A Survey of Scratch-Pad Memory Management Techniques for low-power and -energy

Maha Idrissi Aouad, Olivier Zendra

A Survey of Scratch-Pad Memory Management Techniques for low-power and -energy

Maha Idrissi Aouad and Olivier Zendra

INRIA-Lorraine / LORIA, Building C,
615 Rue du Jardin Botanique, BP 101,
54602 Villers-Lès-Nancy Cedex,
FRANCE
Maha.IdrissiAouad@loria.fr, Olivier.Zendra@inria.fr

Abstract. Scratch-Pad Memories (SPMs) are considered to be effective in helping reduce memory energy consumption. However, the variety of SPM management techniques complicates the choice of the right one to implement. In this paper, we first give a synthesis on existing SPM management techniques for low-power and -energy outlining their comparative advantages, drawbacks and trade-offs. Then, we propose a new general classification which encompasses most existing research works. This classification has the advantage of clearly exhibiting lesser explored techniques, hence providing hints for future research.

1 Introduction

Reducing energy consumption of embedded systems is a topical and very crucial subject. Many systems are energy-constrained and, despite batteries progress, these systems still have a limited autonomy. This mainly concerns numerous daily life objects such as cell phones, laptops, PDAs, MP3 players, etc.

Different options to save energy, hence increase autonomy, exist but we can't detail them here due to the lack of space. The interested reader can refer to [Graybill and Melhem, 2002; Zendra, 2006] for a more comprehensive view. These various approaches can be classified in two main categories: hardware optimizations and software optimizations. Hardware techniques fall beyond the scope of this paper, but a large amount of literature about them is available (see first parts of [Graybill and Melhem, 2002]). Some works interestingly couple hardware techniques with software ones, such as [Poletti *et al.*, 2004] (that relies on Direct Memory Access, or DMA, to reduce the copy cost between SPM and DRAM), or [Benini *et al.*, 2000] (using Application-Specific Memory, ASM).

In this paper, however, we will focus on software, compiler-assisted techniques. Cache memories, although they help a lot with program speed, do not always fit in embedded systems: they increase the system size and its energy cost (cache area plus managing logic). In contrast, SPMs have interesting features. Like caches, they consist of small, fast SRAM, but the main difference is that SPMs are directly and explicitly managed at the software level, either by the developer or by the compiler, whereas caches require extra dedicated

circuits. Compared to cache, SPM thus has several advantages [Zendra, 2006]. SPM requires up to 40% less energy and 34% less area than cache [Banakar *et al.*, 2002]. Additionally, SPM cost is lower and its software management makes it more predictable, which is an important feature for real-time systems.
According to [Adiletta *et al.*, 2002; Brash, 2002], a large variety of chips with SPM is available today in the market. Moreover, trends [LCTES, 2003] indicate that the dominance of SPMs is likely to continue in the future. Consequently, many authors have tried to profit from the advantages of SPMs and various related research directions have been investigated. These techniques and algorithms, synthesized in [Benini and Micheli, 1999], try to optimally allocate application code and/or data to SPM in order to reduce the energy consumption of embedded systems. The interested reader can look at [Benini and Micheli, 2000] for a comprehensive list of references. Although some of the research works we present in this paper have not been targeted specifically to Object-Oriented Languages (OOL), we think their underlying principles still apply to OOL.

The rest of the paper is organized as follows. Section 2 describes some software optimization techniques for SPM. Section 3 presents a discussion with a new classification. Finally, section 4 concludes.

2 Software Optimization Techniques for SPM

Numerous research works focus on SPM optimized management techniques. In this section, we present a survey and a classification of these techniques. In order to manage the SPM space, some approaches try to answer the question of which data to allocate to which memory type. Others are based on optimizing data locality. All these methods rely on profiling information to place the most frequently used or most cache conflicting data in fast memory, and other data in slower memory. The main trade-offs of these approaches revolve around the objects considered (arrays, loops, global, heap or stack variables...).

2.1 Techniques Focusing on Data Placement in Memory According to Memory Type

The first category of SPM management techniques comprises those that can be characterized as focusing on data placement in memory according to memory type. These approaches try to answer the question of which program variables should be allocated to which memory or memory bank. In these techniques, because of the reduced size of SRAM, the lesser-used variables are first allocated to slower memory banks (DRAM), while the most frequently used variables are kept in fast memory (SRAM) as much as possible. These methods use profile data to gather access frequency information in order to place frequently used data in fast memory, and other data in slower memory. To do so, most authors model the problem as a 0/1 integer linear programming (ILP) problem and then use an available IP solver to solve it.

[Avissar *et al.*, 2002] considers global and stack variables and chooses between SPM and cache, while [Steinke *et al.*, 2002b; Wehmeyer *et al.*, 2004] consider global variables, functions and basic blocks and choose between SPM banks. Instead of using one single large SPM, the simulated results obtained by [Wehmeyer *et al.*, 2004] have shown that by using a partitioned SPM improvements of up to 22% in the energy consumption of the memory subsystem can be obtained.

These techniques are all based on the frequency of data accesses. In contrast, [Panda *et al.*, 1997] considers arrays and scalar variables and focuses on data that is the most cache-conflict prone. The authors rely on profile data to place the most conflicting data in SRAM. All of these approaches require knowledge of the SPM size at compile time but [Nguyen *et al.*, 2005] presents a compiler method whose resulting executable is portable across SPMs of any size. It consists on discovering SPM size first, either by making an OS or low-level system call if available, or by probing addresses in memory using a binary search pattern and observing the latency to find the range of addresses belonging to SPM. Then, the memory allocation algorithm is the same as in [Avissar *et al.*, 2002].

2.2 Techniques Focusing on the Locality of Memory Access

The techniques presented in this section can be considered as a refinement of those of section 2.1. Indeed, in addition to finding the best data placement with respect to memory types, it is interesting to investigate the locality of memory accesses in order to further optimize energy usage.

Spatial Locality Some methods are based on optimizing spatial locality, that is on ensuring that successive SPM accesses use the same SPM bank as much as possible. Indeed, increasing the spatial locality for a set of SPM banks clearly increases the duration of idleness for other SPM banks, which in turn helps to amortize the cost of placing a bank into low-power mode and then later transitioning it back to normal operation mode.

[Athavale *et al.*, 2001] explores the energy consumption of array allocation mechanisms in Java. Using a set of array-dominated benchmarks and a partitioned memory architecture with multiple low-power operating modes, the authors study two data optimization techniques: memory layout modification and array interleaving. This memory layout modification consists in changing the storage order of data inside an array in order to improve its spatial locality. In addition, array interleaving groups together in the same memory module elements that belong to different multi-dimensional arrays, thus increasing the inter-access interval (time between two references to the same module) for the unused modules. This provides an opportunity to operate the unused memory modules in a lower power mode for a longer time. Their experimental results show that using layout transformation and array interleaving optimization provides an average of 9.68% and 14.96% energy savings, respectively.

The compiler-based strategy proposed in [Kandemir *et al.*, 2005] is also effective in reducing leakage energy of on-chip SPMs and has the advantage of

dealing with arrays and loops in general without a restriction to a specific language. In addition to having some similarities with [Athavale *et al.*, 2001], the technique presented in [Kandemir *et al.*, 2005] also seems general enough to be applicable to any object-oriented language. The idea in [Kandemir *et al.*, 2005] is to divide SPM into banks and use compiler-guided data layout optimization and data migration to maximize SPM bank idleness, thereby increasing the chances of placing banks into low-power state. This work focuses on reducing leakage consumption of on-chip SPMs without hurting performance.

Temporal Locality Other methods are based on temporal locality, that is the fact that recently accessed SPM banks are likely to be accessed again in a near future. [Verma *et al.*, 2004] presents a profile based approach which, on the basis of live ranges of both variables and code segments, replenishes the content of the SPM. These elements are optimally chosen in order to minimize the energy overheads due to spilling memory object to and from the SPM. This technique also computes addresses within the SPM address range where variables and code segments have to be copied. These addresses are computed such that a large number of variables and code segments fit in the same SPM space.

2.3 Comprehensive Techniques Dealing with all Memory Objects

The techniques we mentioned in the previous sections have the drawback of not taking into account all kinds of objects. In the current section, we will focus on Udayakumaran and Barua's works, which conversely deal with *all* memory objects: arrays, loops, global, heap and stack variables. Udayakumaran and Barua's approach is based on works by Kandemir *et al.* and tries to improve them.

Both methods move data back and forth between DRAM and SPM under compiler control, but two improvements are brought by [Udayakumaran and Barua, 2003] over [Kandemir *et al.*, 2001].

First of all, [Kandemir *et al.*, 2001] considers global and stack array variables only and has the three additional following restrictions. One, the programs should primarily access arrays of the innermost loops. Two, the loops must be well-structured and must not have any other control flow such as *if-else*, *break* and *continue* statements. Three, the codes containing these constructs must be well written, that is to say without any of the hand-made optimizations often found in many such codes, because these optimizations consider not only the loop nest in question, but also a much larger context. Combining these three restrictions, Kandemir *et al.*'s method applies to well-structured scientific and multimedia codes. However, as underlined by Udayakumaran and Barua, most programs in embedded systems do not fit within these strict restrictions. [Udayakumaran and Barua, 2003] has improved the generality of the method and applies it to all global and stack variables, and all access patterns to those variables. The method thus becomes more general and is able to exploit locality for all codes, including those with irregular accesses patterns, variables other than arrays, code with pointers and irregular control flow.

The second improvement brought by [Udayakumaran and Barua, 2003] is that [Kandemir *et al.*, 2001] considers each loop nest independently, whereas Udayakumaran and Barua's method is a whole-program analysis across all control structures. This has several consequences. One is that the method presented by Kandemir *et al.* is locally optimized for each loop, while the method of Udayakumaran and Barua is globally optimized for the entire program. Another consequence is that with the method of Kandemir *et al.* the entire SPM is available for each loop nest. In contrast, the approach of Udayakumaran and Barua might choose to do this, but is not constrained to do so. It may choose to use part of the SPM for data that is shared between successive control constructs thus saving on transfer time and energy to DRAM.

Note however that for arrays, it is possible to bring in parts of an array instead of considering the whole array with [Kandemir *et al.*, 2001], whereas this is impossible with [Udayakumaran and Barua, 2003].

Udayakumaran and Barua have extended their work in other papers. For example, the approach in [Udayakumaran *et al.*, 2006] also handles code objects and provides some measurements for energy consumption. Their results from simulation show that their scheme reduces runtime by up to 39.8% and energy by up to 31.3% on average for their benchmarks, depending on the SRAM size used, when compared to [Avissar *et al.*, 2002].

The much cited [Dominguez *et al.*, 2005] is also a very interesting piece of work because it is, to our best knowledge, the only work that considers heap data and has an SPM management policy at runtime that allows fixed moves (as shown in Table 1). Their simulation results show that this method reduces the average runtime by 34.6% and the average power consumption by 39.9% for the same size of SPM fixed at 5% of total data size, when compared to placing all heap variables in DRAM and only global and stack data in SPM.

Finally, [Udayakumaran and Barua, 2006] extends the work done by investigating SPM allocation for arrays. This last paper modifies the algorithm already proposed by adding a code to identify partial variables such as a row, a column or even a collection of elements belonging to an array variable that is accessed by a loop nest, using an affine analysis pass. The aim of this pass is to enable allocation of parts of an array, for instance when the whole array does not fit into the SPM. However, this paper presents results according to runtime only, energy consumption is not considered. We think this should be addressed.

3 Discussion

In this paragraph, we try to bring a fresh look at the SPM management techniques. Thus, we have done a general study that allows us to propose a new classification presented in Table 1 which considers two criteria.

The first criterion deals with the way information is collected and the second criterion refers to the SPM management policy at runtime. For the *Information Collection* criterion, *Compilation* means that the code is analyzed at compile

Table 1. A Classification of SPM Management Phases

References	**Information Collection**	**SPM Management Policy at Runtime**
[Kandemir *et al.*, 2001]	Compilation + Profiles	Static
[Udayakumaran and Barua, 2003]	Compilation	Free Moves
[Udayakumaran *et al.*, 2006]	Compilation	Free Moves
[Udayakumaran and Barua, 2006]	Compilation	Free Moves
[Nguyen *et al.*, 2005]	Compilation + Profiles	Static
[Egger *et al.*, 2006]	Compilation + Profiles	Static
[Absar and Catthoor, 2005]	Compilation	Static
[Poletti *et al.*, 2004]	Compilation	Static
[Steinke *et al.*, 2002a]	Compilation	Static
[Verma *et al.*, 2003]	Compilation	Static
[Verma *et al.*, 2004]	Compilation	Free Moves
[Dominguez *et al.*, 2005]	Compilation	Fixed Moves
[Avissar *et al.*, 2002]	Compilation + Profiles	Static
[Steinke *et al.*, 2002b]	Compilation	Static
[Wehmeyer *et al.*, 2004]	Compilation + Profiles	Static
[Panda *et al.*, 1997]	Compilation + Profiles	Static
[Athavale *et al.*, 2001]	Compilation	Static
[Kandemir *et al.*, 2005]	Compilation + Profiles	Static
[Hiser and Davidson, 2004]	Compilation + Profiles	Static

time, whereas *Profiles* indicates the use of runtime profiling. The *SPM Management Policy at Runtime* can be *Static* which means that data could be overwritten but not moved to another place. With *Moves* some existing variables in SPM could be evicted to make space for incoming ones. In this way, data is never lost. On the one hand, *Fixed Moves* always place data at the same offset in the SPM or DRAM. On the other hand, with *Free Moves* the SPM allocation can be dynamically adapted at runtime by placing most frequently used data at any free location in the SPM or DRAM.

Several methods are *Static* and are based on *Compilation* information only [Absar and Catthoor, 2005; Steinke *et al.*, 2002b; Athavale *et al.*, 2001]. However, with execution *Profiles* an accurate view of the data access patterns can be obtained, since the profiles contain information about which variables are accessed during which program phase. In this context, [Egger *et al.*, 2006; Avissar *et al.*, 2002; Kandemir *et al.*, 2001] provided experimental results which generally improve the ones they had obtained with the *Static* approach. Furthermore, SPM management policies based on *Moves* [Dominguez *et al.*, 2005; Verma *et al.*, 2004; Udayakumaran and Barua, 2003] are more effective than *Static* ones, because they optimize the use of the SPM space and allow to change the content of the SPM at runtime. In other words, just like in a cache, data is moved back and forth between DRAM and SPM, but under compiler control. The energy overhead of performing a move is compensated by a better placement.

As we can see from Table 1, there are different combinations of *Information Collection* and *SPM Management Policy at Runtime*. However, our synthesis shows that there is no method merging compilation and profiles information with an SPM management policy based on moves. This thus seems a potentially interesting area for new research. Furthermore, most of the presented works are not targeted to OOL but apply to them nonetheless. We consider it would be interesting to also study SPM management techniques by taking into account some more specific features of OOL such as object memory layout.

4 Conclusion and perspectives

In this paper, we have given in section 2 a global structure of the use of optimized SPM management techniques. The classification we have proposed enables to make a synthesis of most SPM management techniques, which makes it possible to have a more global and precise view of these techniques aiming at reducing energy and/or power consumption in embedded systems. Indeed, our classification in section 3 exhibits very clearly the fact that combining *Compilation*, *Profiles* information and an SPM management policy based on *Moves* at runtime has not been explored yet. In our point of view, this could be more effective in reducing energy consumption as the results obtained separately are interesting. We plan to explore this in our future works, as well as to study SPM with respect to some specific features of OOL.

References

[Absar and Catthoor, 2005] M. J. Absar and F. Catthoor. Compiler-based approach for exploiting scratch-pad in presence of irregular array access. In *DATE*, 2005.

[Adiletta *et al.*, 2002] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, 6(3), Aug. 2002.

[Athavale *et al.*, 2001] R. Athavale, N. Vijaykrishnan, M. T. Kandemir, and M. J. Irwin. Influence of array allocation mechanisms on memory system energy. In *IPDPS*, page 3, 2001.

[Avissar *et al.*, 2002] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Transaction. on Embedded Computing Systems.*, 1(1):6–26, 2002.

[Banakar *et al.*, 2002] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES*, pages 73–78, New York, NY, USA, 2002. ACM Press.

[Benini and Micheli, 1999] L. Benini and G. De Micheli. System-level power optimization: Techniques and tools. In *ISLPED-99:ACM/IEEE*, pages 288–293, 1999.

[Benini and Micheli, 2000] L. Benini and G. De Micheli. System-level power optimization: techniques and tools. *IEEE Design and Test*, 17(2):74–85, 2000.

[Benini *et al.*, 2000] L. Benini, A. Macii, E. Macii, and M. Poncino. Increasing Energy Efficiency of Embedded Systems by Application Specific Memory Hierarchy Generation. *IEEE Design and Test*, 17(2):74–85, 2000.

[Brash, 2002] D. Brash. The ARM architecture Version 6 (ARMv6). In *ARM Ltd.*, January 2002. White Paper.

[Dominguez *et al.*, 2005] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):115–192, 2005.

[Egger *et al.*, 2006] B. Egger, J. Lee, and H. Shin. Scratchpad Memory Management for Portable Systems with a Memory Management Unit. In *EMSOFT*, 2006.

[Graybill and Melhem, 2002] Robert Graybill and Rami Melhem. *Power aware computing*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[Hiser and Davidson, 2004] J. D. Hiser and J. W. Davidson. EMBARC: An Efficient Memory Bank Assignment Algorithm for Retargetable Compilers. In *ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 182–191. ACM Press, 2004.

[Kandemir *et al.*, 2001] M. T. Kandemir, I. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic Management of Scratch-pad Memory Space. In *Pmc. DAC*, 2001.

[Kandemir *et al.*, 2005] M. T. Kandemir, M. J. Irwin, G. Chen, and I. Kolcu. Compiler-guided leakage optimization for banked scratch-pad memories. *IEEE Trans. VLSI Syst.*, 13(10):1136–1146, 2005.

[LCTES, 2003] LCTES. Compilation Challenges for Network Processors. In *Compilers and Tools for Embedded Systems*. Industrial Panel, ACM Conference on Languages, June 2003.

[Nguyen *et al.*, 2005] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *CASES*, 2005.

[Panda *et al.*, 1997] P. R. Panda, N. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *DATE*, 1997.

[Poletti *et al.*, 2004] F. Poletti, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *DAC*, pages 238–243, 2004.

[Steinke *et al.*, 2002a] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. In *ISSS*, 2002.

[Steinke *et al.*, 2002b] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE*, page 409. IEEE Computer Society, 2002.

[Udayakumaran and Barua, 2003] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *CASES*, pages 276–286. ACM Press, 2003.

[Udayakumaran and Barua, 2006] S. Udayakumaran and R. Barua. An integrated scratch-pad allocator for affine and non-affine code. In *DATE*, pages 925–930, 2006.

[Udayakumaran *et al.*, 2006] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *Embedded Comput. Syst*, 5(2):472–511, 2006.

[Verma *et al.*, 2003] M. Verma, S. Steinke, and P. Marwedel. Data Partitioning for Maximal Scratchpad Usage. In *ASPDAC*, 2003.

[Verma *et al.*, 2004] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic Overlay of Scratchpad Memory for Energy Minimization. In *CODES+ISSS*, 2004.

[Wehmeyer *et al.*, 2004] L. Wehmeyer, U. Helmig, and P. Marwedel. Compiler-optimized usage of partitioned memories. In *WMPI*, 2004.

[Zendra, 2006] O. Zendra. Memory and compiler optimizations for low-power and -energy. In *ICOOOLPS*, 2006.