SPM Management Using Markov Chain Based Data Access Prediction*

Taylan Yemliha
Syracuse University, Syracuse , NY

Shekhar Srikantaiah, Mahmut Kandemir
Pennsylvania State University, University Park, PA

Ozcan Ozturk
Bilkent University, Ankara, Turkey

***Abstract*— Leveraging the power of scratchpad memories (SPMs) available in most embedded systems today is crucial to extract maximum performance from application programs. While regular accesses like scalar values and array expressions with affine subscript functions have been tractable for compiler analysis (to be prefetched into SPM), irregular accesses like pointer accesses and indexed array accesses have not been easily amenable for compiler analysis. This paper presents an SPM management technique using Markov chain based data access prediction for such irregular accesses. Our approach takes advantage of inherent, but hidden reuse in data accesses made by irregular references. We have implemented our proposed approach using an optimizing compiler. In this paper, we also present a thorough comparison of our different dynamic prediction schemes with other SPM management schemes. SPM management using our approaches produces 12.7% to 28.5% improvements in performance across a range of applications with both regular and irregular access patterns, with an average improvement of 20.8%.**

I. Motivation

Scratchpad memory (SPM), is a small, high-speed on chip data memory (SRAM) that is physically addressed but mapped into the virtual address space. The advantages of on-chip scratchpad memory over a conventional hardware managed on-chip cache is two fold. Firstly, references to a cache are subject to conflict, capacity and compulsory misses, while references to scratchpad guarantee that they will result in a hit, as data movements are managed by software. Secondly, scratchpads are accessed by direct addressing. This mitigates the overheads of expensive hardware cache tag comparison, typically present in set associative caches. However, exploiting these advantages of SPMs is possible only when we have appropriate compiler analysis techniques to effectively analyze the data access patterns exhibited by the application code and identify the frequently reused data to be maintained in limited scratch pad memory space that is available.

While there are numerous publications ([1], [2], [3], [4], [5], [6]) that focus on SPM management for programs with regular array accesses, only a few prior studies have considered irregular accesses. What we mean by "irregular accesses" in this paper are data accesses that cannot be statically resolved at compile time. Two examples of such irregular accesses are illustrated in Figure 1. In (a), a pointer is used to access data elements within a loop. Since in general it may not be possible to completely resolve pointer accesses statically, the compiler may not be able to determine which data elements will be accessed at runtime. Similarly, in (b), the set of elements accessed from array A depends on the contents of index array X, which may not be known in general until runtime. In both these cases, it is not possible at compile time to determine the best set of elements to place into the SPM.

However, we want to point out that the lack of static analyzability does *not* necessarily mean lack of locality in data accesses. Consider, for example, Figure 1(b) again. Within the main loop of this code fragment (loop t), the same array elements may be reused over and over again. Consequently, based on the contents of this index array, accesses to array A can also exhibit high levels of data reuse, although this is not evident at compile time. To be more specific, assuming N is 20 for illustrative purposes, if the contents of array X happen to be $\{8, 3, 6, 3, 3, 17, 18, 3, 3, 3, 6, 8, 18, 18, 17, 6, 8, 8, 6, 18\}$, the same five elements of array A ({A[3], A[6], A[8], A[17], A[18]}) are accessed repeatedly by loop t. Therefore, if somehow this pattern can be captured dynamically (during the course of execution), via a compiler inserted code, significant performance gains can be achieved. In this paper, we present and evaluate a novel approach to this problem. Specifically, targeting data-intensive applications with irregular memory access patterns, this paper makes the following contributions:

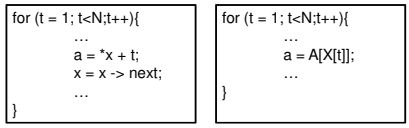

Fig. 1. Sample irregular access patterns. (a) Irregular access to data by pointers (b) Irregular access to data by indexed array expression.

- We propose a Markov Chain (MC) based data access pattern prediction scheme. The goal of this scheme is to predict the next data block to be accessed by execution, given the current data block access.
- We present a compiler-based code restructuring scheme that employs this MC based approach. This scheme transforms a loop into two sub-loops. The first sub-loop forms the training part and is responsible for constructing a MC based memory access pattern prediction model. The second sub-loop is the prefetching part where data is prefetched into the SPM based on the MC based prediction model constructed in the training part.
- We quantify the benefits of this approach using seven data-intensive applications. Five of these applications have irregular data accesses and two have regular data accesses. Our experimental results show that the proposed MC based scheme is very successful in reducing execution time for all seven applications. We also present the results from our sensitivity experiments, and compare our approach to several previously proposed SPM management schemes.

II. Related Work

Scratch-pad memories (SPMs) have been widely used in both research and industry, focusing mainly on the management strategies such as static versus dynamic and instruction SPM versus data SPM

*This work is supported in part by NSF Grants 0702519 and 0720749, Microsoft Research and GSRC.

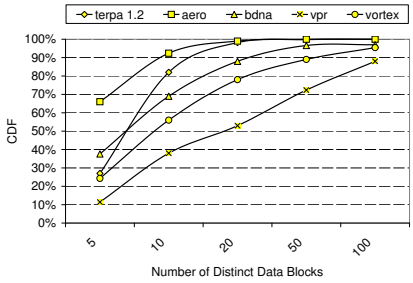

Fig. 2. CDF of the number of distinct data elements accessed by a reference.

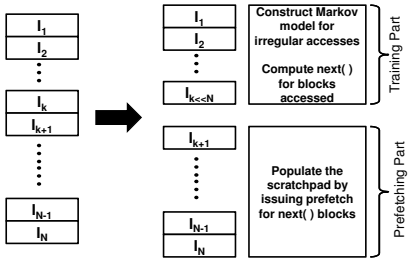

Fig. 3. High level view of our approach depicting the division of iterations into training part and prefetching part and associated transformation procedures.

[7], [8]. Egger et al [9] present a dynamic SPM allocation strategy targeting a horizontally partitioned memory subsystem for processors in the embedded system domain. In [10], authors propose a fully automatic dynamic SPM management technique for instructions, where required code segments are being loaded into the SPM on demand at runtime. Puaut and Pais [11] present an algorithm for off-line selection of the contents of on-chip memories. Li et al [4] employ a compiler-based memory coloring technique to allocate the arrays of a program onto an SPM. Golubeva et al [12] tackle the SPM management problem from a leakage energy perspective. Nguyen et al [5] present an SPM allocation scheme that does not require any compiler support for interpreted-language based applications such as Java. In [13], authors present a compile-time method for allocating heap data to SPM. Nguyen et al [14] discuss an SPM allocation scheme targeting a scenario where SPM capacity is unknown at compile time. This compiler method provides portability to different processor implementations with different SPM sizes.

Some embedded array-intensive applications do not have regular access patterns that can easily be analyzed by static techniques. For such applications, conventional SPM management schemes will fail to produce the best results and will prevent allocating the SPM efficiently [15], [16], [17]. To tackle this problem, Absar et al [15] propose a compiler-based technique for analyzing irregular array-access, and mapping such arrays to the SPM. On the other hand, Chen et al [16] present an approach for data SPMs, where the task of optimization is divided between compiler and runtime. Cho et al [17] present a profiling based technique that generates a memory access trace. This trace, then, is used to identify the data placement within the SPMs. While [15] and [16] can handle only irregular accesses due to indexed array expressions, our approach can handle pointer codes as well. Also, as against [17], we do not use profile data, and instead use compiler support to capture runtime behavior and exploit it. Since [15], [16] and [17] are the most relevant prior works to this paper, in our experiments we compare our approach to these three approaches.

III. Our Approach

A. Hidden Data Reuse in Irregular Accesses

As stated earlier, the main motivation for our work is the fact that the lack of compile-time analyzability does *not* necessarily mean lack of locality in data accesses. To quantify this, we collected statistics on five data-intensive applications that are hard to analyze using compile-time techniques alone. The graph in Figure 2 plots the CDF of the number of distinct data elements accessed by a reference (when all references are considered). A point (x, y) in this plot indicates that $y\%$ of the accesses made by the reference are to x or fewer distinct data elements. For example, for application vpr, 11.4% of the memory accesses made by a reference are to only 5 distinct data elements, that is, there is significant data reuse per reference. Unfortunately, due to irregular data accesses (*i.e.*, because of the way the code is written), this data reuse cannot be captured and exploited at compile-time.

We propose to use Markov Chains (MC) to capture and optimize such data accesses at runtime. Figure 3 shows the high-level view of our approach. For each loop nest of interest, the first few loop iterations are used to build a Markov Model, which is used to fill a compiler-generated data structure, so that the remaining iterations can take advantage of the available SPM. In the remainder of this section, we present the details of our MC based approach.

We can think of MC as a finite state machine such that if the machine is in state q_i at time i, then the probability that it moves to state q_{i+1} at time $i+1$ depends solely on the current state. In our MC based formulation of the SPM optimization problem for irregular data accesses, each state corresponds to an access to a data block, i.e., a set of consecutive data elements that belong to the same data structure. The weight associated with edge (i, j), i.e., the edge that connects states q_i and q_j, is the probability with which the execution touches block q_j, right after touching data block q_i.

B. Different Versions

Figure 4 gives an example that shows the code transformation performed by our proposed approach. Our approach operates at a loop nest granularity, that is, it is given one loop nest at a time. It divides the given loop nest in two parts (sub-loops). The first part is the *training part* and its main job is to fill a compiler-generated data structure, which is subsequently used in the second part. This data structure represents the MC based model of data accesses encountered in the training part. The second part, called the *prefetching part*, uses this model to issue prefetch requests. Each prefetch request brings a new block to the SPM ahead of time, i.e., before it is actually needed. Therefore, at the time of access, the execution finds that block in the SPM and this helps improve performance and power, though in this work only performance benefits have been evaluated. We can see from Figure 4 that the first k iterations ($k << N$) are used for the training part. The remaining iterations are tiled into tiles of t iterations each, and prefetching for the each tile is performed at the beginning of the tile. Selection of t is done such that off-chip memory access latency can be hidden.

We now want to discuss the functionality of next(.). For a given data block B_i, next(B_i) gives the set of blocks that are to be prefetched within the prefetching part. Clearly, there are many different potential implementations of next(.). Below, we summarize the implementations evaluated in this work, using the sample Markov Model illustrated in Figure 5:

- A1: It returns only one block which corresponds to the edge in the Markov Model with the highest weight (transition of

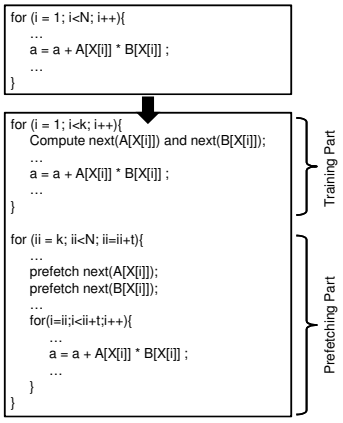

Fig. 4. Code transformation depicting the training and prefetching parts in our approach.

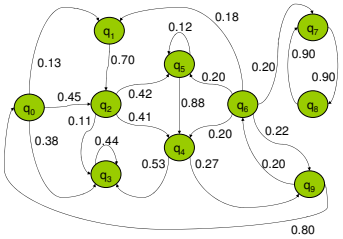

Fig. 5. Sample Markov Model. Note that this figure shows only high transition probabilities; low ones are omitted for clarity. Each state q_i denotes an access of block B_i, and the weight associated with edge (i, j),between states q_i and q_j corresponds to the probability with which the execution touches block q_j, right after touching data block q_i.

probability). For example, in Figure 5, if the current data block being accessed is B_0, next(.) will return B_2. While this implementation is simple and can be effective in many cases, it may not perform well in every case, as even the highest weight may not be very high. For example, in the same transition diagram, if the current block being accessed is B_6, the prefetched block will be B_9, but, the corresponding probability is only 22%.

- A2: This alternative is a variant of the previous one and returns a block only if the corresponding transition probability is the largest among all blocks and above a preset threshold value (δ). In this way, we guarantee that the likelihood of the prefetched block being accessed by execution is high. Again, in the example of Figure 5, if the current data block is B_0 and δ is 50%, no block is prefetched under the A2 scheme. As another example, if the threshold value is 50%, next(B_4) is B_3.
- A3: The third alternative prefetches k blocks with the largest transition probabilities. In our example of Figure 5, next(B_2) would be $\{B4, B5\}$ if k is set to 2.
- A4: The last alternative we experiment with selects k blocks to prefetch such that the cumulative sum of the transition probabilities of these blocks is larger than a preset threshold value (δ). As an example, if δ is set to 80%, next(B_4) would be $\{B3, B9\}$ under this alternative. Notice that under this scheme next(.) set can contain any number of blocks.

It is to be noted that some of these alternatives work with parameters, the values of which may be critical to their success. More specifically, schemes A2 and A4 use a threshold parameter (δ), whereas A3 operates with a k parameter.When there are multiple options (combination) that lead to the same threshold value of δ, the A4 alternative selects the combination with minimum number of blocks. The important point to note is that the code shape shown in Figure 4 does not change much with the particular scheme (alternative) adopted; the different schemes change only the contents of next(.).

After determining the next(.) blocks in the training part, it is important to efficiently insert the prefetch instructions to the scratchpad memory for each next block to be used in the successive iterations of the prefetching part. We use an algorithm similar to [19] in order to insert prefetch instructions in the code to prefetch data into the SPM. The prefetch distance (the time difference between time of prefetch and time of first use of a data block) is an important parameter that is determined using the approach in [19], which can be computed as a simple function of the estimated time for a single prefetch and the estimated cycle of each loop iteration. Note that, although this compiler prefetch algorithm is efficient, the choice of the compiler algorithm for prefetching is orthogonal to the problem of predicting the next(.) blocks. It is also important to note that, the next(.) set of each block could potentially consist of more than one block (depending on whether A1, A2, A3 or A4 is being used), and in such a case, we conservatively insert prefetch for each block in the next(.) set.

TABLE I
BENCHMARKS AND THEIR CHARACTERISTICS.

Name	Data Size (MB)	Dominant Access Type
terpa 1.2	3.88	index arrays
aero	5.27	index arrays
bdna	5.9	index arrays
vpr	4.43	pointer based
vortex	2.71	pointer based
oa_filter	2.86	regular
swim	3.76	regular

TABLE II
SIMULATION PARAMETERS.

CPU	2-issue embedded core
SPM Capacity	64KB
Block Size	1KB
SPM latency	2 cycles
Off-chip memory latency	200 cycles

A fully adaptive scheme that selects these parameters dynamically can be expensive to implement. Therefore, we fix the values of these parameters at compile time. Obviously, a programmer can experiment with different values of parameters in a given alternative, and select the best performing one for the application at hand.

Another potential issue is what happens when our approach is applied to code with regular data access patterns. While our approach works with such codes as well, the results may not be as good as those that could be obtained using a conventional (static) SPM management scheme. This is due to the overheads incurred by our approach (mainly within the training part) at runtime. In order to quantify this behavior, we also applied our approach to two codes with regular data access patterns, and reported the results in Section IV. Note that a compiler implementation can select between our approach and a conventional static scheme, depending on the application code at hand. This is possible because a compiler can infer that a given reference is irregular, though it cannot fully analyze the irregularity it detects.

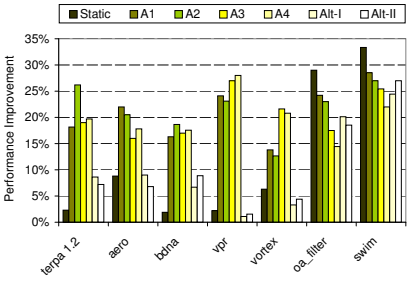

Fig. 6. **Percentage improvement (reduction) in execution cycles under different schemes.**

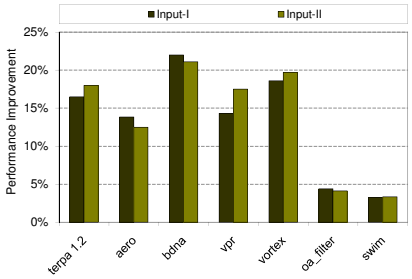

Fig. 7. **Additional performance improvements our approach brings over the approach in [17].**

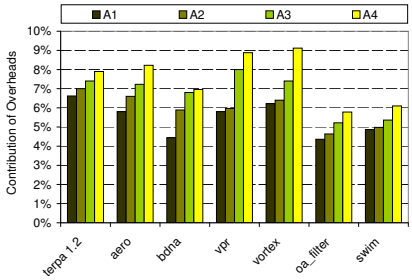

Fig. 8. **Contribution of overheads to the overall execution cycles.**

IV. Experiments

We implemented our proposed approach using the SUIF compiler [21], and performed simulations with the schemes (A1 through A4) above as well as three SPM management schemes. To perform our simulations, we enhanced SimpleScalar [22]. The important simulation parameters and their default values are listed in Table II. The set of applications used in this study are given in Table I. In the following discussion, Static, Alt-I and Alt-II represent the schemes explained in [3], [16], and [15], respectively. The results of our schemes include both the training and prefetching parts, *i.e.*, all overheads of our schemes are captured. All the performance improvement results presented below are with respect to a version that uses a conventional (hardware managed) cache of the same size as the SPM capacity used in other schemes.

Our first set of results are present in Figure 6, and give the percentage improvement (reduction) in execution cycles under the different schemes explained above. Our first observation is that the average performance improvements brought by schemes Static, A1, A2, A3, A4, Alt-I, and Alt-II are 11.9%, 21.0%, 21.5%, 20.5%, 20.0%, 10.4% and 10.6%, respectively. We also note that our dynamic schemes (A1 through A4) generate much better savings than the static scheme for all five applications with irregular access patterns. This is expected as the static SPM management scheme in [3] can only optimize a few loop nests in these applications, namely, the nests with compile-time analyzable data access patterns, and the remaining loop nests remain unoptimized. In contrast, our approach, using the explained MC based model, successfully optimizes these applications. We also observe that our dynamic scheme improves performance for our two regular applications (oa_filter and swim) as well, though the results (savings) are not as good as those brought by the static scheme. This difference is mainly due to the runtime overheads incurred by our scheme as discussed earlier. However, as explained earlier in Section III-B, an optimizing compiler may choose between the static and dynamic schemes depending on the application code at hand.

Among our schemes, we observe that A2 generates better results than the rest in terpa 1.2. This is because the transition diagram for terpa 1.2 is very dense, and as a result, given a node, transition probabilities are almost equally distributed in many cases. This behavior in turn favors A2 over A1, as A2 is more selective in prefetching and does not perform useless prefetches. On the other hand, A3 and A4 issue too many prefetches in this application, and this contributes to the runtime overheads. In applications vpr and vortex, the extra overheads brought by A3 and A4 are compensated by their benefits (the increase in SPM hit rate as a result of more prefetches), and the overall performance is improved.

We now discuss how our approach compares against two previously-proposed schemes that try to address irregular data accesses. Alt-I tracks the statements that make assignments to index arrays and use these values to determine the minimum and maximum bounds of the data arrays. Since this scheme targets irregularity that arises from indexed array accesses, it does not offer a solution for pointer based applications, and consequently, it performs no better than the static scheme for our pointer applications (vpr and vortex). In fact, due to the overheads involved, Alt-I performs worse than the static scheme [3] in these two applications. The same observation goes for Alt-II as well, which also targets exclusively indexed array accesses. When the index array applications (terpa 1.2, aero, and bdna) are considered, our schemes are better than both Alt-I and Alt-II, thanks to the inherent locality exhibited by the indexed array based data accesses.

We also compared our approach (version A1) to the SPM management scheme in [17] which uses profile data to place data into the SPM. To do this, we profiled each application using an input set (Input-0) and then executed the same application using two different input sets, Input-I and Input-II, both of which are different from Input-0. The bar-chart in Figure 7 gives the additional performance benefits our approach brings over the scheme in [17]. The average improvement when considering all benchmarks is around 13.5%. The reason for this is that in irregular applications the input data used for execution can change the behavior of the application significantly. Therefore, any profile based method will have difficulty in optimizing irregular codes, unless the profile input is the same as the input used to execute the application.

Since our schemes (A1 through A4) incur runtime overheads, it is also important to quantify these overheads. Figure 8 gives the contribution of these overheads to the overall execution cycles in our applications. We observe that the overheads range between 4.4% and 9.1%, depending on the particular alternative. As expected, most overheads are incurred by the A4 alternative.

As noted earlier, different versions (A1 through A4) work with different parameters. Now, we quantify the impact of these parameters. Due to space constraints, we focus on A2 and A3 versions only. First, in Figure 9, we present the sensitivity of the A2 version to the threshold value (δ), for our irregular applications. It is easy to see from these curves that, for each application, there is an optimum threshold value (among those tested). Working with a smaller threshold value causes unnecessary prefetches to the SPM, while employing a larger threshold value suppresses a lot of prefetches, some of which could have been useful. Similarly, Figure 10 plots the sensitivity of the A3 version to parameter k. It can be seen that the different applications reach differently to varying k. For example,

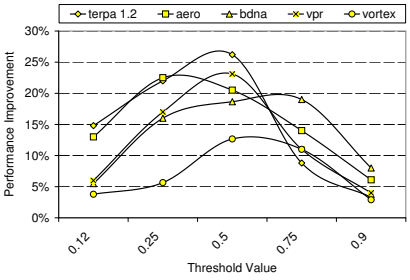

Fig. 9. **Sensitivity of the A2 scheme to the threshold value (δ).**

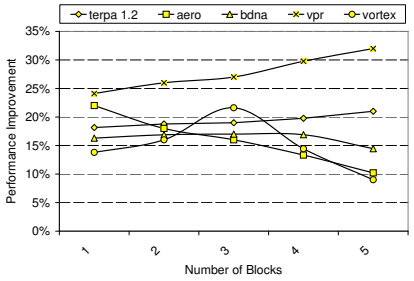

Fig. 10. **Sensitivity of the A3 scheme to parameter k.**

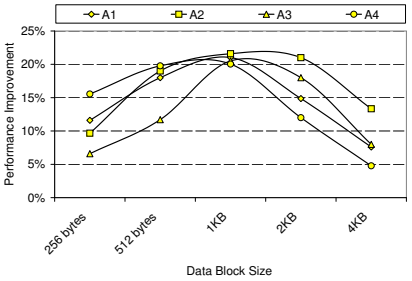

Fig. 11. **Sensitivity to the data block size.**

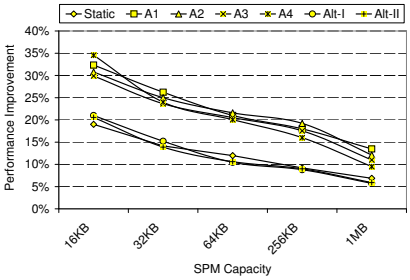

Fig. 12. Average improvement values across all applications.

terpa 1.2 and vpr take advantage of increasing k values, whereas aero's performance decreases as we increase k.

We now quantify, in Figure 11, the influence of the granularity of prefetch on our savings. Each curve in this plot represents the *average improvement* value (across all applications) under varying data block sizes. the default block size used in our experiments so far was 1KB (Table II). We see from these results that block size selection is a critical issue. For example, working with large blocks is not very useful as it causes frequent displacements from the SPM. While this argues for employing smaller blocks, doing so can lead to complex Markov models, which may be costly to maintain at runtime. In addition, small block sizes also increase the activity between the SPM and the off-chip memory, which can in turn affect overall performance. Considering these two factors, one has to make a careful choice for the block size.

It is also important to study the behavior of our scheme under different SPM capacities. The default SPM capacity used in our experiments is 64KB (Table II). The results plotted in Figure 12, which represent *average performance improvement* values across all applications, show that our dynamic scheme is consistently better than the remaining schemes for all SPM capacities tested. As can be seen, our performance improvements reduce a bit with increasing SPM capacities. This is expected as the presented results are values normalized with respected to the original case, i.e., the case with conventional hardware-managed cache. As the on-chip memory capacity (SPM or cache) is increased, the difference between our scheme and the original case gets reduced. It should also be noted however that, as the increase in data set size is usually much higher than increase in on-chip memory capacities, we can expect higher savings from our scheme in future systems.

V. Concluding Remarks

We proposed various schemes to predict irregular data accesses in data intensive applications using a Markov chain based model. Using such a data access pattern prediction model for prefetching data into scratchpad memory helps improve the performance of applications with irregular data accesses to a large extent. We observe that scratchpad memory management using our approaches produces 12.7% to 28.5% improvements in performance across a range of applications with both regular and irregular access patterns, with an average improvement of 20.8%. Our current work includes porting this SPM management scheme to a chip multiprocessor environment and testing its effectiveness using multithreaded applications.

References

[1] I. Issenin et al. "Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies," in *DAC*, 2006.
[2] M. Kandemir et al. "Compiler-directed scratch pad memory hierarchy design and management," in *DAC*, 2002.
[3] M. Kandemir et al. "Dynamic management of scratch-pad memory space," in *DAC*, 2001.
[4] L. Li et al. "Memory coloring: A compiler approach for scratchpad memory management," in *PACT*, 2005.
[5] N. Nguyen et al. "Scratch-pad memory allocation without compiler support for java applications," in *CASES*, 2007.
[6] M. Verma et al. "Data partitioning for maximal scratchpad usage," in *ASPDAC*, 2003.
[7] P. R. Panda et al. "Efficient utilization of scratch-pad memory in embedded processor applications," in *EDTC*, 1997.
[8] R. Banakar et al. "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *CODES*, 2002.
[9] B. Egger et al. "Scratchpad memory management for portable systems with a memory management unit," in *EMSOFT*, 2006.
[10] B. Egger et al. "A dynamic code placement technique for scratchpad memory using postpass optimization," in *CASES*, 2006.
[11] I. Puaut et al. "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison," in *DATE*, 2007.
[12] O. Golubeva et al. "Architectural leakage-aware management of partitioned scratchpad memories," in *DATE*, 2007.
[13] A. Dominguez et al. "Heap data allocation to scratch-pad memory in embedded systems," *JEC*, vol. 1, no. 4, 2005.
[14] N. Nguyenet al. "Memory allocation for embedded systems with a compile-time-unknown scratch-pad size," in *CASES*, 2005.
[15] M. J. Absar et al. "Compiler-based approach for exploiting scratch-pad in presence of irregular array access," in *DATE*, 2005.
[16] G. Chen et al. "Dynamic scratch-pad memory management for irregular array access patterns," in *DATE*, 2006.
[17] D. Cho et al. "Software controlled memory layout reorganization for irregular array access patterns," in *CASES*, 2007.
[18] I. Issenin et al. "Data reuse driven energy-aware mpsoc co-synthesis of memory and communication architecture for streaming applications," in *CODES*, 2006.
[19] T. C. Mowry et al. "Design and evaluation of a compiler algorithm for prefetching," in *ASPLOS-V*, 1992.
[20] V. Srinivasan et al. "A static filter for reducing prefetch traffic," in *CSE-TR-400-99, UMich*, 1999.
[21] M. W. Hallet al. "Maximizing multiprocessor performance with the SUIF compiler," *Computer*, vol. 29, no. 12, 1996.
[22] T. Austin et al. "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, 2002.