An Integrated Scratch-Pad Allocator for Affine and Non-affine Code

Sumesh Udayakumaran and Rajeev Barua,
University of Maryland, College Park,
skumaran@glue.umd.edu, barua@eng.umd.edu

Abstract

Scratch-Pad memory (SPM) allocators that exploit the presence of affine references to arrays are important for scientific benchmarks. On the other hand, such allocators have so far been limited in their general applicability. In this paper we propose an integrated scheme that for the first time combines the specialized solution for affine program allocation with a general framework for other code. We find that our integrated framework does as well or outperforms other allocators for a variety of SPM sizes.

1. Introduction

The prevalence of SPM in embedded systems is evident from the large variety of chips with SPM available today in the market.(*e.g.,* [1, 4]). Such memory is typically software managed. Software management, apart from avoiding extra caching hardware, also allows for more predictable program execution than when caches are present. Studies have further shown other benefits of using SPM including reductions in power consumption, memory latency, area and even significant runtime gains for embedded applications. Trends in embedded designs indicate that this dominance of SPM's will continue in the future [9].

Compared to caches, the biggest drawback of embedded systems their inability to dynamically adapt to changing working sets. Towards bridging this gap, recent research has focussed on developing compiler-guided dynamic allocation strategies [6, 7, 10]. Such research holds out the potential of making SPM's more competitive with the performance of caches. Unfortunately, current solutions have either been optimized for programs that contain tightly nested loops with affine references [1] such as media applications or have been targeted at general programs. Although, the wide variety of applications on embedded platforms calls for general solutions, the presence of large amount of media/signal processing applications means optimizations for such applications cannot be overlooked either. A naive solution is that the designer having access to both the strategies, uses heuristics to choose the one that works best for his application. However, such an approach is not only expensive but fraught with other issues like longer design time. In contrast, our integrated scheme simplifies the work of the designer while never performing worse and occasionally performing better than the non-integrated approaches.

In this paper, we for the first time propose a tightly integrated framework that is optimized for loop dominated benchmarks while retaining the features of a general framework like handling non-affine references, arbitrary control flow and scalar variables. We have extended the general framework proposed by Udayakumaran et al [10] and added an affine analysis pass. The aim of the pass is to enable allocation of parts of an array, for instance when the whole array does not fit into the SPM. The pass essentially identifies the footprint of array references inside loop nests.[2] In doing this, it uses several novel features like flexible transfer points and scatter-gather copying for non-contiguous elements that further aid in a improved allocation. The pass results are then fed into the allocator that is now enhanced to use such analysis. The result is a general dynamic SPM allocation strategy that also exploits the presence of affine access patterns in the programs.

The rest of the paper is organized as follows. Section 2 overviews related work. Section 3 describes our integrated method for affine and non-affine code. Section 4 presents an evaluation of our methodology. Section 5 concludes.

2. Related work

Several researchers have looked at the issue of dynamic data/code allocation for SPM systems. Among those considering data allocation for SPM, some have focussed on allocation strategies for affine programs only [2, 5, 6, 8] while the rest have proposed more general solutions [7, 10]. So far as we are aware, none of the existing strategies provide an integrated solution.

1 An affine reference is a linear combination of the enclosing induction variables plus a constant, eg: A[2i+j] is affine while A[2*i*j] is not. For simplicity we term programs with only such references as affine programs

2 we will formally define what a footprint is later, but for now consider it as the set of elements accessed

Among the proposed strategies handling affine programs, there are two categories. The first category similar to ours, identifies the footprint of an affine reference. The second category of papers rely on data/code transformations to copy a part of the working set. Some aspects of these techniques such as precise footprint analysis are complementary to our method. One of the strengths of our integrated framework is that if so desired it can easily accommodate any of these optimizations.

The papers by Eisenbeis et al [5] and Schreiber et al [8] find the smallest footprint while Anantharaman et al [2] similar to us find a bounding rectangle around the footprint. Certainly, finding the smallest size may improve the utilization of the SPM. However, it would also make code generation for these references extremely difficult and hinder optimization of the address generation code.

These solutions also differ from ours in the granularity of the loop nest at which they work. The granularity is in terms of loop iterations [5] or the whole loop nest [2, 8]. The granularity of an iteration may mean not only extra overhead but also large transfer cost due to transfers inside the loop. On the other hand doing the transfers only outside loop nests may not provide any benefit of affine handling for loops that access the entire array. In contrast, our method although with conservative footprints, *generates all possible footprints for different transfer points in the loop nest*. The choice of the footprint and the corresponding transfer point is left to our cost-benefit model driven integrated allocator. The cost-model takes decisions in a more sophisticated fashion by taking note of the memory contents at that point. Finding footprints at different loop levels allows us to leverage this cost-model.

The other category of work is due to Kandemir et al [6]. Their solution involves tiling the loops to reduce working set and then copying it to the SPM. Such tiling is illegal for benchmarks with imperfectly nested loops, hand optimized code, arbitrary control structures and it also means that that only programs where there is no overlap between successive working sets can be targeted. So the solution flushes out the working set before the next tile can be brought in. The other part of their work determines one common layout for an array based on how it is accessed in different loop nests. We circumvent this issue by using copying code that collects non-contiguous elements and scatters them back into their original addresses when copying back.

3. Method Description

Our integrated method extends the framework proposed by Udayakumaran et al in [10] to now also handle affine programs. We first provide an overview of this framework. The method proposed in [10] provides a dynamic allocation method for global and stack variables. The method inserts code at different points which changes the contents of the SPM at those points.

More formally, the method follows the following steps. First, it partitions the program into regions where the start of each region is a *program point*. Second, the method associates a timestamp with every program point such that (i) the timestamps form a partial order; and (ii) the program points are reached during runtime roughly in timestamp order. Third, the method visits these different points in a timestamped order and determines memory transfers for each program point by using the cost-model. The cost-model takes into account different factors like the present contents, possible transfer points (eg. just outside the different loops of a loop nest), the transfer costs etc. Since this framework is used for both affine and non-affine data allocation by our integrated method, it can leverage its more comprehensive cost model to obtain additional benefits. Finally, the allocator assigns addresses to these variables in different regions. Code is then generated to access these variables using new temporaries. The end result is a modified code which can change the working set dynamically.

Our integrated method augments the framework in two parts. The first part involves an initial affine analysis pass to identify partial variables such as a row, a column or even a collection of elements belonging to an array variable that is accessed by a loop nest. These partial variables are then entered as *additional* variables for our framework to consider in addition to the original program variables for allocation into SPM. The other property of these additional variables is that they can be safely allocated without affecting the correctness of the program. The second part of the framework involves modifications to now accommodate these partial variables. The modifications include checks to avoid copies of the same array element in the the SPM and appropriate code generation for these partial variables. We now describe these two parts in more detail.

3.1. Affine analysis

The affine analysis phase finds partial variables that can be considered by the dynamic allocator. Potentially, for every reference and loop level there can be a partial variable. Before a potential partial variable can be considered by the allocator, two questions have to be answered. First, what are the elements in the variable? Two, is it safe to place the partial variable in the SPM? Two different partial variables can be intersecting. This then can introduce correctness issues because of multiple copies of an array element in the SPM. We call this issue as the *intersection problem*. We look into these issues in more detail.

Finding the elements in a partial variable The size of a partial variable is due to the set of elements accessed by a reference at a particular loop level. The footprint of a reference r in loop x, *footprint(r, x)*, is the set of data elements that can be accessed by the reference r in its enclosing loop

x. For simplicity of our analysis, we over estimate this set of elements as the entire set of elements along the direction of the stride. This extended set is called the spanned-footprint. So when the set of elements is a subset along a row/column, we estimate the extended set as the whole row/column. When the set is a subset of a plane, then the extended set is the whole plane itself. In this way we extend footprint(r,x) to spanned-footprint(r,x). The shape of the spanned-footprint is rectangular and so an address can be generated using base of the array + an offset.

To aid in a more precise identification of the spanned-footprint, we use the concept of data access matrix and data offset matrix. The data access function of each reference can be represented as F= Hz + K where H is the data access matrix and K is the data offset matrix. We represent the loop nest that we want to examine by vector $\vec{z}$= z_1,....z_k,..z_n where z_1 is the outermost loop and z_n is the innermost.

As a first step in identifying the spanned-footprint for loop z_k, our method reduces each data access function F of a reference to a reduced data access function using the following steps. First, H is reduced to H' by setting all elements in columns $k <= j <= n$ to 0. Then K is reduced to K' by setting the element of the rows that are all zero in H' to zero. The resulting expression $H'z + K'$ represented by F' is our desired reduced data access function at z_k. The reduced access function represents the array subspace affected only by the loop variables z_k,..z_n. This subspace identifies the spanned-footprint(r, z_k). To illustrate with an example, consider a reference A[i][2j][2k+10] in a loop nest {i, j, k} where i is the outermost and k is the innermost. The data access function F for this reference is

$$F = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} * \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 10 \end{pmatrix} = \begin{pmatrix} i \\ 2j \\ 2k+10 \end{pmatrix}$$

The reduced data function F' for loop j which identifies spanned-footprint(A[i][2j][2k+10], loop j) is

$$F' = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ 0 \\ 0 \end{pmatrix}$$

In general,

$$F' = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

represents element(s) A[x][y][z], except if any of x,y,z are zero, then it is replaced by '*' in the array expression,'*' representing the range of values for that array dimension.

Intersection analysis The second part of the problem is examining the intersection between different spanned-footprints. The intersection between spanned-footprints can happen in many different ways. The spanned-footprints can be exactly equal, partially intersecting or one can be the subset of the other. Finding the kind of intersection has two uses. One, such an analysis is important for the correctness of the program. If two spanned-footprints intersect and they are both allocated into the SPM at different addresses, then multiple inconsistent copies of array elements may exist in the SPM. Two, in the case of equal or subset/superset spanned-footprints, only one variable needs to be generated. This in turn eliminates redundant transfers.

To determine the kind of intersection between two references, we define three classes of intersection. In the first intersection class, a spanned-footprint of a reference is exactly equal to the other spanned-footprint. In this case we term them as being *common-spanned-footprint references* or in short *CSF references* and the intersection class as *common-spanned-footprint* class. The second class happens when the spanned-footprints are intersecting but not subset of each other. We term these references *intersecting but not subset* or in short *IBNS references*. We now examine how to first identify and then handle these classes.

The first class we define is when two spanned-footprints are exactly equal ie. for reference r1, r2 and loop variable x, spanned-footprint(r1, x) is equal to spanned-footprint(r2, x). *In terms of the mathematical framework that we have described earlier, two references at a particular loop level which have the same reduced data function have a common spanned-footprint.* For example references A[i][2j][2k] and A[i][j][2k] have the same spanned-footprint for only loop j, the reduced data access function for loop j being (i, 0, 0). These references from the same common-spanned-footprint class can be represented by one common partial variable.

The second class we define is due to intersecting but not subset references and we call it the *IBNS class*. Two references are called *IBNS references* if for any particular array dimension, the index expression for that dimension contains one loop variable in one reference but not in the other. For example for references A[i][j], A[i][i], dimension 2 contains j in one reference but not in the other. Such references cause partial intersection. To handle the spanned-footprints from IBNS references in the same way as common-spanned-footprint references needs identifying the precise intersecting elements and accessing them by using conditional code. To avoid such overhead, we instead adopt one of two approaches: in the first approach references are not considered at all for SPM allocation or alternatively we revert to our default solution of considering these references separately. The choice of approach is based on whether the IBNS references involve true dependence or not. This is since the problem of inconsistent copies arises only in case of true dependence between two references, when one copy is written to and thereafter the other copy is read from. For all other dependencies, the references can be handled separately without impacting correctness.

The third class called the *subset class* occurs when one spanned-footprint is a subset of the other. This class does

not impact correctness and is discussed in our modifications to the framework.

```
int A[10][10], B[10][10], C[10][10]
  for i =0 to 10
    for j=0 to 10
      C[i][j]=A[i][j]+A[i][j+1]+A[i+1][j-1]+A[i+1][j+1]
      B[j][j]=B[i][j] + 10
      C[j][j]=10
    endfor
  endfor
```

Partial Variables	Spanned Footprint of	Description	Induced by CSF ref.	Size words
A_1	loop j	Row i in A	A[i][j] A[i][j+1]	10
A_2	loop j	Row i +1 in A	A[i+1][j-1] A[i+1][j+1]	10
C_1	loop j	Diagonal in C	C[j][j]	10
C_2	loop j	Row i in C	C[i][j]	10

Figure 2. Example loop and output of affine analysis phase

Example To illustrate these ideas lets consider the loop in figure 2. The loop has one inner-loop and eight references or a possible set of eight partial variables. The output of the analysis is also given in the figure 2. An example partial variable is A_1 representing an entire row A[i], of size 10 and induced by references (A[i][j], A[i][j+1]). Formally, the first column gives the 4 different partial variables prefixed by their parent variable name. The fourth column gives the references which cause the partial variable to be generated. The last column gives the size of the spanned-footprint. Notice that references to variable B are not considered for SPM allocation since they intersect partially and involve true dependence between them. On other hand, references to variable C can still be considered as there is no true dependence between them.

The complete algorithm is shown as pseudocode in figure 3. The algorithm defines two functions *find-partial-set* that does the affine analysis we described before and *Compute-sizeof-CSF* that finds the size of a common-spanned-footprint class. *Find-partial-set* examines unconsidered references in a nested loop in three parts. In the first part (line 4-7) it determines the *CSF-class* of r at every loop level in the loop nest. Additionally, it also finds *Super-CSF-class*, the union of all the individual CSF-classes of r. In the second part (line 8), it finds the narrowed *IBNS-set*, the set of references that intersect partially but involve true dependence with the references in Super-CSF-class. These references are not handled any further. Finally in lines 10-13, for each CSF-class containing the reference, it generates a new partial variable. In line 13, the algorithm finds the size of the CSF class using Compute-sizeof-CSF-class function.

The Compute-sizeof-CSF function returns the size based on how the particular loop level affects the reference. It first finds the size at the current loop level (lines 16-23). If the current loop level does not affect the reference (lines 17-18) then the size at the level is 1. Otherwise (lines 18-22) if the iteration progress along the row or column or diagonal, then the size of the particular dimension is returned. Lines 23 uses the idea that the sizes in different loops inside a nested structure form a hierarchy. Hence the size of a spanned-footprint in a outer loop is product of the size at the current level and size of the spanned-footprint from one level lower.

3.2. Framework modifications

We now describe the second part of our integrated framework – the modifications to the framework described in [10]. One of the key aspects of integrated framework is that for most part of the algorithm, the additional variables generated can be treated like other variables. This allows us to retain the biggest advantage of non-affine frameworks which is their general applicability. We now look at two aspects of the original framework which have to be modified.

The first modification required is due to the hierarchy of partial variables originating from different loop levels. One partial variable for a particular reference is a subset of another partial variable generated for the same reference but at a higher loop level. The first case that needs to be handled differently is when a partial variable is being considered for swap in, and a variable that includes it is already in the SPM then the swap in can be ignored. On the other hand, if after the swap-in has been ignored, the superset variable itself gets swapped out, then the previous decision to ignore the swap-in has to be reversed and the partial variable swapped in. The opposite case that can happen is if a superset variable is to be brought in, and a subset variable already exists, then the subset variable needs to be evicted out. One another issue is that the cost-model needs to consider that the overhead of invoking the copying call is lesser in the case of the superset variable.

The second aspect of the original framework that needs different handling is generating the copying code. The copying code sometimes needs to collect non-contiguous elements. This is done by adding an additional parameter to the copy procedure. The parameter describes the stride between the elements. For contiguous elements this stride is trivially one. The other part of code generation - accessing the footprint in SPM, is done as in the original framework; that is, by using a temporary variable. In most cases this temporary can be referenced using base of array + offset kind of formula. In cases when that is not possible we borrow the re-indexing technique used in [8]. Apart from these changes, the rest of the framework including the allocation skeleton and address assignment remains the same.

```
Define Set Partial-set /* Set of partial variables */
Set    find-partial-set()
1. for each outer loop o_loop with i_loop as the innermost loop do
2.     Unconsidered-references ← all affine references 'r' inside o_loop
3.     while there are references in Unconsidered-references do
4.         Super-CSF-class(r) ← r
5.         for each loop x_loop from i_loop -1 to o_loop do
6.             Find set of references CSF-class(r, x_loop) to which r belongs
7.             Super-CSF-class(r) = Super-CSF-class(r) ⋃ CSF-class(r, x_loop)
8.         IBNS-set(r) ← References that in o_loop that partially intersect and are true
           dependent with any reference in Super-CSF-class(r)
9.         if IBNS-set == NULL_SET
10.            for each loop x_loop from i_loop -1 to o_loop do
11.                Generate new variable V(r, x_loop) to represent CSF-class(r, x_loop)
12.                Partial_set = Partial_set ⋃ V(r, x_loop)
13.                Size(r,x_loop) = Compute-sizeof-CSF(r, x_loop)
14.        Unconsidered-references = Unconsidered-references - Super-CSF-class
15. return (Partial-set)

void  Compute-sizeof-CSF(r, x_loop)
16.Set index_set = Index expressions with loop variable
       x_loop present in them
17.if size of index_set == 1
18.    size_at_current_loop_level = size of dimension
19.else if size of index_set == 0
20.    size_at_current_loop_level = 1
21.else /* direction of iteration not parallel to dimension */
22.    size_at_current_loop_level=size of dimension
23.size=size_at_current_loop_level*size(r, x_loop -1) /* size (r,x) is computed in line 13*/
24. return (size)
```

Figure 3. Pseudo-code for determining partial variables using affine analysis

4. Results

Experimental setup This section presents results comparing our integrated method against several other allocation alternatives. These include the static allocation described in [3] and the dynamic method for non-affine programs [10]. We also consider three variations of an affine-only method that only handles affine loops, flushing the SPM after each loop. The objective is to approximately illustrate how some of the other methods cited earlier can benefit from being a part of a integrated framework. The three variations named "Inflexible-Affine1", "Inflexible-Affine2" with respect to the affine reference perform the transfer before the loop, outside the whole loop nest respectively. Both of these variations employ a simple knapsack allocation that is based on a list of spanned-footprints sorted by their frequency per byte. The third affine-only method, called the Flexible-Affine method, allows transfers at any of loop levels. The final choice of the transfer point is left to the framework since it uses a more sophisticated cost model. The Flexible-Affine method differs from our integrated method in one it empties the SPM after each loop nest and two, the integrated method can also handle non-affine loops.

The applications and experimental setup traits are as follows. Our experiments study three benchmarks from MiBench: Lpc, Gsm, G721 (fullname G721.Wendyfung), one from UTDSP suite: Compress, one from Perfect Club: Wss and one from Spec95: Tomcatv. In our experiments we simulate an 1 cycle SPM and a 20 cycle external DRAM. Our integrated allocation method is implemented in the gcc 4.0 compiler for a MCORE processor.

Figure 4 shows the runtime at some key SPM sizes using 6 different allocation methods. Sizes are chosen to show overall trends. Typically the sizes include the first size when the runtime of the integrated dynamic methods changes significantly (by more than 1%), the final size beyond which the runtime of the integrated method is very close to static allocation and sizes in between this range where any of the dynamic methods perform significantly differently than the other methods. Outside of this SPM-size range dynamic methods generally perform no better than static methods because for smaller sizes no variable fit, while for larger sizes all variables fit in the SPM with no need for dynamic swapping. The figure also shows two important observations in favor of our integrated method.

The first observation is that the integrated method always does as good or better than all the other other methods. In particular, the method does as well or better than both affine and non-affine method. This can be seen in affine benchmarks Tomcatv, Compress and Wss where the integrated method does as well as the Flexible-Affine method while for benchmarks with no affine loops G721, Gsm and Lpc, the integrated and non-affine methods do better.

It is instructive to see why different benchmarks have different best-allocation strategies. In the case of benchmarks with affine loops, the affine and the integrated method can allocate partial variables to SPM whereas neither the static nor the non-affine method can do that. On the other for the same benchmarks but larger SPM sizes like for example Tomcatv at size 85,000, the integrated method does better than the affine methods. This happens due to the simple cost model of the affine methods and higher transfer cost arising from the flushing of the SPM at the end of the loop nest. The other three non-affine benchmarks namely Gsm, G721 and Lpc do not offer any opportunities for partial variable allocation. Hence the integrated and the non-affine method do equally well and better than the affine alternatives. In conclusion, the best-allocation strategy is a function of both the benchmark characteristic and SPM size, with the integrated method uniformly being the best.

The second salient observation is that even among the affine alternatives, flexible transfers makes a difference. This is supported by the observation that flexible-affine either outperforms or equals the other affine alternatives. For two benchmarks Wss and Compress, the flexible-Affine

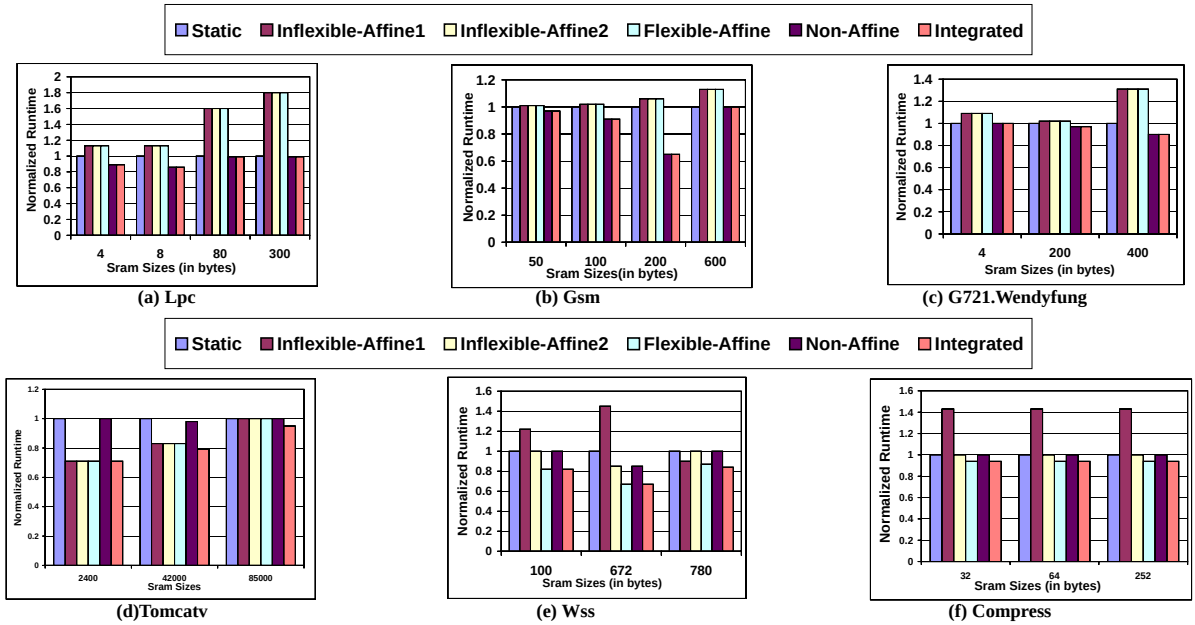

Figure 4. Normalized runtime for our integrated method, affine only method, non-affine method and static method .

does better than the other affine alternatives. These benchmarks contain triply nested loops and three points where transfer can be done. Transfers outside the outer loop mean that the whole variable has to be placed while transfer just outside the innermost loop causes the transfer cost to be high. The flexible-Affine method on the other hand compares between all the three transfer points and chooses the most beneficial one, which in this case is the middle loop. In case of Tomcatv that contains only doubly nested loops, the runtime of Flexible-Affine is the same as one of the other affine methods.

5. Conclusion

In this paper we present an integrated algorithm to allocate both affine and non affine programs into SPM. Our integrated algorithm is an extension of a previously proposed general framework [10]. Besides the advantages of an non-affine algorithm like general applicability, the algorithm can also places parts of variables when they accessed using affine functions. Integration also allows for selecting the transfer point using a cost-model. Our results show that the integrated algorithm combines the benefits of an affine and non affine allocator. For a wide range of sizes, the integrated algorithm does as well or better than either affine only or non-affine only allocators.

References

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

[2] S. Anantharaman and S. Pande. Compiler optimization for real time execution of loops on limited memory embedded systems. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, 1998.

[3] O. Avissar, R. Barua, and D. Stewart. An Optimal Memory Allocation Scheme for Scratch-Pad Based Embedded Systems. *ACM Transactions on Embedded Systems (TECS)*, 1(1), September 2002.

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

[5] D. C. Eisenbeis, W. Jalby and C. Fran. A strategy for array management in local memory. In *Technical Report 1262, INRIA, Domaine de Voluceau, France*, 1990.

[6] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic Management of Scratch-Pad Memory Space. In *Design Automation Conference*, 2001.

[7] L. W. M. Verma and P. Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *Intnl. conference on Hardware/Software Codesign and System Synthesis(CODES+ISIS)*. ACM, 2004.

[8] D. C. R. Schreiber. Near-optimal allocation of local memory arrays. In *HPL-2004-24*, 2004.

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

[10] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proc. of the int'l. conference on Compilers, architectures and synthesis for embedded systems (CASES)*, pages 276–286. ACM Press, 2003.