Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications[*]

Preeti Ranjan Panda Nikil D. Dutt Alexandru Nicolau

Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA

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 array 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.

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 [5]. 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 [10]. 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 caches are fast local memory serving as 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*.

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 data 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.

Optimization techniques for improving the data cache performance of programs have been reported [4, 7, 9]. The analysis in [9] 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 [7], a data layout strategy for avoiding conflict misses is presented. However, array access patterns in some applications 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 cache conflict problems (Section 2). 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.

2 Problem Description

Figure 1(a) shows the architectural block diagram of an application employing a typical embedded core processor (e.g., the LSI Logic CW33000 RISC Microprocessor core[3]), 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*

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

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.

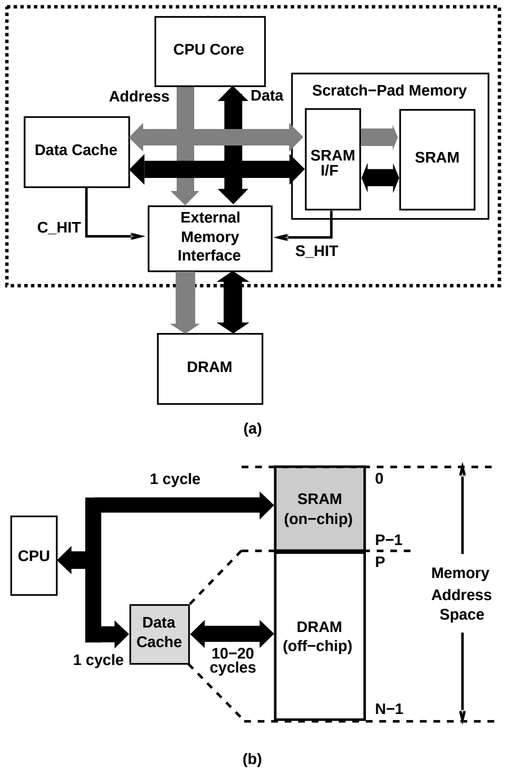


Figure 1. (a) Block Diagram of Embedded Processor Configuration (b) Division of Data Address Space between SRAM and DRAM

The data address space mapping is shown in Figure 1(b), for a 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 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.

We illustrate the necessity of including both data cache as well as Scratch-Pad SRAM in the architecture with the following example of a *Histogram Evaluation* code from a typical Image Processing application, which builds a histogram of 256 brightness levels for the pixels of an $N \times N$ image.

```
char BrightnessLevel [512][512];
int Hist [256];   /* Elements initialized to 0 */
. . .
for ($i = 0; i < N; i++$)
    for ($j = 0; j < N; j++$)
        /* For each pixel $(i, j)$ in image */
        level = BrightnessLevel $[i][j]$;
        Hist [level] = Hist [level] + 1;
```

Suppose the above code is executed on a processor configured with a data cache of size 1 KByte. The performance is degraded by the conflict misses in the cache between elements of the two arrays *Hist* and *BrightnessLevel*. Data layout techniques, such as [7] are not effective in eliminating the above type of conflicts, because the accesses to *Hist* are data-dependent. Note that this problem occurs in both direct-mapped as well as set-associative caches. However, the conflict problem can be solved elegantly if we include a Scratch-Pad SRAM in the architecture. Since the *Hist* array is relatively small, we can store it in the SRAM, so that it does not conflict with *BrightnessLevel* in the data cache. This storage assignment improves the performance of the *Histogram Evaluation* code significantly.

We present a strategy for partitioning scalar and array 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.

3 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.

3.1 Features Affecting Partitioning

3.1.1 Scalar Variables and Constants

In order to prevent interference with arrays in the data cache, we map all scalar variables and scalar constants to the Scratch-Pad memory. [1] 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

[1]We assume that register allocation, the task that assigns frequently accessed program variables such as loop indices to processor registers, has already been performed.

which will map into the same cache line as the scalars, causing conflict misses. The 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.

3.1.2 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.

3.1.3 Life-Times of Array 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 define a measure *Intersecting Life Times, ILT*(u), which indicates the number of array variables having a non-null intersection of life-times with u. The *ILT* value of each array 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.

3.1.4 Access Frequency of Array Variables

The *ILT* measure indicates the possibility of cache conflicts, but does not define the extent of these conflicts. To obtain a more accurate picture of the extent of conflicts, we have to consider the frequency of accesses. For example, if the number of accesses to an array d with high *ILT* 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 its high *ILT* value. For each array 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* arrays during the lifetime of an array, 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 array u, we define the *Interference Access Count, IAC*(u), to be the number of accesses to other arrays during the lifetime of u.

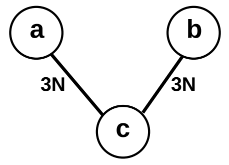
Note that the use of *VAC*(u) and *IAC*(u) separately may result in a misleading metric for the possible conflicts involving u. Clearly, the conflicts are determined jointly by the two factors considered together. A good indicator of the conflicts involving array u is the product of the two individual 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 array variables with high *IF* values into the SRAM.

3.1.5 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 2(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 2. (a) Example Loop (b) Loop Conflict Graph

We note that arrays a and b have an identical access pattern, which is different from that of c. Data placement techniques [7] 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 2(b) is generated. We have one access to a and two to c in one iteration of the loop. The 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* [2]. Note that if the access pattern of an array is data-dependent in a loop, as in the *Hist* array of the *Histogram Evaluation* example (Section 2) it is not compatible with other arrays accessed in the same loop.

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.

3.2 The Partitioning Algorithm

The algorithm for determining the mapping decision of each (scalar and array) program variable to SRAM or DRAM/cache uses the factors mentioned in Section 3.1. Due to lack of space, we briefly summarize the algorithm here. The complete algorithm is described in [8].

We first assign the scalar constants and scalar variables to the SRAM, and the arrays that are larger than the Scratch-Pad memory, to the DRAM. For the remaining (n) arrays (all of which are small enough to fit into the SRAM), we generate the life-time intervals and compute the *LCF* and *IF* values (Section 3.1). We sort the $2n$ interval end-points thus generated, and traverse them in increasing order. For each array (u) encountered, if there is sufficient SRAM space for u *and* all arrays with life-times intersecting the life-time interval of u, with more critical *LCF* and *IF* numbers, we map u to SRAM; otherwise we map it to DRAM/cache. The algorithm has an overall complexity of $O(n^2)$.

4 Experiments

We present the results of simulation experiments we performed on several benchmark examples that frequently occur as code kernels in embedded applications, to evaluate the efficacy of our Scratch-Pad memory/DRAM data partitioning algorithm. We use 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 assume a total on-chip memory capacity of 2 KB, and consider different uses of this memory: only Scratch-Pad SRAM; equal cache and SRAM; and only cache. We compare 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**) *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; (**B**) *Data cache of size 2K*: in this case, there is no SRAM in the architecture; (**C**) *Random Partitioning*: in this case, we use a 1K SRAM and 1K Data cache, and a random data partitioning technique (variables are considered in the order they appear in the code, and mapped into SRAM if there is sufficient space); and (**D**) *Our Technique*: here we use a 1K SRAM and 1K data cache, and our data partitioning algorithm. Thus, in each case, the total on-chip memory is 2 KBytes. We use a direct-mapped data cache with line size = 4 words, and an off-chip memory latency given by: (10 cycles for initialization) + (1 cycle per word in cache line) $\times$ (4 words in cache line) = 14 cycles.

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

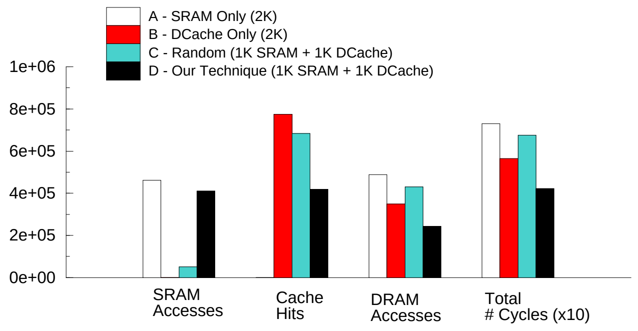


Figure 3. Performance Details for *Beamformer* Example

Figure 3 shows the details of the memory accesses for the *Beamformer* benchmark example [6]. 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 (Figure 3 shows the total number of cycles scaled down by a factor of 10).

Figure 4 presents a comparison of the performance for the four configurations **A, B, C**, and **D** mentioned earlier,

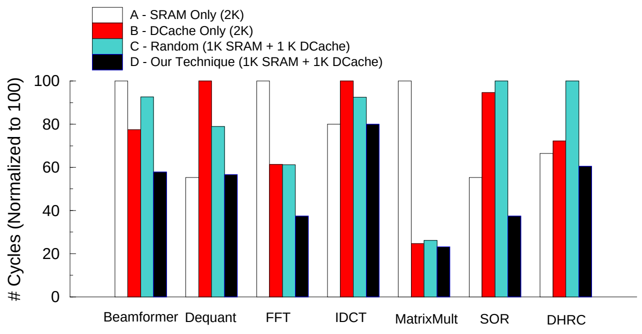


Figure 4. Performance Comparison of Configurations A, B, C, and D

on code kernels extracted from seven benchmark applications. *Dequant* is the de-quantization routine in the MPEG decoder application. *IDCT* is the Inverse Discrete Cosine Transform, also used in the MPEG decoder. *SOR* is the Successive Over-Relaxation algorithm, frequently used in scientific computing [6]. *MatrixMult* is the matrix multiplication operation, optimized for maximizing spatial and temporal locality by reordering the loops. *FFT* is the Fast Fourier Transform application [6]. *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 4. 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 our algorithm). Our technique results in an average improvement of 31.4% over **A** (only SRAM), 30.0% over **B** (only cache), and 33.1% over **C** (equal cache and SRAM – random partitioning).

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.

5 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 structures in the same chip, with the data memory space being disjointly divided between the two.

We presented a strategy for partitioning scalar and array variables 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 improvements in memory latency (30 – 33%) over architectures with comparable on-chip memory capacity and random partitioning strategies.

Our analysis assumes that the loop bounds and the number of invocations of sub-routines are statically known. If they are dynamically determined, we would have to rely on profiling information. Currently, our analysis is limited to a single execution thread. We are investigating the assignment of data to SRAM and cache in the context of multiple execution threads. In the future, we plan to consider the area and performance trade-offs between the architecture presented in this paper, and one with multiple data caches (each mapped to a disjoint memory space), which would require a data partitioning technique similar to the one we have presented here. We also plan to develop techniques for addressing the problem of the relative sizing of SRAM and data cache.

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, et. al., “The cache performance and optimizations of blocked algorithms,” Proceedings ASPLOS, April 1991.

[5] P. Marwedel and G. Goosens, “Code Generation for Embedded Processors,” Kluwer Academic Publ., 1995.

[6] P. R. Panda and N. D. Dutt, “1995 High Level Synthesis Design Repository,” Intl. Symp. on System Synth., 1995.

[7] P. R. Panda, N. D. Dutt, and A. Nicolau, “Memory Organization for Improved Data Cache Performance in Embedded Processors,” Intl. Symp. on System Synth., 1996.

[8] P. R. Panda, N. D. Dutt, and A. Nicolau, “SRAM vs. Data Cache: The Memory Data Partitioning Problem in Embedded Systems,” Tech. Rep. #96-42, U.C. Irvine, Sep. 1996.

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

[10] James L. Turley, “New Processor Families Join Embedded Fray,” Microprocessor Report, Vol. 8, No. 17, Dec. 1994.