considered (e.g., degree of associativity, cache replacement policy, cache sub-banking). The best memory organization is obtained by simulating the workload for all possible alternative architectures.

Several authors have adopted the explorative approach to analyze power-efficient cache structures and management policies, but they did not depart from the assumption that local low-power storage is organized as a cache memory. Our approach gives up the flexibility offered by caches, in exchange for a more power-efficient memory structure.

3 Application-Specific Memory Architecture

In this section, we propose a memory architecture that can be used instead of a conventional cache. We provide details on the actual structure of the application-specific memory (ASM henceforth), as well as on the energy model we have adopted to compare ASM consumption to that of various cache options. Then, we focus on architectural issues such as the interface between ASM and both the core processor and the main memory.

3.1 Memory Array

The ASM is implemented as a simple memory array. This choice minimizes the cost of memory accesses. For comparable sizes, cache accesses are intrinsically more energy-demanding, due to the extra overhead required by the tag array and tag comparison logic. Tags increase the number of bit-lines in the array, and this effect is more evident for smaller line sizes, because the cost of the tag array is not amortized over longer data array lines.

Besides tag circuitry, caches also have additional overhead which is not present in a conventional memory: *i*) Cache associativity implies using several parallel arrays (i.e., *ways*), each with a tag section. Associativity, besides increasing the total length of a bit-line, also requires multiple comparators for parallel tag comparison. *ii*) Updating cache contents to increase locality requires the implementation of a proper write policy in case of a write hit and a proper replacement policy in case of a miss. Obviously, this further complicates the cache control logic.

Analytical evaluation of these effects requires a model that accurately accounts for all sources of cache energy dissipation. Furthermore, to allow a consistent comparison between ASMs and caches, the model should be easily customizable to represent the energy consumption of a conventional memory.

Several cache models have appeared in the literature, although most of them are intended for performance, rather than energy estimation. Among the existing energy models, we have adopted the one introduced in [3], that is derived from the performance model of [7]. In this model, cache energy consumption is broken down in four main components: Bit-lines dissipation, word-lines dissipation, input and output lines dissipation. In formula:

$$E_{cache} = E_{bit} + E_{word} + E_{input} + E_{output} \quad (1)$$

All energy contributions are parameterized in such a way that the same model can be used for both the ASM and the cache. It is important to emphasize that the above model, as explicitly mentioned in [3], provides a conservative estimate of the actual cache energy. In fact, minor energy contributions such as the energy due to tag comparators, control and replacement logic, and address comparators in the write-back queue are neglected. Although these components are of a smaller order of magnitude than those of Equation 1, they are peculiar of caches, while they must not (and will not) be part of the energy model for the ASM.

When the model of Equation 1 is used to estimate the dissipation of the ASM, the various contributions to total energy are weighted differently. Each term in the equation is a product $E = V^2 C N$, where N is the number of times the corresponding memory component is used, C is an estimate of the switched capacitance, V is the power supply voltage. Because of space limitation, we do not describe the formulae for the computation of C and N (the interested reader may refer to [3] for details). The main differences between ASM and cache memory are in E_{bit} and E_{word} . Both these contributions have much smaller C in ASMs, because of the absence of tags. E_{output} can change as well. In this case, C_{out} is the same, but N_{out} changes substantially, depending on cache organization and replacement policy.

3.2 Decoding Logic and Hit Function

Figure 1 shows a conceptual scheme of the proposed architecture, where the ASM interacts with the core in the same way a cache does. We assume that the M most frequently accessed data are put in the ASM. The 32-bit address is fed to the ASM decoder (block *Dec*), which implements the mapping between 32-bit processor addresses and $\log_2 M$ -bit ASM addresses. This translation is carried out for every memory data access. The dotted box denotes the chip boundary, meaning that ASM and related decoding logic is intended for on-chip implementation.

If the currently issued data address corresponds to one of the M data stored in the ASM, we have a hit; otherwise, there is a miss. The decoder, besides implementing address translation, has an additional output that signals to the processor that the current address cannot be found in the ASM but it has to be fetched from the main memory. As for cache memories, this handshake implies a latency penalty, because the processor has to wait during the access in the main memory.

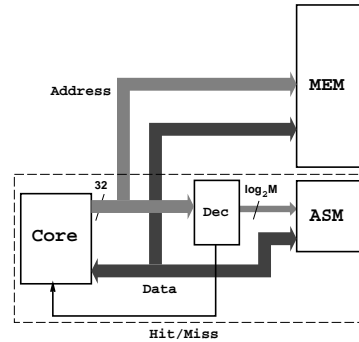


Figure 1: ASM-Based Architecture.

In our experiments we have adopted the architecture of Figure 1, where the ASM is used as a cache memory with low cost per access. However, ASMs can be effective even in cacheless architectures, which are still quite common in embedded systems (where execution time predictability is a primary concern). In this case, there is no need to explicitly signal an ASM miss to the processor. In fact, a hit/miss signal is required in cache-based architectures because the processor needs to be warned (and stalled accordingly): *i*) On a miss, that a given block has to be replaced; *ii*) On a write, that a block has to be copied into the main memory (on every write with a write-through policy, or when a block is replaced with a write-back policy).

In a cacheless system, the management of ASM misses is made transparent to the processor. The hit/miss signal can be used directly to control the tri-state bus drivers that access the main memory. More precisely, one may think of properly decoding addresses (in the processor address decoder) in such a way that the main memory and the ASM belong to disjoint address spaces. Main memory addresses are normally disabled (i.e., drivers are in high-impedance state), and they are activated only when an ASM miss occurs. Notice that in this case, memory access time is constant and equal to the (small) access time to the ASM plus the (large) access time to external memory.

If there is freedom in the design of the ASM, the hardware overhead due to the decoder may be reduced by merging block *Dec* in Figure 2-(a) with the internal decoder of the memory array (block *IDec* in Figure 2-(a)). The unified decoder (block *UDec* in Figure 2-(b)) accepts 32-bit addresses from the processor and directly translates them into 1-out-of- M codes, suitable to select the rows of the memory array.

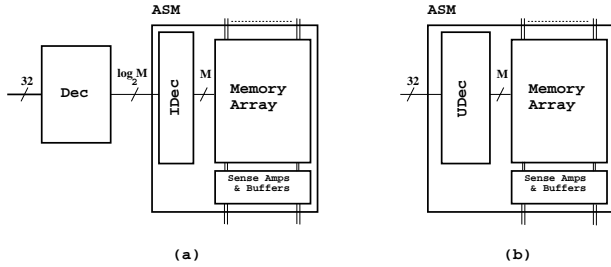


Figure 2: Alternative Decoder Structures.

4 Experimental Results

The fundamental assumption upon which our optimization technique is based is that, normally, memory locations accessed by an embedded program are very localized. Such assumption is confirmed by the results of extensive profiling we have done on a number of programs that are typically executed on embedded processors. More specifically, we have considered a set of C benchmarks distributed together with the Ptolemy system [8]. ARMulator [9], a software emulator for core processors of the ARM family, has then been used to trace memory accesses. The data of the profiling are collected in Table 1. For each benchmark, column *Tot. Acc.* gives the total number of memory accesses, column *256 Acc.* reports the number of accesses to the 256 most frequent locations and column *256 %* the corresponding percentage over all accesses. Similar data are shown for the 512 and 1024 most frequently accessed locations. On average, for more than 95% of the time, programs access the 256 most frequent locations. Such average goes up to almost 99% if 1024 locations are considered. In view of these results, we can state that the replacement of conventional caches with ASMs is advantageous; in the remainder of this section, we experimentally demonstrate the foundation of our claim. The experimental flow we have adopted to compare energy consumption of ASMs against that of traditional caches is depicted in Figure 3.

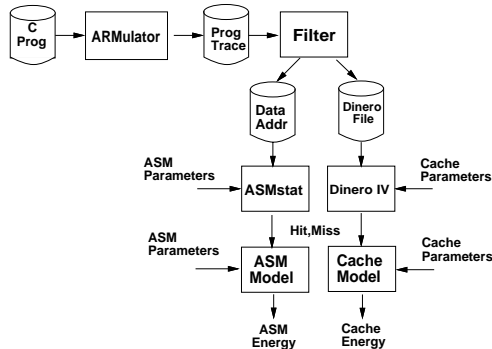


Figure 3: ASM/Cache Energy Estimation Flow.

The entry point is a C program, which is fed to ARMulator. The memory access trace is then filtered to extract data address information. For the ASM, profiling information is static, that is, the hit-ratio coincides with the fraction of total addresses contained in the ASM. Conversely, evaluation of cache statistics requires a cache simulator, in order to account for the dynamic effects caused by the cache replacement mechanism. For this reason, the filter generates two data address files, in different formats. The first one is used by a in-house profiling

program (ASMstat) that, based on the ASM parameters (number of words), yields the number of “hits” that have occurred in the data address trace. The second file contains data addresses in one of the formats supported by the Dinero cache simulator [10], that we have used to evaluate cache statistics. Dinero uses cache parameters such as total size, line size, associativity, write and replacement policy to compute miss and hit rates, for both read and write accesses.

The resulting data are plugged into the ASM and cache models of Section 3 to get the corresponding energy consumption values for the given memory access trace.

We have compared the use of ASMs to a cache with *write-through* policy and to a cache with *write-back* policy. Data of the experiments are pictorially summarized in Figure 4.

For each benchmark, three different ASM and cache sizes have been considered, namely, 1KB, 2KB, and 4KB. In the case of the ASM, where the word-length is fixed (4 bytes), only one configuration per chosen size is allowed. On the contrary, the cache may have different “shapes” for a given size. In the chart, L denotes the line size, and a the degree of associativity. Under the constraint that the minimum size of a cache line is $L = 4$ bytes, the 1KB cache can have only one configuration ($L = 4, a = 1$), while there are three alternatives for a 2KB cache, and six for a 4KB cache.

The results prove that replacing the cache with an ASM always produces sizable energy savings. The advantage is remarkably higher with respect to write-through caches, since the main, off-chip memory is accessed every time a datum is updated in the cache. Smaller, yet substantial savings occur with respect to caches implementing a write-back policy. In this case, energy reduction is mainly due to the intrinsically more efficient architecture of an ASM w.r.t. that of a traditional cache.

The highest energy savings (averaged over all benchmarks) is 68% and it occurs with respect to a cache with minimum line size and degree of associativity (i.e., $L = 4, a = 1$) adopting a write-through policy. On the other hand, in the worst-case the energy reduction is 12%, corresponding to the case of a direct-mapped cache with the largest allowable line size ($L = 16, a = 1$) adopting a write-back policy.

We observe that, normally, the main target of cache-based architectures is speed optimization. Therefore, to enforce performance constraints, designers may select caches that are heavily sub-optimal from the energy stand-point. Using ASMs, minimum energy always corresponds to maximum performance.

Concerning the synthesis of the ASM decoding logic, the starting point is a 32-input (i.e., the ARM word-width), $(\log_2 M) + 1$ -output two-level logic function specifying address re-mapping and the hit/miss signal (M indicates the number of ASM locations). This function is synthesized as a multi-level network using Synopsys DesignCompiler and mapped onto a technology library from ST Microelectronics containing approximately 400 primitives.

Energy consumption estimates are obtained through Synopsys DesignPower assuming a clock frequency of 50 MHz and refer to the execution of the complete benchmark. Therefore, they can be properly compared to the energy values referred to ASM and cache consumption.

Table 2 reports, for each benchmark, area (A , in μm^2), delay (D , in $nsec$), and energy consumption (E , in nJ) of the decoding logic for ASMs with three different sizes (1KB, 2KB, 4KB).

Comparison of the energy values to those of both ASMs and caches has shown a difference of 1 to 3 orders of magnitude. Consequently, we can safely state that the impact of ASM decoders to the total energy budget is negligible.

Benchmark	Tot. Acc.	256		512		1024	
		Acc.	%	Acc.	%	Acc.	%
AdaptFilter	104479	102376	97.987%	103679	99.234%	104321	99.848%
Butterfly	123250	122048	99.024%	122728	99.576%	123250	100.000%
Chaos	103999	102869	98.913%	103352	99.377%	103864	99.870%
Dft	127660	108786	85.215%	112882	88.423%	120111	97.610%
Dhry	98936	94100	95.112%	94760	95.779%	95525	96.552%
FilterBank	391356	345875	88.378%	355159	90.750%	364013	93.013%
IirDemo	240597	235972	98.077%	239333	99.474%	240036	99.766%
Integrator	355334	353529	99.492%	354061	99.641%	354818	99.854%
Interp	540819	536785	99.254%	538213	99.518%	540200	99.885%
Scramble	558603	549534	98.376%	556089	99.549%	558036	99.898%
Average			95.983%		97.132%		98.630%

Table 1: Profiling Results.

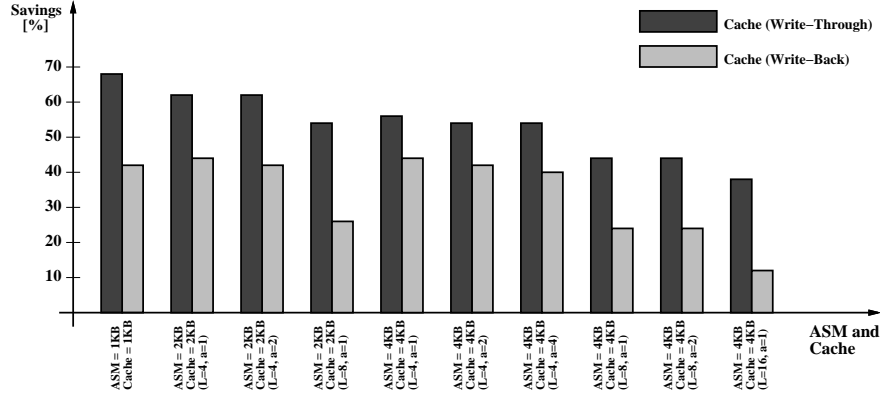


Figure 4: Energy Savings (Average Over All Benchmarks).

Benchmark	1 KB			2 KB			4 KB		
	A	D	E	A	D	E	A	D	E
AdaptFilter	18405	5.97	4.99e3	35604	7.65	8.59e3	49428	13.94	1.44e4
Butterfly	20610	9.31	5.52e3	34011	9.96	9.72e3	49536	10.65	1.51e4
Chaos	24597	9.46	6.17e3	32814	8.39	8.86e3	39303	14.64	1.08e4
Dft	9684	4.23	2.96e3	14427	4.75	5.13e3	32841	7.06	1.14e4
Dhry	19494	5.31	4.03e3	35568	11.54	6.78e3	66132	15.79	1.47e4
FilterBank	32328	11.25	2.68e4	59571	14.88	5.17e4	61135	18.37	7.83e4
IirDemo	20655	9.23	1.23e4	40554	12.27	2.25e4	57879	16.51	3.35e4
Integrator	24750	7.06	1.96e4	29277	8.48	2.52e4	38700	7.00	3.44e4
Interp	21870	8.06	2.62e4	28440	9.81	4.19e4	41742	12.37	5.24e4
Scramble	19134	5.87	2.63e4	33381	8.29	4.46e4	38745	9.53	6.65e4

Table 2: Area, Delay and Energy Consumption of ASM Decoders.

5 Conclusions

We have presented a novel solution for the design of the memory hierarchy of low-power embedded systems. The method is based on the idea of mapping the most frequently accessed data onto a small memory, called *application-specific memory* (ASM), which can be placed very close to the processor. Using such a memory instead of a cache is advantageous because it makes memory accesses less energy consuming, while still keeping the advantage of localizing non-contiguous memory locations.

Experimental results on a set of typical embedded programs have shown very promising energy savings (from 12% up to 68%, on average) with respect to equivalent caches having different sizes, organizations and configurations.

References

- [1] J. Hennessy, D. Patterson, Computer Architecture, A Quantitative Approach, Morgan Kaufman, 1996.
- [2] C. Su, A. Despain, "Cache Design Tradeoffs for Power and Performance Optimization: A Case Study," ISLPD-95, pp. 63-68, Apr. 1995.
- [3] M. Kamble, K. Ghose, "Analytical Energy Dissipation Models for Low Power Caches," ISLPED-97, pp. 143-148, Aug. 1997.
- [4] 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, Jun. 1998.
- [5] R. Bahar, G. Albera, S. Manne, "Power and Performance Tradeoffs using Various Caching Strategies," ISLPED-98, pp. 70-75, Aug. 1998.
- [6] W. Shiue, C. Chakrabarti, "Memory Exploration for Low Power, Embedded Systems," DAC-35, pp. 140-145, Jun. 1998.
- [7] S. J. E. Wilton, N. P. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," IEEE Journal of Solid-State Circuits, Vol. 31, No. 5, pp. 677-687, May 1996.
- [8] J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neundorffer, J. Reekie, N. Smyth, J. Tsay and Y. Xiong, Overview of the Ptolemy Project, ERL Technical Report UCB/ERL No. M99/37, UC Berkeley, Jul. 1999.
- [9] ARM Corporation, ARM Software Development Toolkit, Version 2.50, Reference Guide, ARM DUI 0041 C, Chapter 12, Nov. 1998.
- [10] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, A. J. Smith, "Cache Performance of the SPEC Benchmark Suite," IEEE Micro, Vol. 3, No. 2, pp. 17-27, Aug. 1993.