On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems

PREETI RANJAN PANDA
Synopsys, Inc.
and
NIKIL D. DUTT and ALEXANDRU NICOLAU
University of California at Irvine

Efficient utilization of on-chip memory space is extremely important in modern embedded system applications based on processor 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, so that critical data can be stored there with a guaranteed fast access time. 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. We also present extensions of our proposed memory assignment strategy to handle context switching between multiple programs, as well as a generalized memory hierarchy. Our experiments on code kernels from typical applications show that our technique results in significant performance improvements.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

Additional Key Words and Phrases: Data cache, data partitioning, on-chip memory, memory synthesis, scratch-pad memory, system design, system synthesis

1. INTRODUCTION

Modern embedded systems are characterized by a trend towards increasing levels of chip-level integration. System design is gradually changing its

This work was partially supported by grants from NSF (CDA-9422095) and ONR (N00014-93-1-1348). We are grateful for their support.
Authors' addresses: P. R. Panda, Synopsys, Inc., 700 East Middlefield Road, Mountain View, CA 94043; N. D. Dutt and A. Nicolau, Dept. of Information and Computer Science, University of California at Irvine, Irvine, CA 92697.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2000 ACM 1084-4309/00/0700–0682 $5.00

focus from the integration of chips on a board to the integration of complex electronic components on the same chip. This shift in focus, which is possibly due to increasing chip capacities, is driven by the goal of cost reduction that stems from reduced chip count. Higher levels of on-chip integration also imply that the basic building blocks used for chip design change from register transfer level (RTL) components such as ALUs, registers, multiplexers, etc., to system-level components such as processor cores, memories, and coprocessors.

Flexibility and short design time considerations drive the use of CPU cores as instantiable modules in system designs [Marwedel and Goosens 1995]. The flexibility of a processor core arises out of the fact that parameters such as on-chip memory configuration can be suitably selected for the specific application under consideration, whereas this choice is not available in a packaged (off-the-shelf) processor. The short design time arises from the fact that the processor core is a predesigned and debugged macroblock. Examples of commercial processor cores commonly used in system design are LSI Logic's MIPS-based CW33000 series [LSI Logic Corporation 1992] and the ARM series from Advanced RISC Machines [Turley 1994].

Another important family of building blocks characterizing system design is on-chip memory. The types of on-chip memory commonly 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**, refers to data memory residing on-chip 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 allow fast access to their residing data, whereas an access to off-chip memory (usually DRAM) requires relatively longer access times. 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 the cache is subject to *compulsory, capacity,* and *conflict misses* [Patterson and Hennessy 1994].

The concept of Scratch-Pad memory is an important architectural consideration in modern embedded systems, where advances in fabrication technology have made it possible to combine DRAM and ordinary logic on the same chip. As a result, it is possible to incorporate *embedded DRAMs* along with a processor core in the same chip [Margolin 1997; Wilson 1997]. Since data stored in embedded DRAM can be accessed much faster than that in off-chip DRAM, a related optimization problem that arises in this context is how to identify critical data in an application for storage in on-chip memory. In this paper we use the terminology *Scratch-Pad SRAM* to include the embedded DRAM configuration.

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 applica-

tion 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

Several memory-related issues in system synthesis have been addressed in recent years. The problem of mapping scalar variables into single- and multiported on-chip memory structures is addressed in Balakrishnan et al. [1988]; Ahmad and Chen [1991] ; Stok and Jess [1992]; Kim and Liu [1993]. An algorithm for clustering behavioral array variables to determine a low-cost allocation of multiport memory modules was proposed by Ramachandran et al. [1994]. To avoid the offset addition operation during synthesis, faster memory address generation techniques were proposed by Karchmer and Rose [1994] and Schmit and Thomas [1995]. Techniques for synthesizing physical memory structures from a library of memory components are presented in Karchmer and Rose [1994]; Bakshi and Gajski [1995]; Jha and Dutt [2000]. In Wuytack et al. [1996], the authors present a flow graph balancing technique for reducing the worst-case memory access bandwidth satisfying a given time constraint. The reduced bandwidth results in a more cost-effective on-chip memory architecture.

Vanhoof et al. [1991] presented a strategy for storing data streams in DSP applications in register files and SRAMs. In Lippens et al. [1993], the authors present a hierarchical model of data streams and an algorithm for mapping data streams into multiport memories implemented in the PHIDEO synthesis system. Balasa et al. [1995] describe a dataflow analysis-based technique for estimating the on-chip memory storage required for multidimensional signals in a nonprocedural behavioral specification implemented in the CATHEDRAL synthesis environment. The memory estimation problem was also studied by Verbauwhede et al. [1994], who employ an ILP formulation modeling the data dependence and execution sequences. Our work differs from the ones above in that we address the problem of memory assignment to data in a hierarchical memory architecture.

The problem of efficiently utilizing memory banks in DSPs was recently addressed. Techniques for partitioning variables for simultaneous access into two memory banks of the Motorola 56000 DSP processor are reported in Sudarsanam and Malik [1995] and Saghir et al. [1996]. However, parallel access is not the objective of the partitioning problem we are addressing—we wish to maximize performance in the sequential access scenario.

For embedded processor-based systems, Liao et al. [1995a; 1995b] describe a technique for reducing the size of compiled code targeting DSP architectures, with the objective of minimizing on-chip instruction ROM. Tomiyama and Yasuura [1996a] have studied the problem of code placement for improving instruction cache performance. However, the code placement strategy could lead to large increases in memory size. In

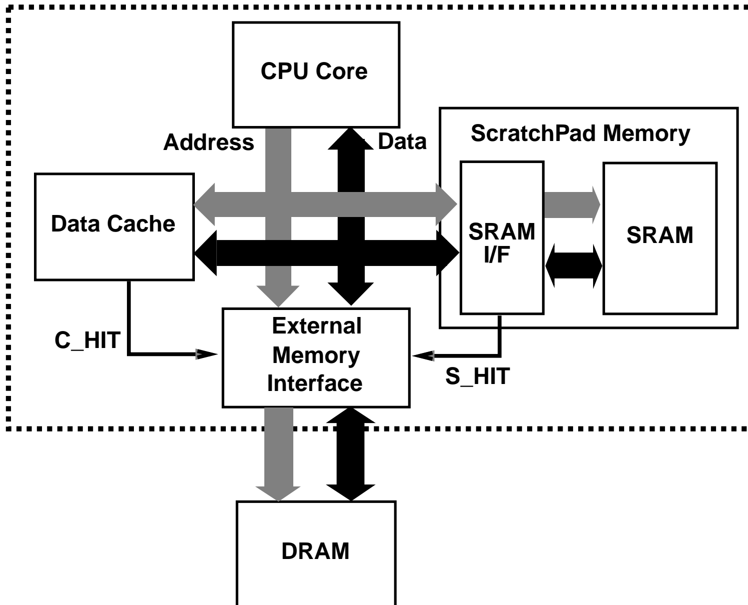


Fig. 1. Block diagram of a typical embedded processor configuration.

Tomiyama and Yasuura [1996b], they present a technique for code placement targeting instruction cache performance improvement under a code size constraint. Similarly, optimization techniques for improving the data cache performance of programs have been reported [Lam et al. 1991; Rawat 1993; Panda et al. 1996]. The analysis in Rawat [1993] is limited to scalars, and hence not generally applicable. Iteration space *blocking* for improving data locality is studied in Lam et al. [1991]. This technique is also limited to the type of code that yields naturally to blocking. In Panda et al. [1996], 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.

3. PROBLEM DESCRIPTION

Figure 1 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, interfacing 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**

[1]For example, the LSI Logic CW33000 RISC Microprocessor core [LSI Logic Corporation 1992].

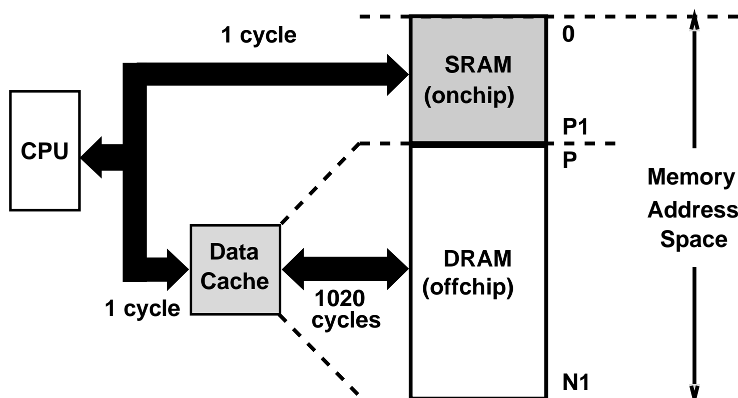


Fig. 2. Division of data address space between SRAM and DRAM.

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.

The data address space mapping is shown in Figure 2, for a sample addressable memory of size N data words. Memory addresses $0 \ldots P - 1$ map into the on-chip Scratch-Pad memory and have a single processor cycle access time. Thus, in Figure 1, 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.

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, our goal 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 overall memory access 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. We motivate the need for both types of on-chip memory using the following example. 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:

(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. 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. In many cases, having only a cache results in certain unavoidable cache misses that degrade performance due to stalled CPU cycles. We illustrate this 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;
   end for
end for
```

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 Panda et al. [1996] are not effective in eliminating the above types of conflict because accesses to *Hist* are data-dependent. Note that this problem occurs in both direct-mapped as well as set-associative caches.

(4) *1K Data cache + 1K Scratch-Pad SRAM:* The problem incurred in (3) above could be elegantly solved using the architecture of Figure 1, with a 1K data cache and 1K SRAM. 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. The single-cycle access time guarantee for data stored in SRAM and the possibility of avoiding conflicts makes the architecture with a combination of cache and Scratch-Pad memory superior to cache alone.

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—for example, when 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.

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. We assume in this work that the array sizes and loop bounds are known, either statically or through profiling.[2]

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 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 code characteristics: (1) scalar variables and constants; (2) size of arrays; (3) life-times of variables; (4) access frequency of variables; and (5) conflicts in loops. We describe below each of the above factors and how our partitioning strategy addresses 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 off-chip memory is the natural choice, as it simplifies the array addressing. If a part of the array were to map into

[2]This is a reasonable assumption for many embedded applications.

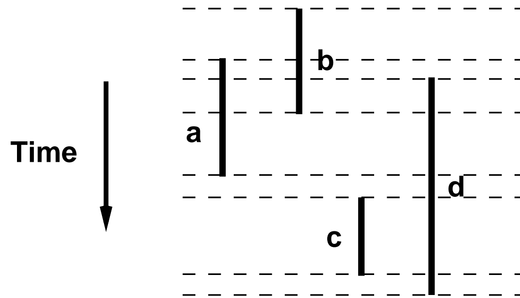


Fig. 3. Example lifetime distribution.

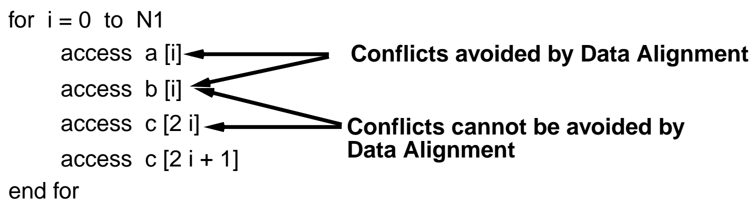
the SRAM, the compiler would have to generate book-keeping code that keeps track of which region of the array is addressed, thereby making the code inefficient. Further, since most loops access array elements more or less uniformly, there is little or no motivation to map different parts of the same array to memories with different characteristics.

4.1.3 *Lifetimes of Variables*. The *lifetime* of a variable, defined as the period between its *definition* and *last use* [Aho et al. 1993], 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.

Life-time information can also be used to avoid possible conflicts among arrays. To simplify the conflict analysis, we assume that an array variable accessed inside a loop is *alive* throughout the loop. Figure 3 shows an example lifetime distribution of four array variables, a, b, c, and d. Since b and c have disjoint lifetimes, the memory space allocated to array b can be reused for array c. Since arrays a and d have intersecting lifetimes, cache conflicts among them can be avoided by mapping one of them to the SRAM. However, since on-chip resources are scarce, we need to determine which array is more critical. The quantification of this criticality depends on the access frequency of the variables.

4.1.4 *Access Frequency of Variables*. To obtain an estimate of the extent of conflicts, we have to consider the frequency of accesses. For example, in Figure 3, note that array d could potentially conflict with all the other three arrays, so we could consider storing d in the SRAM. However, if the number of accesses to d is relatively small, it is worth considering some other array (e.g., a) first for inclusion in SRAM, since d does not play a significant role in cache conflicts. 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. For each



(a)

LCF (a) = k (a) + (k (c)) = N + 2 N = 3N

LCF (b) = k (b) + (k (c)) = N + 2 N = 3N

LCF (c) = k (c) + (k (a) + k(b)) = 2N + N + N = 4 N

(b)

Fig. 4. (a) Example loop; (b) computation of LCF values.

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 sum of the two metrics. We define the *interference factor, IF,* of a variable u as

$$IF(u) = VAC(u) + IAC(u) \tag{1}$$

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*. The *IF* factor defined in the previous section can be used to estimate conflicts in straightline code and conditionals (i.e., nonloop code). In the case of arrays accessed in loops, 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 4(a).

We notice that arrays a and b have an identical access pattern, which is different from that of c. Data alignment techniques [Panda et al. 1996] can be used to avoid data cache conflicts between a and b—the arrays can be appropriately displaced in memory so that they never conflict in the cache. However, when the access patterns are different, cache conflicts are unavoidable (e.g., between b and c—the access patterns are different due to differing coefficients of the loop variables in their index expressions). 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 define the *loop conflict factor, LCF* for a variable u as

$$LCF(u) = \sum_{i=1}^{p}(k(u) + \sum_{v}k(v)) \tag{2}$$

where the summation $\Sigma_{i=1}^{p}$ is over all loops $(1...p)$ in which u is accessed and Σ_v is over all arrays (other than u) that are also accessed in loop i, and for which it is not possible to use data placement techniques to completely eliminate cache conflicts with u. In the example above, where we have only one loop ($p = 1$), the LCF values shown in Figure 4(b) are 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 $N + 2N = 3N$. Thus, we have $LCF(a) = 3N$, since cache conflicts between a and b can be completely eliminated by data placement techniques.[3] Similarly, $LCF(b) = 3N$, and $LCF(c) = 4N$. The *LCF* value 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 Formulation of the Partitioning Problem

In Sections 4.1.4 and 4.1.5, we defined two factors, interference factor (*IF*) and loop conflict factor (*LCF*). We integrate these two factors and arrive at an estimate of the total number of accesses that could theoretically lead to conflicts due to an array. We define the *total conflict factor (TCF)* for an array u as

$$TCF(u) = IF(u) + LCF(u) \tag{3}$$

where the $IF(u)$ value is computed over a lifetime that excludes regions of code where u is accessed inside loops. $TCF(u)$ gives an indication of the total number of accesses that are exposed to cache conflicts involving array u, and hence denotes the importance of mapping u to the SRAM.

We formulate the data partitioning problem as follows: given a set of n arrays $A_1 ... A_n$, with TCF values $TCF_1 ... TCF_n$, sizes $S_1 ... S_n$, and an SRAM of size S, find an optimal subset $Q \subseteq \{1, 2, ..., n\}$ such that $\Sigma_{i \in Q} S_i \leq S$ and $\Sigma_{i \in Q} TCF_i$ is maximized.

[3] Conflicts between a and b can be eliminated because their respective array index expressions differ only by a constant, i.e., independent of the loop variable.

We note that the problem, as formulated above, is similar to the *Knapsack Problem* [Garey and Johnson 1979]. However, there is an additional factor to consider in this case—*several arrays with nonintersecting lifetimes can share the same SRAM space*. Thus, the data-partitioning problem is NP-hard, since a polynomial-time algorithm for solving this problem, with all variable life-times constrained to be identical, would also solve the Knapsack problem in polynomial time.

4.3 Solution of the Partitioning Problem

An exhaustive-search algorithm to solve the memory assignment problem would have to first generate clusters of all combinations of *compatible arrays* (arrays that can share the same SRAM space) and then generate all possible combinations of these clusters and pick the combination with total size fitting into the SRAM that maximizes the cumulative TCF value. This procedure requires $O(2^{2^n})$ time, which is unacceptably expensive, since the function $y = 2^{2^n}$ grows very rapidly, even for small values of n.

In our solution to the memory data-partitioning problem, we first group arrays that could share SRAM space into clusters, and then use a variation of the *value-density* approximation algorithm [Garey and Johnson 1979] for the knapsack problem to assign clusters to the SRAM. The approximation algorithm first sorts all the items in terms of *value per weight* (i.e., the *value density*), and selects items in decreasing order of value density until no more items can be accomodated. We define *access density (AD)* of a cluster c as

$$AD(c) = \frac{\sum_{v \in c} TCF(v)}{\max\{\text{size}(v) \mid v \in c\}} \tag{4}$$

and use this factor to assign clusters of arrays into the SRAM. Note that the denominator is the size of the largest array in the cluster. This is because, at any time, only one of the arrays is *live*, and consequently the arrays, which share the same memory space, need to be assigned only as much memory as the largest in the cluster.

The memory data-partitioning algorithm *MemoryAssign* is shown in Figure 5. The input to the algorithm is the SRAM size and the application program P, with the register-allocated variables marked. 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. For the remaining arrays, we first generate the *compatibility graph* G, in which the nodes represent arrays in program P, and an edge exists between two nodes if the corresponding arrays have disjoint lifetimes. The analogous problem of mapping scalar variables into a register file is solved by *clique partitioning* [Gajski et al. 1992] of the compatibility graph. A clique is a fully connected subgraph of the compatibility graph and represents a single register—all nodes in a clique have disjoint lifetimes and can

```
Algorithm MemoryAssign
Input: Application program P with Register-allocated variables marked;
SRAM_Size: Size of Scratch-Pad SRAM
Output: Assignment of arrays to SRAM/DRAM
    AvSpace = SRAM_Size    -- Available SRAM space
    Let U = {array  u|u  is an array in P}
    -- U is the set of all behavioral arrays in program P
    Let W = ϕ    -- W is the set of arrays assigned to DRAM
    for all variables v
        if v is a scalar variable or constant
            Assign v to SRAM
            AvSpace = AvSpace − size(v)
        else
            if size(v) > SRAM_Size
                W = W ∪ {v}    -- Assign v to DRAM
            end if
        end if
    end for
    Generate compatibility graph G from life-times of remaining arrays
    U = U − W    -- U is the set of all arrays <SRAM size
    while (U ≠ ϕ)
        for each array u ∈ U
            Find largest clique c(u) in G such that u ∈ c(u) and
            size(v) ≤ size(u) ∀v ∈ c(u)
            Compute access density AD(u) = (Σ_{v∈c(u)} TCF(v)) / size(u)
        end for
        Assign clique c(i) to SRAM, where AD(i) = max {AD(u)|u ∈ U}
        -- Assign cluster with highest access density to SRAM
        AvSpace = AvSpace − size(c)    -- size(c) =size of largest array in c
        X = {v ∈ U|size(v) > AvSpace}
        -- X = set of arrays in U larger than AvSpace
        W = W ∪ X  -- Arrays in X are mapped to DRAM
        U = U − {v|(v ∈ c)} − X
        -- Remove from U arrays assigned to SRAM and arrays in X
    end while
    Assign arrays in W to DRAM
end Algorithm
```

Fig. 5. Memory assignment of variables into SRAM/DRAM.

be mapped into the same register. The clique-partitioning problem, which is known to be NP-complete [Garey and Johnson 1979], is to divide graph G into a minimal number of disjoint cliques—this minimizes the number of registers required for storing the scalar variables. However, we cannot apply the clique-partitioning algorithm in a straightforward manner when we cluster arrays because *the cliques we are interested in are not necessarily disjoint*. For example, consider a clique consisting of three arrays, A, B, and C, where size (A) is larger than size (B) and size (C). Assuming that the currently available SRAM space during one iteration of the assignment is *AvSpace*, the clique $\{A, B, C\}$ will not fit into the SRAM if size $(A) > AvSpace$. However, the subset $\{B, C\}$ will fit into the SRAM if

size (B) and size (C) are smaller than *AvSpace*. Thus, we need to consider both cliques $\{A, B, C\}$ and $\{B, C\}$ during the SRAM assignment phase.

To handle the possibility of overlapping clusters, we generate one cluster (clique in graph G) $c(u)$ for every array u, which consists of u and all nodes v in graph G, such that size $(v) \leq$ size (u), and the subgraph consisting of the nodes of $c(u)$ is fully connected. Thus, for each array u, we attempt to select the clique with maximum access density $AD(u)$ (i.e., largest ΣTCF, since maximum memory space required = size(u)). This problem is easily seen to be NP-hard by using an instance where $TCF(u) = 1$ and size (u) = 1 for every node u, and inferring a reduction from the *maximal clique problem* [Garey and Johnson 1979]. We use a greedy heuristic [Tseng and Siewiorek 1986], which starts with a single node clique consisting of node u and iteratively adds neighbor v with maximum $TCF(v)$, provided size (v) $\leq$ size (u) and v has an edge with all nodes in the clique constructed so far. After generating the cliques for each array u, we assign the one with the highest access density to the SRAM, following which we delete all nodes in the clique and connecting edges from the graph G. In case of more than one array with the same access density, we choose the array with the larger *TCF* value. After every SRAM assignment, we assign all the unassigned arrays that are larger than the available SRAM space to the DRAM. We then iterate through the process until all nodes in G have been considered. Note that the clustering step is recomputed in every iteration because the overlapping nature of the clusters might cause the optimal clustering to change after a clique is removed.

Analyzing algorithm *MemoryAssign* for computational complexity, we note that determining the largest clique is the computationally dominant step, with our greedy heuristic requiring $O(n^2)$ time, where n is the number of arrays in program P. Determining the cliques for all arrays requires $O(n^3)$ time in each iteration. Since the algorithm could iterate a maximum of $O(n)$ times, the overall complexity is $O(n^4)$.

When applying algorithm *MemoryAssign* to practical applications, we noticed that the number of arrays n is not necessarily small, but the number of arrays in a program with nonintersecting lifetimes is usually small. Consequently, the compatibility graph of arrays for a program tends to be sparse. This is in contrast to the corresponding graph for scalars, which is usually dense. This indicates that an exhaustive-search algorithm for determining the cliques might be acceptable. In our implementation, we use exhaustive search if the number of edges in a compatibility graph is $\leq 2n$, where n is the number of nodes, otherwise we use our greedy heuristic.

5. CONTEXT SWITCHING

Although an embedded system based on a processor core is likely to execute only a single program most of the time, some applications might require a

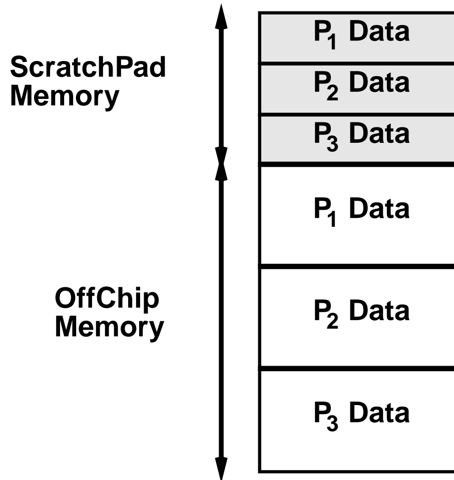


Fig. 6. Memory space partitioned between data from different programs.

context switch among different programs. In the architecture of Figure 1, if the entire SRAM is allotted to one program, the rest of the programs would not benefit from the presence of the SRAM in the architecture. One logical solution is to partition the SRAM space and allocate different partitions to different programs (Figure 6).[4] This partitioning of the SRAM space is acceptable because the number of programs involved is usually small in an embedded system that executes only a single application.

The problem we address in this section is that of efficient utilization of the SRAM in the presence of multiple application programs. To allocate the SRAM space efficiently among arrays of different programs, we use the same *TCF* metric identified in Eq. (3). We first build a compatibility graph $CG(i)$ for each program P_i, and build the forest G consisting of the union of all $CG(i)$. Finally, we use the same procedure used in algorithm *Memory-Assign* (Section 4.3) for the clustering and memory assignment of arrays of different programs. Note that, since we have no prior knowledge of the order in which the different programs will be executed, we assume that arrays from different programs cannot share SRAM space. The partitioning algorithm *SRAMPartition* is summarized in Figure 7.

As discussed in Section 4.3, the computational complexity of the partitioning procedure is $O(n^4)$, where n is now the total number of arrays in all the programs. An important point to remember is that the SRAM space needs to be partitioned among variables in different programs only if the programs are expected to be executed in a threaded fashion. If the execution order of the programs is known to be sequential, then the entire SRAM

[4]Note that, unlike the register file, it is usually prohibitively expensive to save the contents of the entire Scratch-Pad SRAM in external DRAM on a context switch because it would add a considerable delay to the context-switching time.

```
Algorithm SRAMPartition
Input: Application programs P_1 ... P_g with Register-allocated variables marked;
SRAM_Size: Size of Scratch-Pad SRAM
Output: Assignment of arrays in each program to SRAM/DRAM
    AvSpace = SRAM_Size   -- Available SRAM space
    Let W = φ   -- W = set of arrays assigned to DRAM
    for all programs P_i
        for all variables v in P_i
            if v is a scalar variable or constant
                Assign v to SRAM
                AvSpace = AvSpace − size(v)
            else
                if size(v) > SRAM_Size
                    W = W ∪ {v}   -- Assign v to DRAM
                end if
            end if
        end for
        Compute TCF(u) for every array u in P_i, where u ∉ W
        Generate compatibility graph CG(i) for P_i from lifetimes of arrays in P_i ∉ W
    end for
    G = ∪_{1≤i≤n} CG(i)   -- G is the forest consisting of n (disjoint) CGs
    Let U = {array  u|u  is a node in G}
    -- U = set of arrays smaller than SRAM size in all programs
    while (U ≠ φ)
        for each array u ∈ U
            Find largest clique c(u) in G such that u ∈ c(u) and
            size(v) ≤ size(u) ∀v ∈ c(u)
            Compute access density AD(u) = (Σ_{v∈c(u)} TCF(v)) / size(u)
        end for
        Assign clique c(i) to SRAM, where AD(i) = max {AD(u)|u ∈ U}
        -- Assign cluster with highest access density to SRAM
        AvSpace = AvSpace − size(c)   -- size(c) = size of largest array in c
        X = {v ∈ U|size(v) > AvSpace}
        -- X =set of arrays in U larger than AvSpace
        W = W ∪ X  -- Arrays in X are mapped to DRAM
        U = U − {v|(v ∈ c)} − X
        -- Remove from U arrays assigned to SRAM and arrays in X
    end while
end Algorithm
```

Fig. 7. Algorithm for partitioning SRAM space among multiple programs.

is available to every program, and algorithm *MemoryAssign* can be used for memory assignment by considering each program separately.

5.1 Programs with Priorities

Some embedded systems are characterized by programs with varying priorities, which might arise from frequency of execution or the relative criticality of the programs. In such a case, it is necessary to modify algorithm *SRAMPartition* to take the relative priorities of programs $P_1 \ldots P_g$ into account. The approach we adopt in this case is to weight the *TCF* values for each array in a program by the priority of the respective

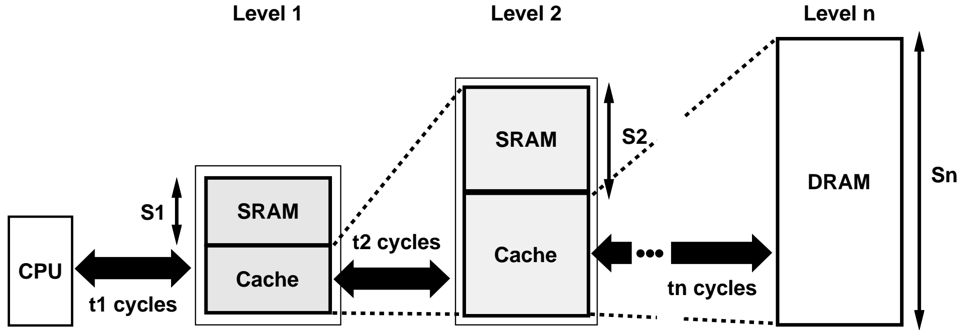


Fig. 8. Generalized memory hierarchy.

program. That is, if the priorities of programs $P_1 \ldots P_g$ are given by $r_1 \ldots r_g$, we update $TCF(u) = TCF(u) \times r(i)$ for every array u in program P_i. The rest of the partitioning procedure remains the same as before.

6. GENERALIZED MEMORY HIERARCHY

The architecture shown in Figure 1 can be generalized into an n-level hierarchy shown in Figure 8, with the data cache at each level acting as the interface between the levels above and below it. The total Scratch-Pad memory at level i is S_i and the time required for the processor to access data at that level is t_i cycles. Such an architecture becomes realisitic with the advent of embedded DRAM technology, where large amounts of memory can now be integrated on-chip.

Algorithm *MemoryAssign* (and algorithm *SRAMPartition*) can be generalized in a straightforward manner to effect memory assignment to program data in the hierarchical memory architecture shown in Figure 8. Since the memory access times of the different levels follow the ordering $t_1 < t_2 < \ldots < t_n$, we store the most critical data in level 1, followed by level 2, etc. Thus, in the first iteration of the memory data partitioning, we invoke algorithm *MemoryAssign* with the SRAM size $= S_1$ and off-chip memory size $= S_2 + S_3 + \ldots S_n$. After the assignment to the first level Scratch-Pad memory, we remove the variables assigned to the first level and continue the assignment for the remaining variables, now with SRAM size $= S_2$ and off-chip memory size $= S_3 + S_4 + \ldots S_n$. In other words, at every level i, the most critical data, as determined by the metric in Section 4.3, is mapped into the Scratch-Pad memory of size S_i.

The above strategy is a greedy algorithm because it is an extension of the *MemoryAssign* algorithm, which itself adopts a greedy strategy. As noted in Section 4.3, if the number of arrays involved is small, then an exhaustive search strategy could also be adopted for the memory assignment.

7. EXPERIMENTS

We performed simulation experiments 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 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 and measured the total number of processor cycles required to access the data during execution of the examples 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 scalar and as many arrays as will fit into the SRAM and the rest into 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 *MemoryAssign* 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;

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 *for initialization* + 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. This is a popular model for cache/off-chip memory traffic.

7.1 Benchmark Examples

Table I shows a list of benchmark examples, on which we performed our experiments, and their characteristics. Columns 1 and 2 show the name and a brief description of the benchmarks. Columns 3 and 4 give the number of scalar and array variables, respectively, in the behavioral specifications. Column 5 gives the total size of the data in the benchmarks.

Beamformer [Panda and Dutt 1995], a DSP application, represents an operation involving temporal alignment and summation of digitized signals from an N-element antenna array. *Dequant* is the dequantization routine in the MPEG decoder application [Gall 1991]. *IDCT* is the inverse discrete

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

Table I. Profile of Benchmark Examples

Benchmark	Description	No. of Scalars	No. of Arrays	Data Size (Bytes)
Beamformer	Radar Application	7	7	19676
Dequant	Dequantization Routine (MPEG)	7	5	2332
FFT	Fast Fourier Transform	20	4	4176
IDCT	Inverse Discrete Cosine Transform	20	3	1616
MatrixMult	Matrix Multiplication	5	3	3092
SOR	Successive Over-Relaxation	4	7	7184
DHRC	Differential Heat Release Computation	28	4	3856

cosine transform, also used in the MPEG decoder. *SOR* is the successive over-relaxation algorithm, frequently used in scientific computing [Press et al. 1992]. *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, which models the heat release within a combustion engine [Catthoor and Svensson 1993].

7.2 Detailed Example: Beamformer

Figure 9 shows the details of the memory accesses for the *Beamformer* benchmark example. 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 consequence, 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 judicious mapping of the most frequently accessed and conflict-prone elements into Scratch-Pad memory.[6]

7.3 Performance of SRAM-Based Memory Configuration

Figure 10 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. The number of cycles for each application is normalized to 100. In the *Dequant* example, **A** slightly outperforms **D** because all the data used in the application fits into the 2K

[6]Figure 9 shows the total number of cycles scaled down by a factor of 10

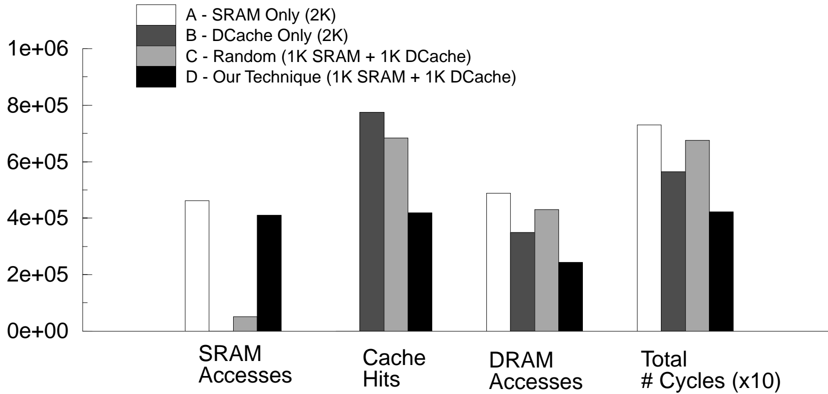


Fig. 9. Memory access details for the *Beamformer* example.

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 computation-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.

7.4 Performance in the Presence of Context Switching

To study the effectiveness of our technique for handling context switching between multiple programs described in Section 5, we simulated the effect of context switching between three benchmark programs: *Beamformer, Dequant*, and *DHRC*, with the same SRAM/cache configuration as in Section 7.3 (1KB SRAM and 1KB data cache).

In our experiment, we compared the performance of two different SRAM assignment techniques: (1) the technique outlined in Section 5; and (2) a random assignment technique (*random*), where one application (out of three) is selected at random, and our technique for SRAM assignment for a single program (configuration **D** in Figure 10) is used to assign SRAM space. In case SRAM space is unused, another program is selected at random, and so on.

We use the following relative priorities (Section 5.1) for the three examples: *Beamformer* - 1; *Dequant* - 2; and *DHRC* - 3. In the experiment,

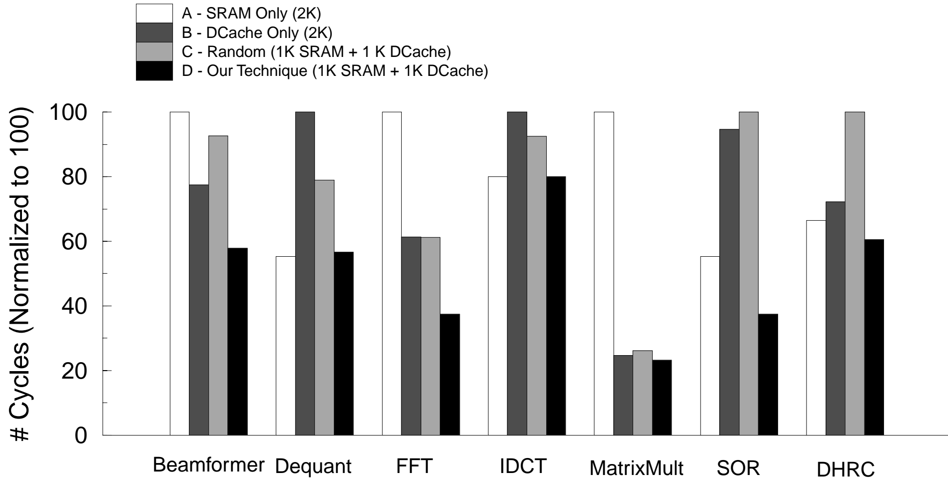


Fig. 10. Performance comparisons of configurations A, B, C, and D.

the priority value is reflected in the frequency of execution of the programs. Thus, *Beamformer* is executed once, *Dequant* twice, and *DHRC* three times.

After performing the memory assignment, we generate a trace of memory locations accessed by each program. The trace data structure is a list of nodes where each nodes contains the memory address and the memory access type (READ or WRITE) for each memory access. For the three programs, *Beamformer, Dequant*, and *DHRC*, we generate 3 traces of memory accesses: T_1, T_2, and T_3 respectively. Since *Dequant* is executed twice (priority = 2), we generate a new trace T'_2 for *Dequant* by duplicating and concatenating two copies of trace T_2. Similarly, we generate T'_3 for *DHRC* by concatenating three copies of the T_3 trace. We now simulate the SRAM/data cache activity by randomly alternating execution between the three traces: T_1, T'_2, and T'_3.

Figure 11 shows the memory access details for the above experiment. The number of *SRAM accesses* is very low for *random* because the number of array accesses in the program whose arrays were assigned to the SRAM by *random* is relatively lower. *Cache hits* were higher for *random* because it assigned more accesses to DRAM data. In our technique, the number of accesses to external DRAM was significantly lower because of efficient SRAM utilization. Overall, there is a 37% reduction in the total processor cycles due the memory accesses. This experiment indicates the utility of a judicious SRAM partitioning/assignment strategy allowing for context switching between different applications.

8. CONCLUSIONS

Modern embedded system applications use processor cores along with memory and other coprocessor hardware on the same chip. Since the CPU now forms only a part of the die, it is important to make optimal utilization

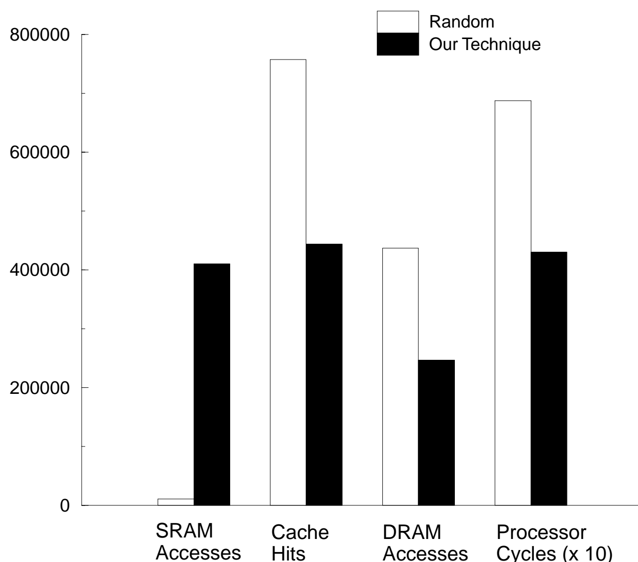


Fig. 11. Performance in the presence of context switching between *Beamformer, Dequant*, and *DHRC* examples.

of the 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 memory (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 memory and data cache that attempts to minimize data cache conflicts. Our experiments on code kernels from typical applications show significant performance improvements (29–33% reduction in memory access time) over architectures of comparable area and random-partitioning strategies. We also presented extensions of our technique to handle context switching between different application programs, as well as a generalized hierarchical memory architecture.

Our data-partitioning algorithm has been incorporated into a memory exploration strategy for determining the best on-chip memory configuration for a given application [Panda 1998]. Data partitioning into on-chip and off-chip memories is combined with a cache performance estimation technique to evaluate candidate divisions of a given on-chip memory space into data cache and Scratch-Pad memory.

REFERENCES

AHMAD, I. AND CHEN, C. Y. R. 1991. Post-processor for data path synthesis using multiport memories. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design* (ICCAD '91, Santa Clara, CA, Nov. 11-14, 1991), IEEE Computer Society Press, Los Alamitos, CA, 276–279.

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

BAKSHI, S. AND GAJSKI, D. D. 1995. A memory selection algorithm for high-performance pipelines. In *Proceedings of the European Conference EURO-DAC '95 with EURO-VHDL '95 on Design Automation* (Brighton, UK, Sept. 18–22), G. Musgrave, Ed. IEEE Computer Society Press, Los Alamitos, CA, 124–129.

BALAKRISHNAN, M., BANERJI, D. K., MAJUMDAR, A. K., LINDERS, J. G., AND MAJITHIA, J. C. 1990. Allocation of multiport memories in data path synthesis. *IEEE Trans. Comput.-Aided Des. 7*, 4 (Apr. 1990), 536–540.

BALASA, F., CATTHOOR, F., AND DE MAN, H. 1995. Background memory area estimation for multidimensional signal processing systems. *IEEE Trans. Very Large Scale Integr. Syst. 3*, 2 (June 1995), 157–172.

CATTHOOR, F. AND SVENSSON, L. 1993. *Application-Driven Architecture Synthesis*. Kluwer Academic Publishers, Hingham, MA.

GAJSKI, D. D., DUTT, N. D., WU, A. C.-H., AND LIN, S. Y.-L. 1992. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Hingham, MA.

LE GALL, D. 1991. MPEG: a video compression standard for multimedia applications. *Commun. ACM 34*, 4 (Apr. 1991), 46–58.

GAREY, M. AND JOHNSON, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., New York, NY.

JHA, P. K. AND DUTT, N. D. 2000. High-level library mapping for memories. *ACM Trans. Des. Autom. Electron. Syst. 5*, 3 (July), 566–603.

KARCHMER, D. AND ROSE, J. 1994. Definition and solution of the memory packing problem for field-programmable systems. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design* (Nov. 1994), 20–26.

KIM, T. AND LIU, C. L. 1993. Utilization of multiport memories in data path synthesis. In *Proceedings of the 30th ACM/IEEE International Conference on Design Automation* (DAC '93, Dallas, TX, June 14–18), A. E. Dunlop, Ed. ACM Press, New York, NY, 298–302.

LAM, M., ROTHBERG, E., AND WOLF, M. 1991. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-IV, Santa Clara, CA, Apr. 8–11), D. A. Patterson, Ed. ACM Press, New York, NY, 63–74.

LIAO, S., DEVADAS, S., KEUTZER, K., TJIANG, S., AND WANG, A. 1995. Code optimization techniques for embedded DSP microprocessors. In *Proceedings of the 32nd ACM/IEEE Conference on Design Automation* (DAC '95, San Francisco, CA, June 12–16, 1995), B. T. Preas, Ed. ACM Press, New York, NY, 599–604.

LIAO, S., DEVADAS, S., KEUTZER, K., TJIANG, S., AND WANG, A. 1995. Storage assignment to decrease code size. In *Proceedings of the Conference on Programming Language Design and Implementation* (SIGPLAN '95, La Jolla, CA, June 18–21), D. W. Wall, Ed. ACM Press, New York, NY, 186–195.

LIPPENS, P. E. R., VAN MEERBERGEN, J. L., VERHAEGH, W. F. J., AND VAN DER WERF, A. 1993. Allocation of multiport memories for hierarchical data stream. In *Proceedings of the International Conference on Computer-Aided Design* (ICCAD '93, Santa Clara, CA, Nov. 7–11), M. Lightner and J. A. G. Jess, Eds. IEEE Computer Society Press, Los Alamitos, CA, 728–735.

LSI LOGIC CORPORATION. 1992. CW33000 MIPS Embedded Processor User's Manual. VLSI Technologies, Inc..

MARGOLIN, B. 1997. Embedded systems to benefit from advances in dram technology. *Comput. Des.*, 76–86.

MARWEDEL, P. AND GOOSENS, J., Eds. 1995. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Hingham, MA.

PANDA, P. R. 1998. Memory optimizations and exploration for embedded systems. Ph.D. Dissertation. University of California at Irvine, Irvine, CA.

PANDA, P. R. AND DUTT, N. D. 1995. 1995 high level synthesis design repository. In *Proceedings of the Eighth International Symposium on System Synthesis* (Cannes, France, Sept. 13–15, 1995), P. G. Paulin and F. Mavaddat, Eds. ACM Press, New York, NY, 170–174.

PANDA, P. R., DUTT, N. D., AND NICOLAU, A. 1996. Memory organization for improved data cache performance in embedded processors. In *Proceedings of the ACM/IEEE International Symposium on System Synthesis* (Nov. 1996), ACM Press, New York, NY, 90–95.

PATTERSON, D. A. AND HENNESSY, J. L. 1994. *Computer Organization & Design—The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

PRESS, W. H., FLANNERY, B. P., TEUKOLSKY, S. A., AND VETTERLING, W. T. 1988. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY.

RAMACHANDRAN, L., GAJSKI, D., AND CHAIYAKUL, V. 1994. An algorithm for array variable clustering. In *Proceedings of the European Conference on Design Automation* (Feb. 1994),

RAWAT, J. 1993. Static analysis of cache performance for real-time programming. Master's Thesis. Iowa State Univ., Ames, IA.

SAGHIR, M. A. R., CHOW, P., AND LEE, C. G. 1996. Exploiting dual data-memory banks in digital signal processors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-VII, Cambridge, MA, Oct. 1–5, 1996), B. Dally and S. Eggets, Eds. ACM Press, New York, NY, 234–243.

SCHMIT, H. AND THOMAS, D. E. 1995. Address generation for memories containing multiple arrays. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD-95, San Jose, CA, Nov. 5–9), R. Rudell, Ed. IEEE Computer Society Press, Los Alamitos, CA, 510–514.

STOK, L. AND JESS, J. A. G. 1992. Foreground memory management in data path synthesis. *Int. J. Circuits Theor. Appl. 20*, 3, 235–255.

SUDARSANAM, A. AND MALIK, S. 1995. Memory bank and register allocation in software synthesis for ASIPs. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design* (ICCAD-95, San Jose, CA, Nov. 5–9), R. Rudell, Ed. IEEE Computer Society Press, Los Alamitos, CA, 388–392.

TOMIYAMA, H. AND YASUURA, H. 1996. Optimal code placement of embedded software for instruction caches. In *Proceedings of the European Conference on Design and Test* (Paris, France, Mar. 1996), 96–101.

TOMIYAMA, H. AND YASUURA, H. 1996. Size-constrained code placement for cache miss rate reduction. In *Proceedings of the ACM/IEEE International Symposium on System Synthesis* (Nov. 1996), ACM Press, New York, NY, 96–101.

TSENG, C. AND SIEWIOREK, D. P. 1986. Automated synthesis of data paths in digital systems. *IEEE Trans. Comput.-Aided Des. 5*, 3 (July 1986), 379–395.

TURLEY, J. L. 1994. New processor families join embedded fray. *Microprocessor Report 8*, 17 (Dec.), 1–8.

VANHOOF, J., BOLSENS, I., AND MAN, H. D. 1991. Compiling multi-dimensional data streams into distributed DSP ASIC memory. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design* (ICCAD '91, Santa Clara, CA, Nov. 11-14, 1991), IEEE Computer Society Press, Los Alamitos, CA, 272–275.

VERBAUWHEDE, I. M., SCHEERS, C. J., AND RABAEY, J. M. 1994. Memory estimation for high level synthesis. In *Proceedings of the 31st Annual Conference on Design Automation* (DAC '94, San Diego, CA, June 6–10, 1994), M. Lorenzetti, Ed. ACM Press, New York, NY, 143–148.

WILSON, R. 1997. Graphics IC vendors take a shot at embedded DRAM. *Elec. Eng. Times 938* (Jan.), 41–57.

WUYTACK, S., CATTHOOR, F., DE JONG, G., LIN, G. B., AND MAN, H. D. 1996. Flow graph balancing for minimizing the required memory bandwidth. In *Proceedings of the ACM/IEEE International Symposium on System Synthesis* (Nov. 1996), ACM Press, New York, NY, 127–132.

Received: May 1997