Analysis of Scratch-Pad and Data-Cache Performance Using Statistical Methods

Javed Absar*† and Francky Catthoor*
*IMEC vzw., Katholieke Universiteit Leuven, Belgium.
† STMicroelectronics Asia Pacific, Singapore.
{javed.absar, francky.catthoor}@imec.be

***Abstract*— An effectively designed and efficiently used memory hierarchy, composed of scratch-pads or cache, is seen today as the key to obtaining energy and performance gains in data-dominated embedded applications. However, an unsolved problem is – how to make the right choice between the scratch-pad and the data-cache for different class of applications? Recent studies show that applications with regular and manifest data access patterns (e.g. matrix multiplication) perform better on the scratch-pad compared to the cache. In the case of dynamic applications with irregular and non-manifest access patterns, it is however commonly and intuitively believed that the cache would perform better. In this paper, we show by theoretical analysis and empirical results that this *intuition* can sometimes be misleading. When access-probabilities remain fixed, we prove that the scratch-pad, with an optimal mapping, will always outperform the cache. We also demonstrate how to map dynamic applications efficiently to scratch-pad or cache and additionally, how to accurately predict the performance.**

I. INTRODUCTION

Multimedia and network applications are well-known to be highly data dominated [5]. To reduce memory-access related energy and performance costs in such applications, the memory hierarchy must be effectively designed (with respect to capacity and number of levels) and efficiently used [18][16] (i.e. data mapping to carefully exploit the hierarchy). Most multimedia and network platforms today have at least one, two or more levels of cache. Lately, software-controlled cache, also known as scratch-pad memory (SPM), have also been viewed as an alternative to the data and instruction cache [9][12][23].

SPM is software controlled. Therefore, the compiler or the application developer must make, and instrument, all the decisions about which data should reside on SPM at any time. This situation is quite different from the cache, where the hardware is in charge of exploiting the locality using a circuit that implements the *least recently used* algorithm. The hardware simplicity of SPM enables it to provide a low power-consumption and access-time number, on a per access basis, compared to cache [4]. Applications in which the data access pattern is regular and manifest (i.e. known at compile time) can be mapped easily and efficiently to SPM [1][9][11]. Such applications include matrix multiplication, filtering and large portions of audio and video compression algorithms. Since SPM is cheaper than cache, in such cases the better choice is, indeed, the SPM. If, however, the applications exhibits more dynamism – objects being accessed in a data-dependent and seemingly random fashion – mapping to SPM requires more ingenuity. Applications involving trees, heaps, tries, graphs and linked lists often exhibit this kind of dynamism. One may reckon that the cache would do a better job in such cases and, therefore, decide to let such objects be handled by the cache. This may or may not hold true. We, therefore, require more sound techniques for deciding between SPM and cache.

In this paper, we study the performance of dynamic applications, when mapped to SPM or cache, using models of *access probability*. This allows us to analyze and predict whether the cache would outperform the SPM, and by how much. We back our theoretical conclusions with empirical results. To the best of our knowledge, this is the first work that analytically compares the performance of SPM with cache. Previous studies [7][12] have focused on finding good mapping techniques for SPM. As such, the comparison of SPM performance with cache was limited to *simulation results*. Unfortunately, that does not provide broader insight.

The remainder of this paper is organized as follows. Section 2 gives a motivating example. Section 3 reviews related work. Section 4 gives a brief description of the probability model that we employ. Section 5 compares SPM and cache performance with this model. And last, section 6 presents empirical results.

II. MOTIVATING EXAMPLE

The function `search` in the program section below, does a spell-check by performing a search of the given `word`, of `n` letters, against its internal dictionary. The dictionary is implemented as a trie (tree with variable number of child-nodes) which enables fast searches.

```
typedef struct node{ //trie data-structure
 struct node *next, *down; char letter;
}Node;

bool search(Node *nptr, char *word, int n){
  for( i = 0 ; i < n ; i++ ){
   while(nptr && (nptr->letter < word[i])){
     nptr = nptr->next;
   }
   if(nptr){
    if(nptr->letter == word[i])
       nptr = nptr->down;
    else  return false; //word not found
   }else  return false; //word not found
  }
  return true;          //word found
}
```

0-7803-9451-8/06/$20.00 ©2006 IEEE.

This application can be mapped to the SPM or the cache. For the case of data-cache, since the tracking and migration of data to and from the cache is handled by hardware, no changes are required on the code. For the case of SPM, however, additional instructions are necessary to place some of the nodes on SPM. The rest of the nodes remain in external memory from where they are directly accessed. Access to the nodes mapped to SPM will be quick and energy efficient. If the nodes are mapped intelligently such that most of the search is to nodes on the SPM, then overall good performance and energy efficiency can be expected. The question is: Can the SPM really compete with the cache in such a dynamic and data-dependent application. We will address such questions both analytically and with empirical results.

As a side remark, note that pointer based data-structures (such as trie) allow placement (of some of the nodes) on SPM in a seamless way without requiring checks with each access. On the other hand, if we place parts of an array on SPM and the rest on the external memory, then each access to the array (using an index expression) will require first a check to see where that segment resides.

III. RELATED WORK

Banakar et al. [4] did a detailed study of the energy and area advantage of the SPM over the cache. On a per access basis, they found that a 2KB, 2-way cache consumed 4.57 nJ., while a 2KB SPM consumed only 1.53 nJ. Initial work by Panda et al. [17] on utilizing the SPM focused on scalars and highly used, small-sized arrays. Their approach was extended and improved upon by other researchers [3][21][23] who applied knapsack formulation and ILP solvers to find the best set of data objects (globals, stack variables and arrays) and program routines that would still fit into the SPM and yet save the highest amount of energy. For arrays larger than the SPM size, these solutions do not work well. However, for arrays accessed in a regular pattern – inside nested-loops by index expressions which are linear functions of the loop iterators – several additional SPM mapping techniques using data-space and iteration space tiling have been proposed [9][11][12]. They work well irrespective of the array size, and always outperform the cache.

Dominguez et al.[7] explain a technique for mapping dynamic data structures, e.g. linked lists and trees, to SPM. The size allocated in the SPM to each set, e.g. nodes of a particular tree, is made proportional to the overall number of access to that set. They do not compare between SPM and cache but are only concerned with finding the best allocation of the SPM for the different dynamic objects, each vying for space on the SPM.

For the cache, itself, there have been numerous studies to estimate its performance for regular and non-regular applications. Several studies have tried to quantify the cache performance by summarizing or analyzing actual memory access trace [2][20]. From analytical comparison perspective, however, trace analysis is not fruitful. On the other hand, the Independent Reference Model (IRM) of Rao [19][13] is more suited to our purpose of analyzing cache behavior for comparison with SPM. This model was recently extended [8][15] to algorithmic analysis. Rao's equations assume a given (specified) data-layout. However, the results in our paper allow conclusions to be drawn across all possible layouts.

IV. MODEL DESCRIPTION

Computations such as matrix multiplication generate *memory reference strings*, i.e. sequence of data memory addresses, that are regular and input-independent, and can be determined without even running the program. This enables good SPM mapping, and as such the SPM is able to outperform the cache for all these types of computations [12].

On the other hand, dynamic applications generate reference strings that are irregular and input-dependent. Therefore, predicting SPM and cache performance (hit-rate) with the techniques used for the regular case turns out to be unwieldy and complicated. In this paper we use statistical methods [22] to compare the SPM performance with the cache. We start by characterizing the reference string using the Independent Reference Model (IRM) [15][19].

Consider a set of objects $O_1, O_2, \ldots, O_n$ and a set of corresponding *access probabilities* $p_1, p_2, \ldots, p_n$. The reference string can be denoted as $r_1, r_2, \ldots, r_t, \ldots, r_N$, where r_t is the object referenced at time t. Under IRM:

$$Pr[r_t = O_i] = p_i \, , \, 1 \leq i \leq n \, , \, t > 0$$

That is, the probability of r_t being O_i is p_i. Though this model does not take into consideration the correlation between accesses, studies [10] show that the behavior modeled assuming *independent reference* gives results that are very close to those obtained using models that do indeed incorporate such correlation. We will also show this with empirical results which reconfirm that IRM is indeed able to accurately predict SPM and cache performance for data-structures such as trees where the access-pattern is clearly data-dependent and correlated.

Now, we will compute the cache hit-rate for the set of access probabilities given above. Suppose we have a cache with just one block and so it contains only the last object accessed. It can been shown [13] that the states of this cache forms a homogeneous Markov chain, where each state S_i is defined as having the object O_i inside the cache block. The equilibrium probability of state S_i equals p_i [19] and hence, the probability that object O_i is inside the cache, at anytime, equals p_i. Therefore, if object O_i is accessed, then the probability that the access results in a hit (object found in cache) equals p_i. Averaging the hit-rate across all possible accesses, we get the expected hit-rate η for a cache of size one as:

$$\eta = \sum_{i=1}^{n} p_i^2 \tag{1}$$

Embedded systems usually contain caches with low associativity to reduce the energy and area. The results that we derive are, therefore, in the context of the direct-mapped cache (DM-Cache). Caches with low associativity perform similar to the

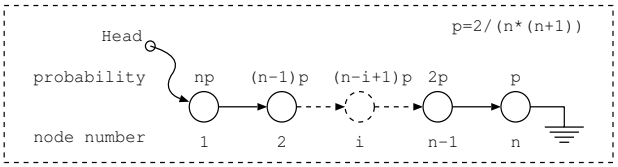

Fig. 1. Access-probabilities in a linked list, when each node is equally likely to be the target of the next search. A search starts at head and stops when target is found.

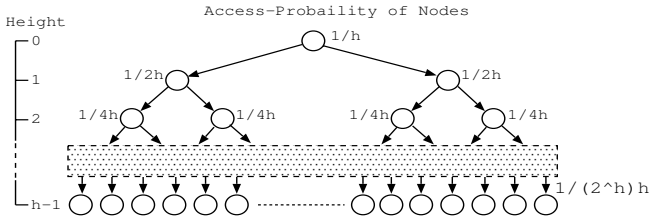

Fig. 2. Access-probabilities of nodes in a binary tree.

direct-mapped [19]. We will later present empirical results that confirm this as well.

The expected hit-rate for a DM-Cache of more than one block can be computed by placing the objects into disjoint groups. Assume that the DM-Cache contains m cache blocks, and each block can hold only one object. Let G_i denote the set of objects which map to cache block i. Out of the n objects $O_1, O_2, \ldots, O_n$, assume (for simplicity) that $k = n/m$ objects map to each cache block. Objects in G_i are denoted as $O_1(i), O_2(i), \ldots, O_k(i)$, and the corresponding probabilities as $p_1(i), p_2(i), \ldots, p_k(i)$, respectively. Let $D_i = p_1(i) + p_2(i) + \ldots + p_k(i)$. The following result, from Rao [19], gives the expected hit-rate of a DM-Cache:

$$\eta_{\text{DM}} = \sum_{i=1}^{m} \frac{1}{D_i} \sum_{j=1}^{k} p_j^2(i) \tag{2}$$

Essentially, a DM-Cache behaves as m disjoint, fully-associative caches, each of size one. So, Eq. 1 can be applied to each cache block, but with conditional probabilities $p_j(i)/D_i$. The overall hit-rate equals the weighted sum of hit-rates of each individual block. The weight for a block equals the probability that the next access would be to that block. For block i, this equals D_i.

Let us now see how to compute access probabilities. A linked list is shown in Fig. 1. Each node contains a key and some data. The search for a node, with a certain key, starts at the head and continues till that node is found. Each key is equally likely to be the target of the next search. Now, if the n^{th} node has access probability p, then the $(n-1)^{th}$ node has access probability $2p$. Reason: the $(n-1)^{th}$ node is referenced when the n^{th} node is the target of a search, and it is also referenced when it is, itself, the target of a search. Probabilities for all the nodes is shown in Fig. 1. It is also possible to assign probabilities in an application-specific way or based on profiling as we will show later.

Next, consider a binary search-tree. A search starts at the root and proceeds downward to the target leaf-node. From any parent, the search-path has an equal chance of moving to the left child-node or to the right child-node. Therefore, if the access probability of a parent is p, each child-node has access probability $p/2$. Fig. 2 shows the access probabilities of nodes in a binary tree.

V. ANALYTICAL STUDY

In this section, we analyze SPM and DM-Cache performance using the model discussed before.

To recapitulate, we have n objects $O_1, O_2, \ldots, O_n$ with access probabilities $p_1, p_2, \ldots, p_n$, respectively. Without loss of generality, let $p_1 \geq p_2 \geq \ldots \geq p_n$. To maximize the SPM hit-rate, objects $O_1, O_2, \ldots, O_m$, where m is SPM size, must be placed on SPM. The rest of the objects remain in the memory, from where they are accessed directly. Hence, each access to O_i, where $i > m$, constitutes a miss. The expected hit-rate of this optimal SPM mapping is:

$$\eta_{\text{SPM}} = \sum_{i=1}^{m} p_i \tag{3}$$

Let us now compare this with the hit-rate of a DM-Cache, also of size m, using Eq. 2. As before, objects $O_1(i), O_2(i), \ldots, O_k(i)$, with access probabilities $p_1(i), p_2(i), \ldots, p_k(i)$, respectively, map to cache block i. Again, without loss of generality, let $p_1(i) \geq p_2(i) \geq \ldots \geq p_k(i)$. Although we assume that exactly $k = n/m$ objects map to each block, it is not a limitation of the proof, but is done so as to simplify the notation.

Since $D_i = p_1(i) + p_2(i) + \ldots + p_k(i)$ in Eq. 2, we have $p_1(i) = D_i - \sum_{j=2}^{k} p_j(i)$ Now, Eq. 2 can be rewritten as:

$$\begin{aligned}
\eta_{\text{DM}} &= \sum_{i=1}^{m} \frac{1}{D_i} \left[p_1^2(i) + \sum_{j=2}^{k} p_j^2(i) \right] \\
&= \sum_{i=1}^{m} \frac{1}{D_i} \left[p_1(i) \left(D_i - \sum_{j=2}^{k} p_j(i) \right) + \sum_{j=2}^{k} p_j^2(i) \right] \\
&= \sum_{i=1}^{m} p_1(i) - \sum_{i=1}^{m} \frac{1}{D_i} \sum_{j=2}^{k} p_1(i) p_j(i) - p_j^2(i) \\
&= \sum_{i=1}^{m} p_1(i) - \sum_{i=1}^{m} \frac{1}{D_i} \sum_{j=2}^{k} p_j(i) \big(p_1(i) - p_j(i) \big)
\end{aligned} \tag{4}$$

Since $p_1(i) \geq p_j(i)$, $1 < j \leq k$, in Eq. 4 each expression $p_j(i)\big(p_1(i) - p_j(i)\big)$ is always positive. Hence:

$$\eta_{\text{DM}} \leq \sum_{i=1}^{m} p_1(i) \tag{5}$$

The expression $\sum_{i=1}^{m} p_1(i)$ in Eq. 5 attains its maximum value when the objects $O_1(1), O_1(2), \ldots, O_1(m)$, with probabilities $p_1(1), p_1(2), \ldots, p_1(m)$, respectively, are any permutation of the objects in the set $\{O_1, O_2, \ldots, O_m\}$. Note that $O_1, O_2, \ldots, O_m$ are the objects with the highest access probabilities among all the n objects. Therefore, the highest

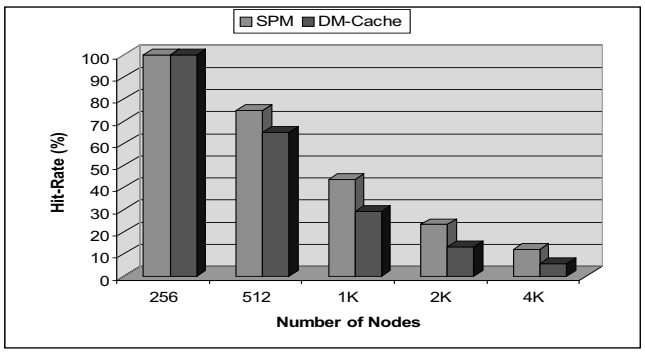

Fig. 3. Hit-rates for key search on the linked list. Cache performance worsens, compared to SPM, with increasing problem size.

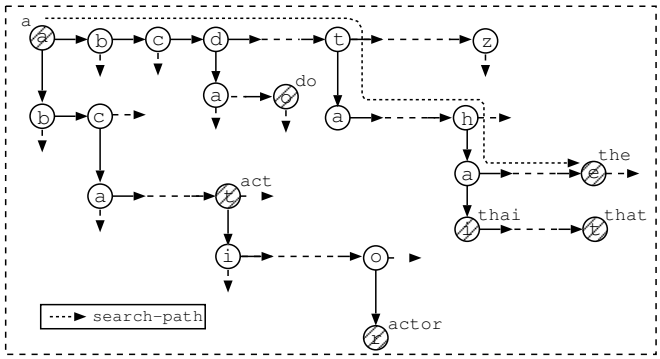

Fig. 4. Spell-checker implemented with trie data-structure. Search-path to *the* is delineated with dotted line.

value attained by $\eta_{\rm DM}$ is $\sum_{i=1}^{m} p_i$. Comparing this with Eq. 3, we conclude that $\eta_{\rm DM} \leq \eta_{\rm SPM}$.

Therefore, under the given assumptions, we see that the SPM with an optimal mapping can always outperform the DM-Cache. The cache can have any data-layout, whatsoever, and yet the SPM, with optimal mapping, will still perform better. In the next section, we will validate this conclusion with experiments. We will also show that our conclusion holds even for set associative caches.

In this paper, we have looked at dynamic data structures whose topology does not change or changes gradually over time. For example, our result holds well for a tree on which the basic operation is traversal. The tree's topology may change only very slowly over time through re-balancing, deletion and insertions. Another class of problems is when the topology changes very fast. In that case, the comparison between SPM and cache has to take into consideration the exact replacement policy of SPM. We are currently studying this class of applications but do not discuss the solution any further in this paper.

VI. EMPIRICAL VALIDATION

Let us now verify the theory with experiments. We will look at three applications and see how they fare on SPM and cache.

A. Linked List

Fig. 3 plots the measured hit-rates for searches conducted on the linked list of Fig. 1. In the experiment, the L1-memory size is 4KB, with cache block-size 16B. Each node is 16 bytes. In the case of SPM, the first m (256) nodes from the head of the list were placed on SPM. For the DM-Cache case, the first m nodes were placed in different cache blocks. From Fig. 3 we see that SPM does indeed outperform the direct-mapped cache. The cache performance, with respect to SPM, worsens for increasing problem size because of increasing conflicts.

B. Spell-Checker

The *spell-checker* [14] checks and reports whether the given word exists in its dictionary. To enable fast searches, the dictionary is built as a *trie* (tree with variable number of child-nodes). A trie is shown in Fig. 4. The search-path to the word *the* is delineated with a dotted line in the figure.

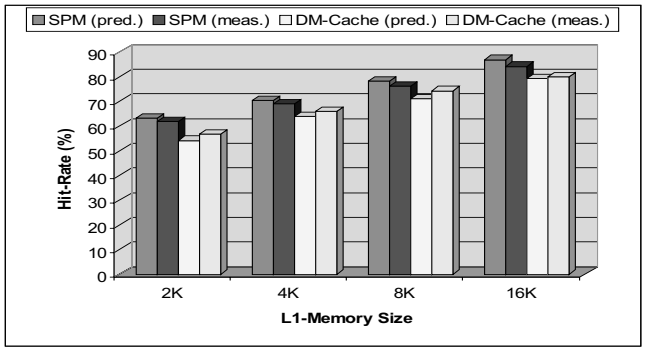

Fig. 5. Spell-Checker: Predicted (pred.) and measured (meas.) hit-rates. Predictions are within 2.6% accuracy.

At first, one might be tempted to map the spell-checker onto the cache because it involves data-dependent traversal of a pointer-based dynamic data structure. However, we will see that with a smart mapping the spell-checker actually does better on SPM. In our experiment, the trie contains over five thousand commonly used words. By performing a *mock* spell-check on a *training-essay*, the access probability of each trie-node is estimated. The access probabilities are then used to predict the hit-rates using Eq. 2 and 3. For SPM, it assumes that the nodes with highest access probabilities are mapped to SPM, while for the DM-Cache case it assumes that they map to different cache-blocks. The actual hit-rates are measured by running the spell-checker over another document of more than hundred thousand words.

Fig. 5 shows the predicted and measured hit-rates. The first observation is that the predictions, both for SPM and DM-Cache, is close to the measured values. Therefore, IRM is indeed able to model SPM and cache behavior for data-dependent traversals to a high degree of accuracy. The second observation is that, as proven in previous section, the SPM with optimal mapping does indeed outperform the DM-Cache – albeit by a small margin. However, since SPM is more energy efficient than cache, SPM is better suited for this application.

Let us now compare the previous SPM and DM-Cache mappings with other mapping techniques. Typically, the trie is *grown* by inserting new words into it. In the first version

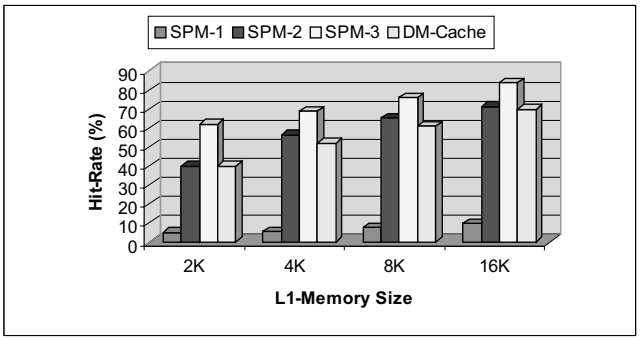

Fig. 6. Spell-Checker: Hit-rate comparison of SPM with DM-Cache. Using access-probabilities (SPM-3) gives better results over conventional mapping schemes (SPM-1 & 2).

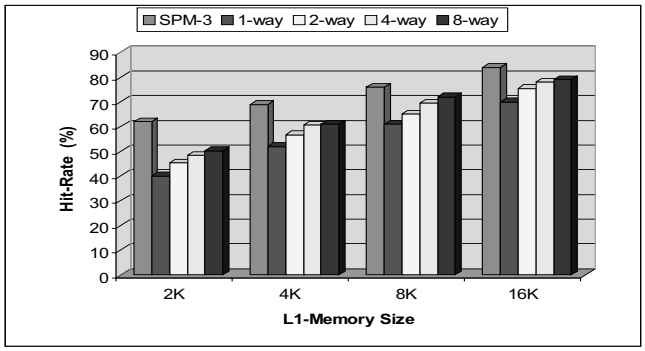

Fig. 7. Spell-Checker: SPM performance compared to 1, 2, 4 and 8 way set-associative caches. Increasing associativity does not improve performance significantly.

SPM-1, the words are inserted into the trie in lexicographical order and the nodes are allocated space in the SPM on a first come first serve basis, till the SPM gets full. In the SPM-2 version, the trie is built by inserting the most commonly used words (e.g. *the*, *as*, *and*) first, and then inserting the remaining words. Therefore, words which are searched most often have their paths *almost* entirely on the SPM. We say *almost* because words inserted later-on could potentially add new nodes in the paths to the commonly used words. Fig. 6 shows the hit-rates for SPM-1 and SPM-2, and compares them with SPM-3 and DM-Cache. The SPM-3 version puts the nodes with highest access probabilities on SPM. Therefore, SPM-3 is identical to SPM (meas.) in Fig. 5 but is shown again for convenience. DM-Cache in Fig. 6 shows the hit-rate when no customized mapping of nodes is done. This is different from the experiment conducted for DM-Cache (meas.) in Fig. 5 where nodes with highest access probabilities were mapped to different cache-blocks. Therefore, as expected, the hit-rate values for DM-Cache in Fig. 6 are less than those of DM-Cache in Fig. 5. From Fig. 6, we therefore conclude that mapping using access probabilities can be superior compared to conventional mapping techniques.

Next, we study the impact of increasing associativity on the performance of the cache. In particular, we would like to see if set-associative caches can outperform the SPM.

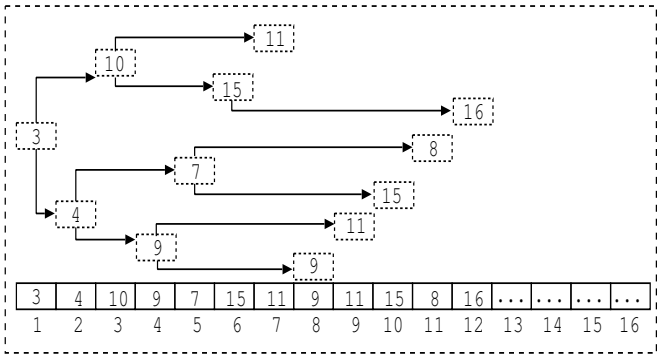

Fig. 8. A binary-heap implementation of the priority-queue for the minimum spanning tree algorithm.

Fig. 7 compares direct mapped (1-way) with 2, 4 and 8-way set-associative cache. The SPM hit-rate (columns for SPM-3) are also shown for comparison. We see that increasing the associativity does not tilt the performance toward cache. Moreover, set-associative caches are power hungry and hence any performance gains comes at high energy penalty.

C. MST - Prim's Algorithm

Prim's algorithm finds a minimum spanning tree (MST) for the given connected graph [6]. The algorithm starts with a single node and adds one edge at a time, till all the nodes have been connected. At any time, during the building of the tree, there is a set of nodes T already in the tree, and another set of nodes T' currently not in the tree. Each node in T' is assigned a weight that is equal to the cheapest edge connecting it to some node in T. Organizing the set of weights in T' such that the cheapest edge can be found quickly is done using a binary-heap. A binary-heap, such as in Fig. 8, is a complete binary-tree embedded into an array. Each node in the heap has a weight which is less than or equal to that of its two children. Therefore, the node h with the least weight is at the top (root). When h is removed, the last heap-node is moved to the top and then it *percolates* down to its new appropriate place.

In estimating the hit-rates (using Eq. 2 and 3), we assume that a node percolating downward, ultimately reaches the bottom. This simplification has a certain degree of associated error. Secondly, note that the tree shrinks in size as nodes are removed from it. Therefore, the estimated hit-rate is the mean for the entire range. Fig. 9 shows the predicted and measured hit-rates on a complete graph of over thousand nodes. We observe that the predicted values are slightly below the measured values. The reason is that since some nodes do not percolate all the way down, the access probabilities are actually higher for nodes near the top of the heap as compared to what was assumed in our calculations.

Fig. 10 gives the hit-rates for different set associative caches. We see that increasing the associative does not have a big impact on the cache performance and the SPM still outperforms the cache.

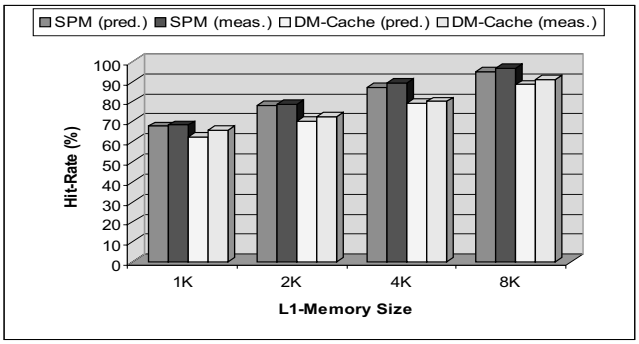

Fig. 9. Minimum Spanning Tree: Predicted (pred.) and measured (meas.) hit-rates of SPM and DM-Cache.

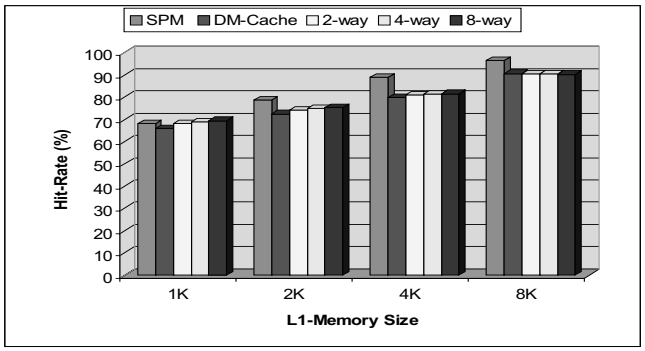

Fig. 10. Minimum Spanning Tree: SPM performance compared to 1, 2, 4 and 8 way set-associative caches. Increasing associativity does not improve performance significantly.

VII. CONCLUSION AND FUTURE WORK

Data-dependent access to data structures such as tries, trees, heaps and linked lists can be modeled to a reasonable level of accuracy in Independent Reference Model (IRM). We use IRM to prove that scratch-pad memories, with an optimal mapping based on access probabilities, can outperform the direct-mapped cache, irrespective of the layout influencing the cache behavior. This analytical result is then verified with experiments. Increasing the associativity in the cache is shown not to improve the cache performance in any significant way. We, therefore, see our main contribution as demonstrating, theoretically and empirically, that scratch-pad memories can be effectively used more than just for regular applications.

This paper does not address the issue of SPM and cache behavior when the topology of the data structure changes rapidly with time – resulting from insertions, deletions and restructuring of the nodes. In that case, scratch-pad and cache performance evaluation has to take into consideration the replacement policy of the scratch-pad, and the conflicts in the cache from different placement (layout) of objects in memory. Scratch-pad replacement policy can be application dependent or be independent (using generic allocators).

REFERENCES

[1] Mohammed Javed Absar and Francky Catthoor. Compiler-based approach for exploiting scratch-pad in presence of irregular array access. In *Design Automation and Test in Europe (DATE)*, pages 1162–1167, March 2005.

[2] A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, 1989.

[3] O. Avissar and R. Barua. An optimal memory allocation scheme for scratch-pad based embedded systems. *ACM Transactions on Embedded Computing Systems*, pages 6–26, November 2002.

[4] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, New York, NY, USA, 2002. ACM Press.

[5] F. Catthoor, F. Balasa, E. D. Greef, and L. Nachtergaele. *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publisher, 1998.

[6] T. Cormen, C. E. Leicerson, and R. Rivest. *Introduction to Algorithms*. Prentice Hall, 1998.

[7] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing, Cambridge Publishing*, 2005.

[8] J. D. Fix. Cache performance analysis of algorithms. *PhD Dissertation, University of Washington*, pages 22–36, 2002.

[9] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt. Data reuse analysis technique for software-controlled memory hierarchies. In *Design Automation and Test in Europe (DATE)*, pages 202–207, March 2004.

[10] P. R. Jelenkovic and A. Radovanovic. Least-recently-used caching with dependent requests. *Theoretical Computer Science*, 326(1-3):293–327, 2004.

[11] M. T. Kandemir, I. Kadayif, and U. Sezer. Exploiting scratch-pad memory using preseburger formulas. *International Symposium on System Synthesis (ISSS)*, 7(12), 2001.

[12] M. T. Kandemir and J. Ramanujan. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, 23(2):243–259, March 2004.

[13] W. F. King. Analysis of paging algorithm. *Proceedings of IFIP Congress*, pages 485–490, August 1971.

[14] Karen Kukich. Technique for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, 1992.

[15] Richard E. Ladner, James D. Fix, and Anthony LaMarca. Cache performance analysis of traversals and random accesses. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 613–622, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[16] Peter Marwedel. *Embedded System Design*. Kluwer Academic Publishers (Springer), Norwell, MA, USA, 2003.

[17] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *EDTC '97: Proceedings of the 1997 European conference on Design and Test*, page 7, Washington, DC, USA, 1997. IEEE Computer Society.

[18] Preeti Ranjan Panda, Alexandru Nicolau, and Nikil Dutt. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[19] Gururaj S. Rao. Performance analysis of cache memories. *J. ACM*, 25(3):378–395, 1978.

[20] Jaswinder Pal Singh, Harold S. Stone, and Dominique F. Thibaut. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Transactions on Computers*, 41(7):811–825, 1992.

[21] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. *Design Automation and Test in Europe (DATE)*, pages 409–414, March 2002.

[22] Kishore S Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons, New York, USA, 2002.

[23] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 104–109, New York, NY, USA, 2004. ACM Press.