LEAKAGE-AWARE SCRATCH-PAD MEMORY BANKING FOR EMBEDDED MULTIDIMENSIONAL SIGNAL PROCESSING

Florin Balasa[*] *Noha Abuaesh*[*] *Cristian V. Gingu*[†] *Doru V. Nasui*[‡]

[*]American University in Cairo, Egypt
[†]Fermilab, Batavia, IL, USA
[‡]American International Radio, Inc., Rolling Meadows, IL, USA

ABSTRACT

Partitioning a memory into multiple banks that can be independently accessed is an approach mainly used for the reduction of the dynamic energy consumption. When leakage energy comes into play as well, the idle memory banks must be put in a low-leakage 'dormant' state to save static energy when not accessed. The energy savings must be large enough to compensate the energy overhead spent by changing the bank status from active to dormant, then back to active again. This paper addresses the problem of energy-aware on-chip memory banking, taking into account – during the exploration of the search space – the idleness time intervals of the data mapped into the memory banks. As on-chip storage, we target scratch-pad memories (SPMs) since they are commonly used in embedded systems as an alternative to caches. The proposed approach proved to be computationally fast and very efficient when tested for several data-intensive applications, whose behavioral specifications contain multidimensional arrays as main data structures.

Index terms– memory management, memory banking, multidimensional signal processing, scratch-pad memory.

1. INTRODUCTION

As on-chip storage, the scratch-pad memories (SPMs) – compiler-controlled SRAMs, are widely employed in embedded systems as an alternative to caches because they achieve comparable performance with higher energy efficiency [1]. They are, in fact, quite similar to caches in terms of size and speed (typically, single-cycle access), but they do not need any mapping logic: different from caches in which the main memory data is *dynamically* assigned during program execution by means of the line replacement mechanism, the SPMs need only a *static* assignment[1] – at design time, so the caches' tag area and hit/miss acknowledging circuitry is no longer needed. In embedded systems, low-energy consumption is more desirable than flexibility in the adaptation of workload. Consequently, SPMs are often employed instead of caches because they are more power-efficient and save some die area, at the cost of a possible loss of performance – which is usually not very significant.

In spite of their superior power efficiency, SPMs are often the largest contributor to the on-chip energy budget: this is caused by the tendency of the designer to store on-chip as much data as possible in order to improve performance. Moreover, since many signal processing systems, particularly in the multimedia and telecommunication domains, are synthesized to execute data-intensive applications, the memory subsystem is, typically, a major contributor to the overall energy budget of the entire system [3] (and often a bottleneck for performance – as the coined term '*memory wall*' suggests [4]). Hierarchical memory organizations reduce energy consumption by exploiting the non-uniformities of memory accesses: the reduction of power consumption can be achieved by assigning the frequently-accessed data to low hierarchy levels [5], [6].

Moreover, within a given memory hierarchy level, power can be reduced by memory partitioning – whose principle is to divide the address space into several smaller blocks, and to map these blocks to physical memory banks [2], [3], [7], [8]. This issue has been widely acknowledged and several teams of researchers came up with a wide spectrum of solutions for partitioning caches [9], [10] and SPMs, foremost targeting the reduction of dynamic energy consumption which expands due to memory accesses. Kandemir *et al.* proposed a compiler-controlled dynamic SPM management using loop and data transformations [2] like, for instance, *array interleaving* [11]. They also explored nonuniform partitioning solutions [12]. Benini *et al.* proposed a recursive partitioning of the SPM address space, that achieved a complete exploration of the banking solutions [7]. A further computational model was described in [8]: the cost function in the optimization was shown to exhibit properties that allowed to apply a dynamic programming paradigm.

Technology scaling past 45 nm offered advantages to embedded systems in terms of energy consumption [13], but the static energy due to leakage currents became more critical for memories: their high density of integration translates into a higher power density that increases temperature, which in turn increases leakage currents significantly. Kandemir *et al.* exploited SPM bank locality for maximizing the idleness, thus ensuring maximal amortization of the energy spent

[1]Dynamic re-fills of SPMs are possible though, and may be beneficial [2].

978-1-4799-2893-4/14/$31.00 ©2014 IEEE

```
C[0] = 0 ;                                  // int  A[64][64][64]: input;
for (int i=16; i<48; i++ )
  for (int j=16; j<48; j++ )
    for (int k=16; k<48; k++ ) {
      B[i][j][k][0] = 0 ;
      for (int p=i-16; p<=i+16; p++ )
        for (int q=j-16; q<=j+64; q++ )
          for (int r=k-16; r<=k+16; r++ )
            B[i][j][k][1089*p+33*q+r-1089*i-33*j-k+17969]
              =  A[i][j][k] - A[p][q][r]
              +  B[i][j][k][1089*p+33*q+r-1089*i-33*j-k+17968] ;
        C[1024*i+32*j+k-16911]
              = B[i][j][k][35937] + C[1024*i+32*j+k-16912] ;
    }
out = C[32768];
```

Fig. 1. Code example of behavioral specification.

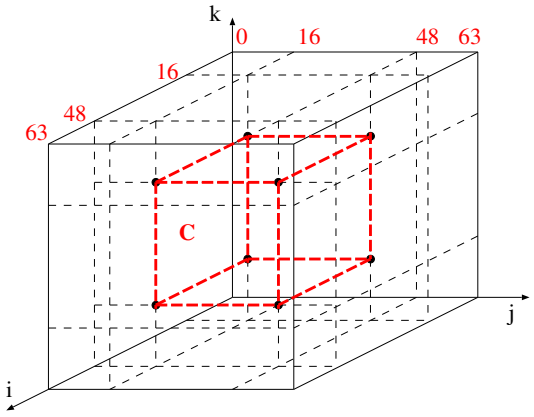

Fig. 2. Partitioning signal A's array space (from the illustrative example in Fig. 1) into linearly bounded lattices (LBLs).

on memory re-activation [14]. Hardware schemes putting a memory block into a dormant (*sleep*) state with negligible energy spending have been proposed [15], [16]. These schemes normally imply a time and an energy overhead: transitioning a memory block into and, especially, out of the dormant state consumes both energy and time. A team of researchers proposed an approach based on traces of memory accesses targeting the reduction of the static energy: in their method, the status of memory block is changed into the dormant state only if the cost of the energy overhead and decrease of performance can be amortized [17], [18].

This paper addresses the problem of partitioning the SPM address space, focusing on the reduction of energy consumption of the SPM. During the exploration of the search space, this approach evaluates also the idleness of the banks, exploiting the possibility of putting them temporarily into a *sleep* state, in order to reduce the static energy consumption. Different from previous techniques that start from the execution trace of the application [3], [7], [8], [17], [18], our approach starts from the behavioral specification, deciding in a preliminary phase the data assignment to the memory layers and the mapping of signals to the physical memories. When tested for several data-intensive applications, whose behavioral specifications contain multidimensional arrays as main data structures, the proposed approach proved to be computationally fast even for SPMs of a larger size, being able to explore banking solutions with a larger number of partitions. The rest of the paper is organized as follows. Section 2 briefly explains the preliminary code analysis and the assignment of data to the memory layers. Section 3 presents the SPM partitioning algorithm that explores the idleness of the banks. Section 4 addresses implementation aspects and experimental results. Section 5 states the main conclusions of this work.

2. PRELIMINARY MEMORY MANAGEMENT

The data-intensive algorithms for (real-time) multimedia applications are typically specified in a high-level programming language, where the code is organized in sequences of loop nests whose boundaries are either constant or linear functions of the outer loop iterators, conditional instructions, whose logical expressions may be both data-dependent or data-independent and multidimensional signals whose array references have (possibly complex) linear indices. This class of specifications is often called *affine* due to the fact that they contain array references whose indexes are affine functions of the loop iterators [19].

Figure 1 shows an illustrative code example with 6 nested loops. We call $M[x_1(i_1,\ldots,i_n)]\ \cdots\ [x_m(i_1,\ldots,i_n)]$ an *array reference* of an m-dimensional signal M, in the scope of a nest of n loops having the iterators $i_1, \ldots, i_n$. The *iterator space* signifies the set of all iterator vectors $\mathbf{i} = (i_1,\ldots,i_n) \in \mathbf{Z}^n$ in the scope of the array reference. The *index* (or *array*) *space* is the set of all index vectors $\mathbf{x} = (x_1, \ldots, x_m) \in \mathbf{Z}^m$ of the array reference. When the indices of an array reference are linear expressions with integer coefficients of the loop iterators, the index space consists of one or several *linearly bounded lattices* [20] (LBLs):

$$\{\, \mathbf{x} \;=\; \mathbf{T}\cdot\mathbf{i} + \mathbf{u} \;\in \mathbf{Z}^m \;\mid\; \mathbf{A}\cdot\mathbf{i} \geq \mathbf{b}\ ,\ \mathbf{i} \in \mathbf{Z}^n \}$$

where $\mathbf{x} \in \mathbf{Z}^m$ is the index vector of the m-dimensional signal and $\mathbf{i} \in \mathbf{Z}^n$ is the n-dimensional iterator vector.

Example: The array reference from the code sample in Fig. 1 $B[i][j][k][1089*p+33*q+r-1089*i-33*j-k+17969]$ can be represented by the following LBL:

$$\left\{ \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ -1089 & -33 & -1 & 1089 & 33 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \\ p \\ q \\ r \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 17969 \end{bmatrix} \right\}$$

where x, y, z, and w are the four indexes; the iterators satisfy the inequalities: $16 \leq i,\ j,\ k \leq 47$, $i-16 \leq p \leq i+16$, $j-16 \leq q \leq j+16$, $k-16 \leq r \leq k+16$ $\square$

The flow of the memory management methodology before SPM partitioning is:

Step 1: Decomposition of the array space of every indexed signal into disjoint linearly bounded lattices.

```
for (int i=0; i<=12; i++)
  for (int j=i+1; j<=2*i; j++)
    for (int k=-i+j; k<=i+2*j; k++)
    {   . . . .
      if ( 10*i+ 7*j+ 9*k <= 118  && 28*i-18*j+44*k >= 227
       && 26*i-10*j+14*k <= 147 && 64*i+26*j+20*k >= 351 )
        . . . .  = A[f(i,j,k)][g(i,j,k)][h(i,j,k)] ; // f, g, h linear functions
    }
```

Fig. 3. Loop nest with an array reference.

This decomposition can be performed analytically [21] by intersecting recursively the lattices of the signal's array references. This allows to compute *exactly* the number of memory accesses to the different parts of the arrays. For instance, the central partition C in the decomposition of signal A's array space – shown in Fig. 2 – gets 1,659,473,920 *read* accesses, out of the total of 2,355,167,232.

Step 2: Signal assignment to the memory layers: select the lattices having the highest numbers of memory accesses, whose total size does not exceed the maximum SPM size (assumed to be a design constraint), and assign them to the on-chip SPM. The other signals will be assigned to the off-chip DRAM. Afterwards, we compute mapping functions to the memory layers for every lattice of signals [21].

3. ALGORITHM FOR SPM BANKING BASED ON ANALYSIS OF DATA IDLENESS

Since information on *data idleness* is relevant in our context, we'll start by explaining the computation of the idleness of an array reference from the specification code. When are the elements of the array reference accessed and when they are not during the code execution? Figure 3 shows a code with one array reference $A[f(i,j,k)][g(i,j,k)][h(i,j,k)]$ within the scope of three nested loops and an *if* statement. Out of the 2392 iterations, the first one accessing the array reference is $(i,j,k)_1 = (2,3,8)$ and the last one is $(i,j,k)_2 = (3,4,6)$. Relative to the lexicographic order,[2] $(i,j,k)_1$ and $(i,j,k)_2$ are the minimum and the maximum iterator vectors. They can be computed by taking the integral polytope [22] determined by the 10 linear inequalities in i, j, and k derived from the loop boundaries and logical expression, and projecting it [23] on the i-axis of the iterator space; replacing the minimum, then the maximum i in the integral polytope, and doing an integral projection on the j-axis, and so on. Note that we considered here only loops with iterators that increase by a step of 1: if some loop iterator is decreasing, and/or some step is different from 1, simple linear transformations [24] can be applied to make all the loop iterators increasing by 1.

If we know whether the array reference is stored off-chip in a DRAM, or onto an SPM, as well as the number of cycles for a *read/write* operation, the time intervals in clock cycles when the array reference is not accessed can be determined. This idleness analysis can be done even more accurately: between $(i,j,k)_1$ and $(i,j,k)_2$ the array reference is not accessed during 4, then during 7 iterations. If we decide the minimum number of cycles such that a time interval be considered *idleness*, the number of intervals may increase.

[2] Let $\mathbf{x} = [x_1, \ldots, x_m]^T$ and $\mathbf{y} = [y_1, \ldots, y_m]^T$ be two m-dimensional vectors. The vector $\mathbf{y}$ is larger lexicographically than $\mathbf{x}$ (written $\mathbf{y} \succ \mathbf{x}$) if ($y_1 > x_1$), or ($y_1 = x_1$ and $y_2 > x_2$), ... , or ($y_1 = x_1, \ldots, y_{m-1} = x_{m-1}$, and $y_m > x_m$).

```
void LeakageAware_Banking (m, M, i, EnergyConsumed)
{  EnergyConsumed +=  δE_{m-1,m} ;
   ΔE = SPM_dyn ( Addr[i] , Addr[n] , RW) ;
   compute the intersection of the idleness intervals of lattices L_{i+1} , ... , L_n ;
   ΔE += SPM_st (Addr[i] , Addr[n] , active) + SPM_st (Addr[i] , Addr[n] , sleep) ;
   ΔE += SMP_transitionsEnergy ;  // between sleep state and active state
   if ( EnergyConsumed + ΔE < crtMinEnergy ) // a better partitioning solution ...
   {  crtMinEnergy = EnergyConsumed + ΔE ;   //  ... was detected: record it!
      crtBestPartitioning = top(SolutionStack) U { Addr[i] } ;
   }
   if ( m < M )   // if the maximum number of banks is not reached yet
                  //    then explore finer partitions
   {  push(SolutionStack, top(SolutionStack) U { Addr[i] }) ;
      for (int k = i+1; k<n; k++)
      {   ΔE = SPM_dyn ( Addr[i] , Addr[k] , RW) ;
          compute the intersection of the idleness intervals of lattices L_{i+1} , ... , L_k;
          ΔE += SPM_st (Addr[i],Addr[k],active) + SPM_st (Addr[i],Addr[k],sleep);
          ΔE += SMP_transitionsEnergy ;  // between sleep state and active state
          if  ( EnergyConsumed + ΔE + δE_{m,m+1} >= crtMinEnergy )  break ;
          LeakageAware_Banking (m+1, M, k, EnergyConsumed + ΔE) ;
      }
      pop(SolutionStack) ;
   }
}
```

Fig. 4. The core of the bank-partitioning algorithm.

Since each array reference is actually a lattice, for each LBL in the decomposition of the array space, we compute the time intervals (in clock cycles) when the lattice is *not* accessed. The small *idleness intervals* can be eliminated though for the following reason. In order to overcome the energy overhead entailed by the transition of a memory bank from the *active* state into the *sleep* state and back to the *active* state, the bank must remain in the *sleep* state at least a minimum number of clock cycles (otherwise, the economy of static energy is lesser than the energy overhead of the transitions). This minimum number of cycles can be estimated; typical values are in the order of hundreds of cycles [18]. The idleness intervals of each lattice are organized into an *interval tree* [25] as the depth of this data structure containing n intervals is $O(log\ n)$, entailing logarithmic complexity for interval operations.

In addition to the maximum number of SPM banks M, the inputs of the SPM partitioning algorithm are:

Input 1: An array *Addr* of ordered SPM addresses where the linearly bounded lattices, assigned to the on-chip SPM, are stored: the LBL L_k is mapped at the SPM addresses $Addr[k-1]$, ..., $Addr[k]-1$. The storage requirement after mapping for each lattice is computed at *Step 2*, Section 2.

Input 2: An array *RW* whose elements represent the numbers of *read/write* accesses for each lattice mapped onto the SPM.

Input 3: An array $\mathcal{E} = [\delta E_{12}\ \delta E_{23}\ \ldots\ \delta E_{M-1,M}]$ whose

Application	Address space	CPU$^{full}_{expl.}$ [3] M=4 [sec]	CPU M=8	Energy savings vs.		
				[3] (M=4)	[8]	monolithic
Gaussian blur filter	14,336	7,305.48	57.64	10.1 %	8.0 %	58.7 %
Motion detection	9,600	5,289.28	34.37	11.2 %	8.9 %	63.8 %
Motion estimation	1,024	736.52	6.12	9.4 %	7.5 %	54.6 %
Durbin's algorithm	512	247.05	2.89	9.8 %	7.7 %	52.5 %
SVD updating alg.	16,384	7,524.13	68.73	12.4 %	9.6 %	68.4 %
Voice coding kernel	2,048	1,377.56	11.87	10.8 %	8.9 %	61.4 %

Table 1. Experimental results for SPM banking, assuming a 32 nm technology.

elements $\delta E_{k,k+1}$ are the energy overheads resulting from moving from an SPM with k banks to one with $k+1$ banks. The elements of these arrays were estimated synthesizing decoding circuits, using the ECP family of FPGA's from Lattice Semiconductor. The Power Calculator from Lattice Diamond [26] was used to estimate the energy overheads.

The banking algorithm starts from the monolithic architecture and searches for the optimal partitioning of the SPM in no more than M memory banks, such that the borderlines between banks are addresses in the array *Addr* (hence, ensuring that each LBL is entirely stored in one bank). The function `SPM_dyn(start_addr, end_addr, RW)` uses CACTI 6.5 [27] and the array *RW* of numbers of accesses to compute the dynamic energy consumed in a bank delimited by the argument addresses. Similarly, the function `SPM_st` computes the bank static energy consumed in the *sleep* or *active* state.

A recursive function `LeakageAware_Banking` (Fig. 4), whose first formal parameter is the current number of banks (initially called for $m = 2$), searches for the optimal solution such that the first $m-1$ banks end at $Addr[i]$. `EnergyConsumed` accumulates the amount of energy consumed from the start of the SPM till the borderline $Addr[i]$. First, the function investigates the case when the m-th bank is between $Addr[i]$ and $Addr[n]$ – the end of the SPM. The idleness intervals of the lattices $L_{i+1}, \ldots, L_n$ assigned to this m-th bank are intersected in order to determine whether there are idleness intervals at the bank level. If this is the case, and if the idleness intervals exceed the size threshold, then the energy cost of the bank is computed taking into account the switch to the *sleep* state during the idleness intervals that are large enough. A time overhead of one clock cycle for the transition from *sleep* to *active* state is also applied, in accordance with simulated data on caches reported in [16].

If the maximum number of banks is not reached yet, then the function explores solutions with $m+1$ banks or more, considering the m-th bank only between $Addr[i]$ and $Addr[k]$ ($i < k$). A backtrack mechanism is incorporated before the recursive call (since the energy cost is monotonically increasing with the SPM size), to prevent the search towards more expensive partitioning solutions. A solution stack is used to keep track of the exploration of the solution space. The *output* of the algorithm is an array of SPM addresses delimiting the banks, and the corresponding energy consumption.

4. EXPERIMENTAL RESULTS

The banking algorithm described in Section 3 has been implemented in C++; the experiments have been carried out on a PC with an Intel Core 2 Quad 2.83 GHz processor. The main input of the software tool is an algorithmic specification of the signal processing application, expressed in a subset of the C language (see, for instance, Fig. 1). An interface to CACTI 6.5 [27] has been also implemented in order to obtain numeric data concerning static and dynamic power. (CACTI 6.5 supports 32, 45, 68, and 90 nm technologies.)

Table 1 shows experimental results for various benchmarks. Column 2 displays the size of the SPM address space. Column 3 reports the computation times for a full exploration, implemented as the technique presented in [7] targeting energy reduction, setting to 4 the maximum number of banks and using CACTI 6.5 [27] for power estimation. Column 4 reports the computation times in seconds for our banking algorithm where M was set to 8 – a value that proved impossible for [7]. Columns 5 and 6 show the savings of energy consumption of our algorithm versus [7] and [8]. Note that the dynamic programming technique of [8] yielded better results than [7] since that algorithm found energetically-better solutions having more than 4 banks. On the other hand, our algorithm found even better solutions since it could exploit the idleness intervals of the memory banks (which [8] does not do). The energy savings versus a monolithic SPM are also displayed in Column 7. The additional exploration constraint – that no LBL can cross a bank boundary – ensures the effectiveness of our approach for $M \geq 4$, reducing significantly the search space but typically yielding near-optimal solutions.

5. CONCLUSIONS

This paper has addressed the problem of energy-aware SPM banking, by exploiting the possibility of putting temporarily the banks into a *sleep* state, in order to reduce the static energy consumption. Different from previous banking algorithms that start from the execution trace of the application, this novel approach starts with an idleness analysis of the behavioral specification, deciding in a preliminary phase the data assignment to the memory layers, followed by the mapping of signals to the off-chip and on-chip memories.

6. REFERENCES

[1] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory : A design alternative for cache on-chip memory in embedded systems," in *Proc. 10th Int. Workshop on Hardware/Software Codesign*, Estes Park CO, May 2002, pp. 73-78.

[2] M. Kandemir, J. Ramanujam, M.J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic management of scratch-pad memory space," in *ACM/IEEE Design Automation Conf.*, Las Vegas NV, June 2001, pp. 690-695.

[3] A. Macii, L. Benini, and M. Poncino, *Memory Design Techniques for Low Energy Embedded Systems*, Boston: Kluwer Academic Publishers, 2002.

[4] M. Verma and P. Marwedel, *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*, Springer, 2007.

[5] P. R. Panda, F. Catthoor, N. Dutt, K. Dankaert, E. Brockmeyer, C. Kulkarni, and P. G. Kjeldsberg, "Data and memory optimization techniques for embedded systems," *ACM Trans. Design Automation of Electr. Syst.*, vol. 6, no. 2, pp. 149-206, Apr. 2001.

[6] F. Balasa, I.I. Luican, H. Zhu, and D.V. Nasui, "Automatic generation of maps of memory accesses for energy-aware memory management," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing*, pp. 629-632, Taipei, Taiwan, Apr. 2009.

[7] L. Benini, L. Macchiarulo, A. Macii, and M. Poncino, "Layout-driven memory synthesis for embedded systems-on-chip," *IEEE Trans. VLSI Syst.*, vol. 10, no. 2, pp. 96-105, Apr. 2002.

[8] F. Angiolini, L. Benini, and A. Caprara, "An efficient profile-based algorithm for scratchpad memory partitioning," *IEEE Trans. CAD*, vol. 24, no. 11, pp. 1660-1676, Nov. 2005.

[9] U. Ko, P. T. Balsara, and A. K. Nanda, "Energy optimization of multilevel cache architectures for RISC and CISC processors," *IEEE Trans. VLSI Syst.*, vol. 6, no. 2, pp. 299-308, June 1998.

[10] W. Shiue and C. Chakrabarti, "Memory exploration for low power embedded systems," *Proc. 36th ACM/IEEE Design Automation Conf.*, pp. 140-145, New Orleans LA, June 1999.

[11] V. Delaluz, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, A. Sivasubramaniam, and I. Kolcu, "Compiler-directed array interleaving for reducing energy in multi-bank memories," in *Proc. Asia & South Pacific Design Automation Conf.*, Bangalore, India, Jan. 2002, pp. 288-293.

[12] O. Ozturk and M. Kandemir, "Nonuniform banking for reducing memory energy consumption," in *Proc. ACM/IEEE Design Automation and Test in Europe*, Munich, Germany, Mar. 2005, pp. 814-819.

[13] A. Papanikolaou, H. Wang, M. Miranda, F. Catthoor, and W. Dehaene, "Reliability issues in deep submicron technologies: time-dependent variability and its impact on embedded system design," in *IFIP Int. Federation for Information Processing*, vol. 249, Springer, 2007, pp. 119-141.

[14] M. Kandemir, M.J. Irwin, G. Chen, and I. Kolcu, "Compiler-guided leakage optimization for banked scratch-pad memories," *IEEE Trans. VLSI Syst.*, vol. 13, no. 10, pp. 1136-1146, Oct. 2005.

[15] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *Proc. Symp. Computer Arch.*, June 2001, pp. 240-251.

[16] K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: Simple techniques for reducing leakage power," in *Proc. Symp. Computer Architecture*, May 2002, pp. 148-157.

[17] O. Golubeva, M. Loghi, M. Poncino, and E. Macii, "Architectural leakage-aware management of partitioned scratchpad memories," in *Proc. ACM/IEEE Design Automation and Test in Europe*, Nice, France, Apr. 2007, pp. 1665-1670.

[18] M. Loghi, O. Golubeva, E. Macii, and M. Poncino, "Architectural leakage power minimization of scratchpad memories by application-driven subbanking," *IEEE Trans. Computers*, vol. 59, no. 7, pp. 891-904, July 2010.

[19] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. V. Achteren, and T. Omnes, *Data Access and Storage Management for Embedded Programmable Processors*, Springer, 2010.

[20] L. Thiele, "Compiler techniques for massive parallel architectures," in *State-of-the-art in Computer Science*, P. Dewilde (ed.), Kluwer Acad. Publ., 1992. Kluwer Acad. Publ., 1992.

[21] F. Balasa, H. Zhu, and I.I. Luican, "Signal assignment to hierarchical memory organizations for embedded multidimensional signal processing systems," *IEEE Trans. VLSI Systems*, vol. 17, no. 9, pp. 1304-1317, Sept. 2009.

[22] Ph. Clauss and V. Loechner, "Parametric analysis of polyhedral iteration spaces," *VLSI Signal Processing*, vol. 19, no. 2, pp. 179-194, 1998.

[23] S. Verdoolaege, K. Beyls, M. Bruynooghe, and F. Catthoor, "Experiences with enumeration of integer projections of parametric polytopes," in *Compiler Construction: 14th Int. Conf.*, R. Bodik (ed.), vol. 3443, Springer, 2005, pp. 91-105.

[24] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*, Morgan Kaufmann Publ., 2001.

[25] M. De Berg, O. Cheong, M. van Krefeld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, Springer, 2010.

[26] *Lattice Power Calculator* [Online]. Available: http://www.latticesemi.com/prod-ucts/designsoftware/diamond/index.cfm

[27] *CACTI 6.5* [Online]. http://www.cs.utah.edu/∼rajeev/cacti6/