Activity Clustering for Leakage Management in SPMs*

M. Kandemir, G. Chen, F. Li, M. J. Irwin
Computer Science and Engineering Department
Pennsylvania State University
University Park, PA 16802, USA
{kandemir,gchen,feli,mji}@cse.psu.edu

I. Kolcu
Compuation Department
UMIST
Manchester M60 1QD, UK
ikolcu@umist.ac.uk

Abstract

This paper we proposes compiler-based leakage optimization strategy for on-chip scratch-pad memories (SPMs). The idea is to keep only a small set of SPM regions active at a given time and pre-activate SPM regions based on the compiler-extracted data access pattern. Our strategy, called activity clustering, increases the length of the idle periods of SPM regions by clustering accesses to a small set of regions at a time. It thus allows an SPM to take better advantage of the underlying leakage optimization mechanism.

1. Introduction

One of the problems with ever-scaling process technology is leakage consumption. Leakage consumption of on-chip memory components (e.g., caches) is particularly problematic since these components typically occupy a significant portion of die area. Prior research that addresses leakage consumption of on-chip memory components have exclusively focused on caches. However, recent trends in embedded computing systems indicate that software-managed scratch-pad memories (SPMs) are being increasingly employed in embedded devices.(e.g., [1, 2]) accommodate SPMs. This paper tries proposes a compiler-based solution for reducing leakage energy consumption of SPMs. The proposed solution employs a code analysis and restructuring framework that clusters the SPM accesses into a small set of SPM regions (unit of leakage control) at a given time. This strategy, called *activity clustering*, tends to increase the length of the idle periods of the SPM regions and thus allows them to take better advantage of the underlying leakage optimization mechanism employed such as [5] and [4]. While different implementations of activity clustering idea are certainly possible, in this paper, we focus on an Omega Library based implementation, a polyhedral tool from the University of Maryland [3].

* This work is supported in part by NSF Career Award #0093082 and a fund from GSRC.

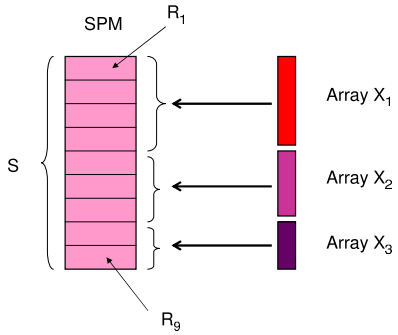

Figure 1. Mapping of three different arrays onto an SPM with nine regions.

2. Omega-Based Data Access Clustering

Let us assume that $\mathcal{S}$ represents the set of array elements assigned to the SPM at a given time, and that the SPM can hold at most s elements, which can be listed as $\mathcal{S}(1), \mathcal{S}(2), \mathcal{S}(3), \cdots, \mathcal{S}(s-1), \mathcal{S}(s)$. Therefore, any update of the SPM can be envisioned as loading new values to one or more elements of $\mathcal{S}$. We divide SPM into R regions of equal size (s/R) and R_i is used to represent the elements in the i^{th} region. In other words, R_i contains the elements $\mathcal{S}((i-1)s/R+1)$ through $\mathcal{S}(is/R)$, as shown in Figure 1. We further assume that the underlying hardware leakage control mechanism operates at a *region granularity*. When a region is idle for a certain period of time, it is placed into the low-leakage mode (also called the low-power mode). While in the low-power mode, a region consumes much less leakage energy than the active (fully operational) mode and it also preserves the contents of the region. A region in the low-power mode will be *reactivated* upon its next access, which incurs both performance and energy overheads. Our goal is to maximize the length of the idle periods of regions. Therefore, we would like to restructure the application code such that the accesses (iterations)

that touch the same SPM region are clustered to the greatest extent possible, allowed by the intrinsic data dependences in the code.

For each loop iteration, we associate an R-bit tag, which captures the SPM regions accessed by that iteration. For example, if we have eight regions (i.e., R = 8) and a particular loop iteration accesses only the first and last regions, the corresponding tag is 10000001. As another example, under the same region partitioning, if a loop iteration accesses all the regions except the last one, its tag is 11111110. Assuming that each loop iteration makes at least one SPM access, we have a total of $2^R - 1$ possible tags.

Figure 2 shows the sketch of our approach. We traverse the set of possible tags one-by-one starting with $000\cdots001$ and ending with $111\cdots111$. In visiting a tag, we schedule all loop iterations (from all nests) that has this tag, to the extent allowed by data dependences and then move to the next tag and schedule the loop iterations associated with it and so on. It is important to note that, based on the dependences in the code being restructured, a tag may need to be visited more than once. There are two energy benefits of this approach. First, since we cluster together the iterations that have the same tag, these iterations access the same set of regions and the remaining (unaccessed) regions can be put in the low-leakage mode. Second, when we need to move to a new tag, we select the one that has minimum Hamming Distance from the last tile processed. This obviously tends to minimize the state changes (active or low-leakage status) of the regions and helps further increase energy savings.

We cluster loop iterations using loop splitting, a compiler transformation that divides a loop into multiple loops. Specifically, for a given loop nest L and a region R_1, we split L into two loop nests, L_1 and L_2, such that all the iterations of L_1 accesses array elements in R_1, while none of the iterations of L_2 accesses any array element in R_1. After that, we further split the resulting loop nests, L_1 and L_2, with respect to another region R_2. We repeat this procedure until all the regions have been considered. Finally, we get a set of loop nests, each of which accesses a certain set of regions. We tag these loop nests based on the regions they access. The procedure for splitting loop nest L can be explained as follows. Let us assume a loop nest contains the following n array accesses:

$$X_1[f_1(\vec{I})], X_2[f_1(\vec{I})], ..., X_n[f_n(\vec{I})],$$

where $\vec{I}$ is the iteration vector of loop nest L, and function f_i maps iteration vector $\vec{I}$ to the subscribe of array X_i. Let us further assume that region R contains the following array elements:

$$X_1[\vec{V}_1, \vec{U}_1], X_2[\vec{V}_2, \vec{U}_2], ..., X_n[\vec{V}_n, \vec{U}_n],$$

where $X_i[\vec{V}_i, \vec{U}_i]$ is the set of elements of array X_i whose subscribe $\vec{S}$ is within range $[\vec{V}_i, \vec{U}_i]$, i.e., $\vec{V}_i \leq \vec{S} \leq \vec{U}_i$,

```
T = "00....0" // initially, all regions are turned off
for each loop nest L in program P {
  // clustering iterations of L
  S = {L}; t[L] = empty string;
  for each region R {
    S' = φ
    for each loop nest N ∈ S {
      split N into two loop nests, N1 and N2
        such that all the iterations of N1 access region R and
        none of the iterations of N2 accesses region ;
      t[N1] = t[N]+ "1"; t[N2] = t[N]+ "0";
      S' = S' ∪ {N1, N2}
    }
    S = S';
  }
  // schedule the clustered loop iterations
  while S ≠ φ {
    select loop nest N from S such that
      the Hamming Distance between t[N] and T is minimized;
    S = S − {N};
    schedule N;
    T = t[N];
  }
}
```

Figure 2. Sketch of our approach.

where "$\leq$" is defined as:

$$(u_1, u_2, ..., u_k)^{\mathrm{T}} \leq (v_1, v_2, ..., v_k)^{\mathrm{T}} \iff u_i \leq v_i (i = 1, 2, ..., k).$$

The set of loop iterations that access region R can be expressed as:

$$P = \bigcup_{i=1}^{n} \{\vec{I} \mid \vec{V}_i \leq f_i(\vec{I}) \leq \vec{U}_i\}.$$

On the other hand, the set of loop iterations that do not access region R can be expressed as:

$$Q = \bigcap_{i=1}^{n} \{\vec{I} \mid f_i(\vec{I}) < \vec{V}_i \text{ or } f_i(\vec{I}) > \vec{U}_i\}.$$

We assume that all f_i are affine functions[1]; therefore, we can use the Omega Library [3] to generate the loop nests whose iteration spaces are P and Q.

References

[1] ARM9E Microprocessor Core Family. http://arm.convergencepromotions.com/ catalog/ 118.htm

[2] ST100 DSP Cores Architecture Overview. http://www.st.com/stonline/ prodpres/ dedicate/ st100/ overview/ overview.htm

[3] Omega Project. http://www.cs.umd.edu/projects/omega/

[4] K. Flautner, N. Kim, S. Martin, D. Blaauw, T. Mudge. Drowsy Caches: Simple techniques for reducing leakage power. Proceedings of the 29th International Symposium on Computer Architecture, Anchorage, AK, May 2002.

[5] S. Kaxiras, Z. Hu, M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. Proceedings of the 28th International Symposium on Computer Architecture, Sweden, June 2001.

1 An affine function $f(\vec{I})$ can be expressed as $f(\vec{I}) = Q\vec{I} + \vec{q}$, where Q is a constant matrix, and $\vec{q}$ is a constant vector.