SRAM vs. Data Cache: The Memory Data Partitioning Problem in Embedded Systems

Preeti Ranjan Panda
Nikil Dutt
Alexandru Nicolau

Technical Report #96-42
September 1996

University of California, Irvine
Irvine, CA 92697-3425
(714) 824-8059

Abstract

Efficient utilization of on-chip memory space is extremely important in modern embedded system applications based on microprocessor cores. In addition to a data cache that interfaces with slower off-chip memory, a fast on-chip SRAM, called Scratch-Pad memory, is often used in several applications. We present a technique for efficiently exploiting on-chip Scratch-Pad memory by partitioning the application's scalar and arrayed variables into off-chip DRAM and on-chip Scratch-Pad SRAM, with the goal of minimizing the total execution time of embedded applications. Our experiments on code kernels from typical applications show that our technique results in significant performance improvements.

Contents

List of Figures

1 Introduction

Complex embedded system applications typically use heterogeneous chips consisting of microprocessor cores, along with on-chip memory and co-processors. Flexibility and short design time considerations drive the use of CPU cores as instantiable modules in system designs [6]. The integration of processor cores and memory in the same chip effects a reduction in the chip count, leading to cost-effective solutions. Examples of commercial microprocessor cores commonly used in system design are LSI Logic's CW33000 series [3] and the ARM series from Advanced RISC Machines [13]. Typical examples of optional modules integrated with the processor on the same chip are: Instruction Cache, Data Cache, and on-chip SRAM. The instruction and data cache are fast local memory serving an interface between the processor and the off-chip memory. The on-chip SRAM, termed **Scratch-Pad memory**, is a small, high-speed data memory that is mapped into an address space disjoint from the off-chip memory, but connected to the same address and data buses. Both the cache and Scratch-Pad SRAM have a single processor cycle access latency, whereas an access to the off-chip memory (usually DRAM) takes several (typically 10–20) processor cycles. The main difference between the Scratch-Pad SRAM and data cache is that, the SRAM guarantees a single-cycle access time, whereas an access to cache is subject to *compulsory, capacity,* and *conflict misses* [9].

When an embedded application is compiled, the accessed data can now be stored either in the Scratch-Pad memory or in off-chip memory. In the second case, it is accessed by the processor through the cache. We present a technique for minimizing the total execution time of an embedded application by a careful partitioning of scalar and array variables used in the application into off-chip DRAM (accessed through data cache) and Scratch-Pad SRAM.

2 Related Work

The problem of partitioning data into different memory banks to facilitate parallel access has been addressed before. A technique to partition variables for simultaneous access into two memory banks of the Motorola 56000 DSP processor was reported in [12]. However, parallel access is not the objective of our partitioning problem we are addressing – we wish to maximize performance in the sequential access scenario.

Optimization techniques for improving the data cache performance of programs have been reported [4, 8, 11]. The analysis in [11] is limited to scalars, and hence, not generally applicable. Iteration space *blocking* for improving data locality is studied in [4]. This technique is also limited to the type of code that yields naturally to blocking. In [8], a data layout strategy for avoiding cache conflicts is presented. However, in many cases, the array access patterns are too complex to be statically analyzable using this

method. The availability of an on-chip SRAM with guaranteed fast access time creates an opportunity for overcoming some of the conflict problems (Section 3). The problem of partitioning data into SRAM and cache with the objective of maximizing performance, which we address in this paper, has, to our knowledge, not been attempted before.

3 Problem Description

Figure 1(a) shows the architectural block diagram of an application employing a typical embedded core processor [1], where the parts enclosed in the dotted rectangle are implemented in one chip, and which interfaces with an off-chip memory, usually realized with DRAM. The address and data buses from the *CPU core* connect to the *Data Cache, Scratch-Pad memory*, and the *External Memory Interface* (EMI) blocks. On a memory access request from the CPU, the data cache indicates a cache hit to the EMI block through the **C_HIT** signal. Similarly, if the SRAM interface circuitry in the Scratch-Pad memory determines that the referenced memory address maps into the on-chip SRAM, it assumes control of the data bus and indicates this status to the EMI through signal **S_HIT**. If both the cache and SRAM report misses, the EMI transfers a block of data of the appropriate size (equal to the *cache line size*) between the cache and the DRAM.

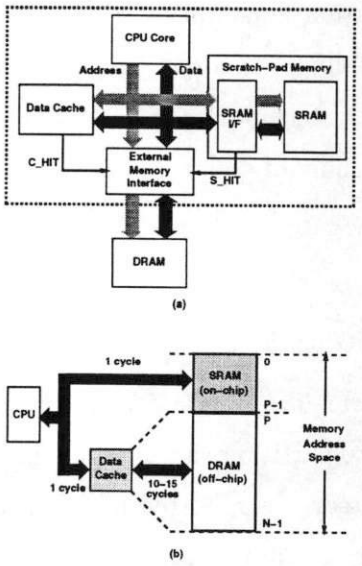


Figure 1: (a) Block Diagram of Typical Embedded Processor Configuration (b) Division of Data Address Space between SRAM and DRAM

The data address space mapping is shown in Figure 1(b), for a sample addressable memory of size N data words. Memory addresses $0 \ldots P-1$ map into the Scratch-Pad memory, and have a single processor cycle access time. Thus, in Figure 1(a), S_HIT would be asserted whenever the processor

[1] For example, the LSI Logic CW33000 RISC Microprocessor core[3]

attempts to access any address in the range $0 \ldots P-1$. Memory addresses $P \ldots N-1$ map into the off-chip DRAM, and are accessed by the CPU through the data cache. A cache hit for an address in the range $P \ldots N-1$ results in a single-cycle delay, whereas a cache miss, which leads to a block transfer between off-chip and cache memory, results in a delay of 10-20 processor cycles.

In this memory partitioning work, we assume that register allocation, the task that assigns frequently accessed program variables such as loop indices to processor registers, has already been performed. Given the embedded application code, (currently restricted to a single program sub-routine) the goal of our approach is to determine the mapping of each scalar and arrayed program variable into local Scratch-Pad SRAM or off-chip DRAM, while maximizing the application's performance.

The sizes of the data cache and the Scratch-Pad SRAM are limited by the total area available on-chip, as well as by the single cycle access time constraint. Hence, we must first justify the need for both the data cache and SRAM. Suppose the embedded core processor in Figure 1 can support a total of 2 KBytes for the data cache and the SRAM. We can analyze the pros and cons of four extreme configurations of the cache and SRAM are possible:

(1) No local memory: In this case, we have the CPU accessing off-chip memory directly, and spending 10–20 processor cycles on every access. Data locality is not exploited and the performance is clearly inferior in most cases.

(2) Scratch-Pad memory of size 2K: In this case, we have an on-chip SRAM of larger size, but no cache. The CPU has an interface both to the SRAM and the off-chip memory. First, the larger SRAM slows down the processor cycle speed [2]. Further, when large arrays that do not fit into the SRAM are present, the direct interface to external memory has to be used, thereby degrading performance.

(3) Data cache of size 2K: Here, we have a larger data cache, but no separate local SRAM. As observed in (2) above, this could also lead to a longer processor cycle time because of the longer cache access time. An advantage of the Scratch-Pad memory over data cache is that, for the variables mapped to the Scratch-Pad memory, we avoid the overhead of stalled CPU cycles during cache misses. Moreover, in many cases, having only a cache results in certain unavoidable cache misses that degrade performance. A simple example is the loop shown below:

```
loop i = 1 to N
        read A[i]      /* Array Variable */
        read x         /* Scalar Variable */
        write A[i]     /* Array Variable */
end loop
```

[2] Note that in Figure 1(a), the address decoding for cache and SRAM is done in parallel.

Only memory accesses in the loop are shown. If we assume that a direct-mapped cache of size C ($C < N$), with line size L is used, then there are L cache misses due to conflicts in every C accesses to array A, arising from a block of the array and variable x mapping to the same cache line.

(4) 1K Data cache + 1K Scratch-Pad SRAM: In the architecture of Figure 1, using a 1K data cache and 1K SRAM, the problem incurred in (3) above could be elegantly solved by assigning x to the SRAM, thereby avoiding all conflicts. Note that we assume the register allocation phase precedes the memory assignment phase. The simple problem above could be solved by assigning x to a register. However, in general, the types of conflicts noted above cannot all be solved by register allocation because registers are limited resources. Examples of such cases are loops involving too many scalar variables, or multiple conflicting arrays. The same argument of guaranteed single-cycle access and avoiding possible conflicts makes the architecture with a combination of cache and Scratch-Pad memory superior to a single associative cache in these circumstances.

From the above, it is clear that both the SRAM and data cache are desirable. Note that there could be applications where the Scratch-Pad SRAM offers no particular advantage over a single cache. Examples are: when no scalars are involved and all arrays are too big to fit into SRAM; or there is little temporal reuse among the arrays so that the cache conflicts do not cause any performance penalty. However, we noticed in our experiments that the SRAM improves performance in most typical applications.

In this work, we present a strategy for partitioning scalar and arrayed variables in an application code into Scratch-Pad memory and off-chip DRAM accessed through data cache, to maximize the performance by selectively mapping to the SRAM those variables that are estimated to cause the maximum number of conflicts in the data cache.

4 The Partitioning Strategy

The overall approach in partitioning program variables into Scratch-Pad memory and DRAM is to minimize the cross-interference between different variables in the data cache. We first outline the different features of the code affecting the partitioning, and then present a partitioning strategy based on these features.

4.1 Features Affecting Partitioning

The partitioning of variables is governed by the following factors:

- Scalars variables and constants
- Size of arrays
- Life-times of variables
- Access frequency of variables
- Conflicts in loops

We describe below each of the above factors and how our partitioning strategy address the features.

4.1.1 Scalar Variables and Constants

In order to prevent interference with arrays in the data cache, we map all scalar variables and constants to the Scratch-Pad memory. This assignment helps avoid the kind of conflicts mentioned in Section 3. If scalars are mapped to the DRAM, (and, consequently, accessed through the cache), it may be impossible to avoid cache conflicts with arrays, because arrays are assigned to contiguous blocks of memory, parts of which will map into the same cache line as the scalars, causing conflict misses.

It is possible to do a more sophisticated analysis of the most frequently accessed scalars to the SRAM, but our decision to map all scalars to the SRAM is based on our observation that for most applications, the memory space attributable to scalars is negligible compared to that occupied by arrays.

4.1.2 Size of Arrays

We map arrays that are larger than the SRAM into off-chip memory, so that these arrays are accessed through the data cache. Mapping large arrays to the cache is the natural choice, as it simplifies the array addressing. If a part of the array were to map into the SRAM, the compiler would have to generate book-keeping code that keeps track of which region of the array is addressed, thereby making the code inefficient. Further, since most loops access array elements more or less uniformly, there is little or no motivation to map different parts of the same array to memories with different characteristics.

4.1.3 Life-Times of Variables

The *life-time* of a variable, defined as the period between its *definition* and *last use* [1], is an important metric affecting register allocation. Variables with disjoint lifetimes can be stored in the same processor

register. The same analysis, when applied to arrays, allows different arrays to share the same memory space.

The life-time information can also be used to avoid possible conflicts among arrays. We perform the lifetime analysis on the topmost block of a function in the program. To simplify the conflict-analysis, we assume that a variable accessed inside a loop is *alive* throughout the loop. Similarly, we also assume that a variable that is alive in one branch of a conditional statement, is also alive in all other branches. We define a measure *Intersecting Life Times, ILT*(u), which indicates the number of program variables having a non-null intersection of life-times with u.

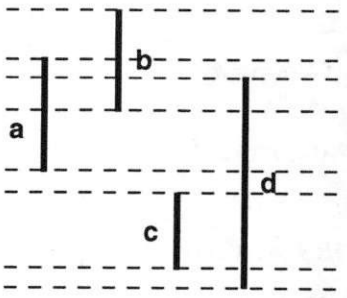


Figure 2: Example Life-Time Distribution

Figure 2 shows an example lifetime distribution of four variables, $a, b, c,$ and d. We have $ILT(a) = 2$, since its lifetime intersects that of 2 other variables, b and d. Similarly, we have $ILT(b) = 2$; $ILT(c) = 1$; and $ILT(d) = 3$. The *ILT* value of each variable gives an indication of the number of other arrays that it could possibly interfere with, in the cache. Consequently, we could map the arrays with the highest *ILT* values into the SRAM, thereby eliminating a large number of potential conflicts. We refine this measure in the next section.

4.1.4 Access Frequency of Variables

The *ILT* measure defined earlier gives an indication of the possibility of cache conflicts, but not the extent. To obtain a more accurate picture of the extent of conflicts, we have to consider the frequency of accesses. For example, in Figure 2, if the number of accesses to d is relatively small, it is worth considering the other arrays first for inclusion in SRAM, because d does not play a significant part in cache conflicts in spite of the high *ILT* value. For each variable u, we define the *Variable Access Count, VAC(u)*, to be the number of accesses to elements of u during its lifetime.

Similarly, the number of accesses to *other* variables during the lifetime of a variable, is an equally important determinant of cache conflicts. The $ILT(u)$ figure, which gives the *number of arrays alive* during the lifetime of u, has to be suitably modified to account for the *number of accesses.* For each variable u, we define the *Interference Access Count, IAC*(u), to be the number of accesses to other

variables during the lifetime of u.

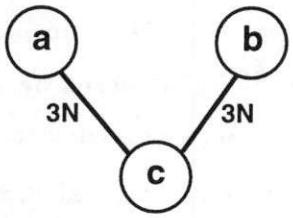
We note that each of the factors discussed above, $VAC(u)$ and $IAC(u)$, taken individually, could give a misleading idea about the possible conflicts involving variable u. Clearly, the conflicts are determined jointly by the two factors considered together. A good indicator of the conflicts involving array u is given by the product of the two metrics. We define the *Interference Factor, IF,* of a variable u as: $IF(u) = VAC(u) \times IAC(u)$. A high IF value for u indicates that u is likely to be involved in a large number of cache conflicts if mapped to DRAM. Hence, we choose to map variables with high IF values into the SRAM.

4.1.5 Conflicts in Loops

In the previous section, we assumed different arrays accessed in the same period (e.g., in the same loop) had an equal probability of conflicting in the cache. However, it is possible to make a finer distinction based on the array access patterns. Consider a section of a code in which three arrays a, b, and c are accessed, as shown in Figure 3(a).

```
for i = 0 to N-1
    access a [i]
    access b [i]
    access c [2 i]
    access c [2 i + 1]
end for
```

(a)



(b)

Figure 3: (a) Example Loop (b) Loop Conflict Graph

We notice that arrays a and b have an identical access pattern, which is different from that of c. Data placement techniques [8] can be used to avoid data cache conflicts between a and b. However, when the access patterns are different, cache conflicts are unavoidable (e.g., between a and c). In such circumstances, conflicts can be minimized by mapping one of the conflicting arrays to the SRAM. For instance, conflicts can be eliminated in the example above, by mapping a and b to the DRAM/cache, and c to the Scratch-Pad memory.

To accomplish this, we first build a *Loop Conflict Graph, LCG* with one node for each array, and edge weight $e(u, v)$ being computed as $e(u, v) = \sum_{i=1}^{p} k_i$, where the summation is over all loops $(1 \ldots p)$ in which u and v are both accessed, and k_i is the total number of accesses to u and v in loop i. In the example above, where we have only one loop $(p = 1)$, the graph in Figure 3(b) is generated. We have one access to a and two to c in one iteration of the loop. Total number of accesses to a and c combined is: $(1 + 2) \times N = 3N$. Thus, we have $e(a, c) = 3N$. Similarly, $e(b, c) = 3N$. We have $e(a, b) = 0$, since

the access patterns to a and b are *compatible* [3].

We now use the graph LCG to define the *Loop Conflict Factor, LCF* for a variable u as: $LCF(u) = \sum_{v \in LCG-\{u\}} e(u,v)$, i.e., $LCF(u)$ is the sum of incident edge weights to node u. This gives us a metric to compare the criticality of loop conflicts for all the arrays. In general, the higher the LCF number, the more conflicts are likely for an array, and hence, the more desirable it is to map the array to the Scratch-Pad memory.

4.2 The Partitioning Algorithm

We now outline an algorithm that determines the mapping decision of each (scalar and arrayed) program variable to SRAM or DRAM/cache by using the factors mentioned in Section 4.1. The input to algorithm *AssignMemoryType* (Figure 4) is the SRAM size and the application code, which, in this work, is assumed to consist of only a single function or procedure. The output is the assignment of each variable to Scratch-Pad memory or DRAM.

The algorithm first assigns the scalar constants and variables to the SRAM, and the arrays that are larger than the Scratch-Pad memory, to the DRAM. Procedure *AssignSRAM* updates a data structure that keeps track of the assignments of different arrays to the SRAM, as well as the free space currently available in the SRAM (tracked by variable *FreeSRAMSpace*). Similarly, procedure *AssignDRAM* performs the book-keeping related to assigning the array to DRAM.

For the remaining n arrays (all of which are small enough to fit into the SRAM), we first compute the LCF and IF values (Section 4.1), and generate the life-time intervals. We sort the $2n$ interval end-points thus generated, and traverse them in increasing order. Each point could represent either the beginning, or ending of a life-time interval.

If the end-point encountered (t_j) is a beginning of interval of array u, we have to make the mapping decision of u into SRAM or DRAM. We first check if there is sufficient space in the SRAM for u and the set ($S(u)$) of the other unmapped arrays intersecting its lifetime. If the space is sufficient, then we set flag C_s indicating that u can map to the SRAM, because there is no contention for SRAM space throughout u's life-time [4]. We then examine the criticality of the Loop Conflict Factor, $LCF(u)$. If there is sufficient SRAM space for u *and* all arrays with life-times intersecting $LT(u)$ (the life-time interval of u), with more critical LCF numbers (this is set $G(u)$), we consider u to be critical enough, in terms of LCF, and set flag C_g. We do a similar check for $IF(u)$ to set flag C_h. If any of C_s, C_g, or C_h is TRUE, we map u to the SRAM, otherwise, to the DRAM. Note that, in practice, the computation

[3] We call two expressions g and h compatible if $g - h = constant$, i.e., $g - h$ is independent of the loop indices.

[4] Note that this is a *greedy* algorithm. The memory assignment problem, as formulated here, can be easily seen to be NP-hard because it reduces to the register allocation problem when all the arrays are reduced to a single element.

Algorithm *AssignMemoryType*
Input: Application code with Register-allocated variables marked;
SRAM_Size: Size of Scratch-Pad SRAM
Output: Assignment of each program variable to SRAM or DRAM

```
    FreeSRAMSpace = SRAM_Size
    for all variables v
        if v is a scalar variable or constant
            AssignSRAM(v)
        else
            if size(v) > SRAM_Size
                AssignDRAM(v)
L1: for all arrays i not yet mapped (i = 1...n)
        Compute LCF(i) values
        Compute IF(i) values
        Generate life-time intervals LT(i) = (i1, i2), where i1 corresponds to
        its definition, and i2 corresponds to its last use
    Sort the 2n end-points of intervals (2 for each interval) in increasing order: t1...t2n
L2: for j = 1...2n (i.e., for each interval end-point in increasing order)
        if tj corresponds to the beginning of life-time interval of array u
            Set Conditions Cs, Cg, and Ch = FALSE
            Let S(u) = {s|(s is unmapped) and (LT(s) ∩ LT(u) ≠ φ)}
            -- S(u) is the set of all unmapped arrays whose life-times intersect LT(u)
            if size({u} ∪ S(u)) ≤ FreeSRAMSpace -- No contention with u for SRAM space
                Cs = TRUE
            Let G(u) = {s|(s is unmapped) and (LT(s) ∩ LT(u) ≠ φ) and (LCF(s) > LCF(u))}
            -- G is the set of unmapped arrays with life-time
            -- intersecting LT(u), with more critical LCF numbers
            if size({u} ∪ G(u)) ≤ FreeSRAMSpace
                Cg = TRUE
            Let H(u) = {s|(s is unmapped) and (LT(s) ∩ LT(u) ≠ φ) and (IF(s) > IF(u))}
            -- H is the set of unmapped arrays with life-time
            -- intersecting LT(u), with more critical IF numbers
            if size({u} ∪ H(u)) ≤ FreeSRAMSpace
                Cg = TRUE
            if (Cs = TRUE) or (Cg = TRUE) or (Ch = TRUE)
                AssignSRAM(u)
            else
                AssignDRAM(u)
            Remove u from list of unassigned arrays
        else -- tj corresponds to end of life-time interval of array u
            if u was mapped to SRAM
                UnAssignSRAM(u)
            else
                UnAssignDRAM(u)
end Algorithm
```

Figure 4: Algorithm for mapping variables into SRAM and DRAM

of all three sets $S(u), G(u)$, and $H(u)$ might not be necessary. For instance, if condition C_s is satisfied, we can directly assign u to SRAM, and need not compute $G(u)$ and $H(u)$. The algorithm, as outlined in Figure 4, is shown only for simplicity, and in practice, we evaluate C_s, C_g, and C_h, in that order, stopping computation when any of them becomes TRUE. Having made the memory assignment, we remove u from the list of unmapped arrays.

If the end-point identified by t_j corresponds to the end of a life-time interval of u, we invoke procedures *UnAssignSRAM* and *UnAssignDRAM* to de-allocate it from the data structure keeping track of array assignments to SRAM/DRAM.

To analyze the complexity of algorithm *AssignMemoryType*, we observe that the computation of *LCF* and *IF* values takes $O(n)$ time for each array, where n is the number of arrays. Generation of life-time intervals takes time linear in the code size of the application. Hence, loop $L1$ takes $O(n^2)$ time. Sorting the $2n$ interval end-points takes $O(n \log n)$ time. The computation of $S(u), G(u)$, and $H(u)$ takes $O(n)$ time. Since the code in loop $L2$ is executed $2n$ times, the algorithm has an overall complexity of $O(n^2)$.

5 Experiments

We performed simulation experiments on several benchmark examples that frequently occur as code kernels in several embedded applications [7], to evaluate the efficacy of our Scratch-Pad memory/DRAM data partitioning algorithm. We used an example Scratch-Pad SRAM and a *direct-mapped, write-back* data cache size of 1 KByte each. In order to demonstrate the soundness of our technique, we compared the performance (measured in total number of processor cycles required to access the data during execution of the example) of the following architecture and algorithm configurations: (**A**) *Data cache of size 2K*: in this case, there is no SRAM in the architecture; (**B**) *Scratch-Pad memory of size 2K*: in this case, there is no data cache in the architecture, and we use a simple algorithm that maps all scalars, and as many arrays as will fit into the SRAM, and the rest to the off-chip memory; (**C**) *Random Partitioning*: in this case, we used a 1K SRAM and 1K Data cache, and a random data partitioning technique[5]; and (**D**) *Our Technique*: here we used a 1K SRAM and 1K data cache, and algorithm *AssignMemoryType* for data partitioning. The size 2K was chosen in **A** and **B**, because the area occupied by the SRAM/cache would be roughly the same as that occupied by 1K SRAM + 1K cache, to a degree of approximation, ignoring the control circuitry. We use a direct-mapped data cache with line size = 4 words, and the following access times:

Time to access one word from Scratch-Pad SRAM = 1 cycle.

[5] Variables were considered in the order they appeared in the code, and mapped into SRAM if there was sufficient space.

Time to access one word from off-chip memory (when there is no cache) = 10 cycles.

Time to access a word from data cache, on cache hit = 1 cycle.

Time to access a block of memory from off-chip DRAM into cache =
10 cycles + 1 × Cache Line Size = 10 + 1 × 4 = 14 cycles. This is the time required to access the first word + time to access the remaining (contiguous) words [6].

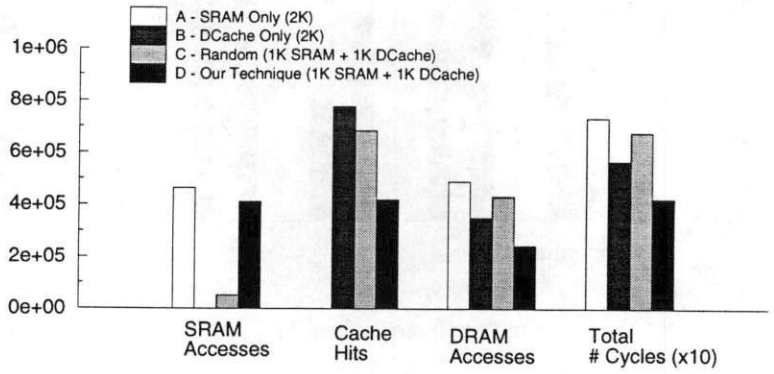


Figure 5: Memory Access Details for *Beamformer* Example

Figure 5 shows the details of the memory accesses for the *Beamformer* benchmark example. The *Beamformer*, a DSP application, represents an operation involving temporal alignment and summation of digitized signals from an N-element antenna array. We note that configuration **A** has the largest number of *SRAM Accesses*, because the large SRAM (2K) allows more variables to be mapped into the Scratch-Pad memory. Configuration **B** has zero SRAM accesses, since there is no SRAM in that configuration. Also, our technique (**D**) results in far more SRAM accesses than the random partitioning technique, because the random technique disregards the behavior when it assigns precious SRAM space. Similarly, *Cache Hits* are the highest for **B**, and zero for **A**. Our technique results in fewer cache hits than **C**, because many memory elements accessed through the cache in **C**, map into the SRAM in our technique. Configuration **A** has a high *DRAM Access* count because the absence of the cache causes every memory access not mapping into the SRAM, to result in an expensive DRAM access. As a result, we observe that the total number of processor cycles required to access all the data is highest for **A**. Configuration **D** results in the fastest access time, due to the judicious mapping of the most frequently accessed, and conflict-prone elements into Scratch-Pad memory [7].

Figure 6 presents a comparison of the performance for the four configurations **A**, **B**, **C**, and **D** mentioned earlier, on code kernels extracted from seven benchmark embedded applications. *Dequant* is the de-quantization routine in the MPEG decoder application[5]. *IDCT* is the Inverse Discrete Cosine Transform, also used in the MPEG decoder. *SOR* is the successive Over-Relaxation algorithm,

[6] This is a popular model for cache/off-chip memory traffic [9].

[7] Figure 5 shows the total number of cycles scaled down by a factor of 10

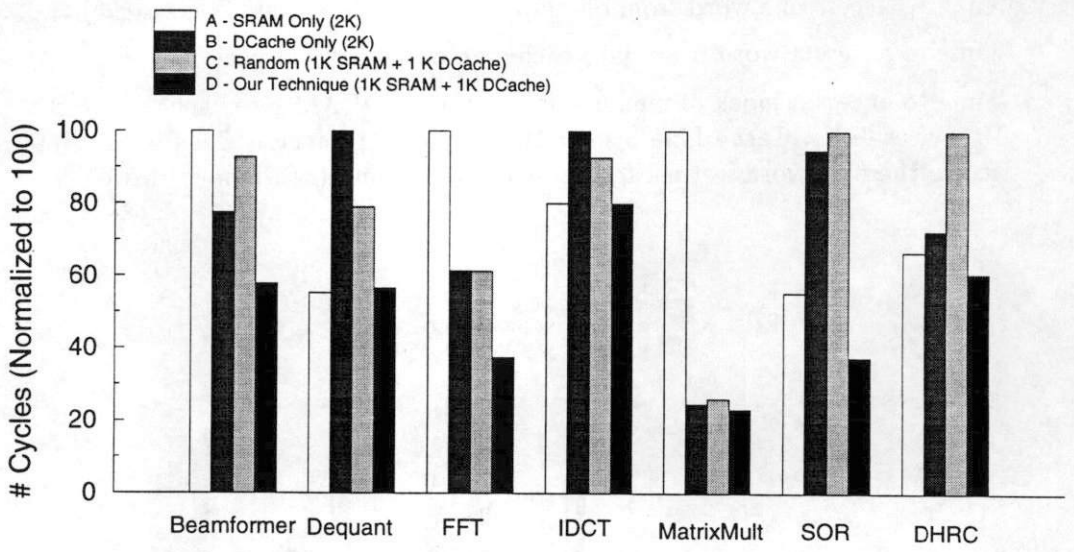


Figure 6: Performance Comparison of Configurations A, B, C, and D

frequently used in scientific computing [10]. *MatrixMult* is the matrix multiplication operation, optimized for maximizing spatial and temporal locality by reordering the loops. *FFT* is the Fast Fourier Transform application. *DHRC* encodes the Differential Heat Release Computation algorithm that models the heat release within a combustion engine [2].

The number of cycles for each application is normalized to 100 in Figure 6. In the *Dequant* example, **A** slightly outperforms **D**, because all the data used in the application fits into the 2K SRAM used in **A**, so that an off-chip access is never necessary, resulting in the fastest possible performance. However, the data size is bigger than the 1K SRAM used in **D**, where the compulsory cache misses cause a slight degradation of performance. The results of *FFT* and *MatrixMult*, both highly computation-intensive applications, show that **A** is an inferior configuration for highly compute-oriented applications amenable to exploitation of locality of reference. Cache conflicts degrade performance of **B** and **C** in *SOR* and *DHRC*, causing worse performance than **A** (where there is no cache), and **D** (where conflicts are minimized by algorithm *AssignMemoryType*). Our technique resulted in an average improvement of 31.4% over **A**, 30.0% over **B**, and 33.1% over **C**.

In summary, our experiments on code kernels from typical embedded system applications show the usefulness of on-chip Scratch-Pad memory in addition to a data cache, as well the effectiveness of our data partitioning strategy.

6 Conclusions and Future Work

Modern embedded system applications use microprocessor cores along with memory and other co-processor hardware on the same chip. Since the CPU now forms only a part of the die, it is important to make optimal use of on-chip die area. In order to effectively use on-chip memory, we need to leverage the advantages of both data cache (simple addressing) and on-chip Scratch-Pad SRAM (guaranteed low access time) by including both types of memory modules in the same chip, with the data memory space being disjointly divided between the two.

We presented a strategy for partitioning variables (scalars and arrays) in embedded code into Scratch-Pad SRAM and data cache, that attempts to minimize data cache conflicts. Our experiments on code kernels from typical applications show a significant performance improvements (29 – 33%) over architectures of comparable area and random partitioning strategies.

Currently, our analysis is limited to individual sub-routines in the code. The optimal partitioning of global variables could change when all sub-routines are examined together. In the future, we plan to extend the analysis to embedded code with multiple sub-routines, and also develop techniques for addressing the problem of the relative sizing of SRAM and data cache.

7 Acknowledgment

This work was partially supported by grants from NSF(CDA-9422095) and ONR(N00014-93-1-1348).

8 References

[1] A. V. Aho, R. Sethi and J. D. Ullman, "Compilers – Principles, Techniques and Tools," Addison-Wesley, 1986.

[2] F. Catthoor and L. Svensson, "Application-Driven Architecture Synthesis," Kluwer Academic Publishers, 1993.

[3] LSI Logic Corporation, "CW33000 MIPS Embedded Processor User's Manual," 1992.

[4] M. Lam, E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991.

[5] D. Le Gall, "MPEG: A Video Compression Standard for Multimedia Applications," Communicatins of the ACM, Vol. 34, No. 4, pp 46-58, April 1991.

[6] P. Marwedel and G. Goosens, "Code Generation for Embedded Processors," Kluwer Academic Publ., 1995.

[7] P. R. Panda and N. D. Dutt, "1995 High Level Synthesis Design Repository," *Intl. Symp. on System Synthesis*, Cannes, September 1995.

[8] P. R. Panda, N. D. Dutt, and A. Nicolau, "Memory Organization for Improved Data Cache Performance in Embedded Processors," International Symposium on System Synthesis, La Jolla, CA, November 1996.

[9] D. A. Patterson and J. L. Hennessy, "Computer Organization & Design – The Hardware/Software Interface," Morgan Kaufman, 1994.

[10] W. H. Press, et. al., "Numerical Recipes in C: The Art of Scientific Computing," Cambridge University Press, 1992.

[11] J. Rawat, "Static analysis of cache performance for real-time programming," Masters thesis, Iowa State University, May 1993.

[12] A. Sudarsanam and S. Malik, "Memory Bank and Register Allocation in software Synthesis for ASIPs," Proceedings, IEEE/ACM International Conference on Computer Aided Design, Nov 5-9, 1995.

[13] James L. Turley, "New Processor Families Join Embedded Fray," Microprocessor Report, Vol. 8, No. 17, 26 December 1994.