On-chip Scratchpad Memory Size Prediction and Allocation for Multiprocess Embedded Applications

Sandra Irobi, Ben Juurlink
{S.I Irobi and B.H.H Juurlink}@ewi.tudelft.nl
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
2628 CD Delft, Netherlands

Abstract—

Because on-chip memory is one of the most expensive computer system resources, embedded system designers prefer to use as little memory as possible. To guarantee real-time performance however, a certain amount of on-chip memory is required since, usually, the execution time of a task decreases as a function of the amount of on-chip memory it is allocated. Due to the need to ensure real-time performance and to avoid run-time surprises, embedded system designers tend to over-budget the size of memory allocated to each task, thereby increasing cost. In this paper we focus on the following research question: given an application consisting of n tasks, their execution times as a function of the on-chip memory size, and their periods, what is the minimum amount of memory required to ensure that all tasks are composed predictably? We considered three scratchpad sharing strategies: partitioned (each task is allocated a private partition), shared (all tasks share the scratchpad) and hybrid (sometasks have a private partition some share a partition). For the partitioned strategy, we present a polynomial-time algorithm that guarantees timeliness with the least memory size possible. For the hybrid strategy we present an exhaustive search and evaluate a heuristic approach using synthetic benchmarks. Results show that the heuristic that assigns high-frequency tasks to static regions and low-frequency tasks to the shared region so as to minimize context switching overhead, does not necessarily yield the least memory solution.

Keywords: Scratchpad memory, predictability, real-time embedded systems

I. Introduction

For the on-chip memory, caches and scratchpads may be used. A major draw-back of the use of caches for real-time embedded applications is the possibility that different tasks scheduled to run on the same processor demand different blocks of data, thereby thrashing one another's data, which poses a concern for timeliness. On the other hand, scratchpads memories are software controlled, which means that they secure predictability at runtime. It is also estimated that an energy reduction of 40% and an average area-time reduction of 46% can be achieved by using scratchpads instead of caches[1]. However, scratchpads are less flexible and require some software support for utilization. Nevertheless, because scratchpads are much more predictable, our work is focused on scratchpad memories.

In a nutshell, the main challenge addressed here is that given a set of tasks, their execution time as a function of the allocated memory size and their periods, can we accurately predict the least amount of on-chip memory needed for the tasks to execute in real-time? Also, for the given set of tasks or application, what particular partitioning strategy (partitioned, shared or hybrid) yields the minimal memory size. Our aim is to gainfully reduce area and hence, its associated costs.

Our contributions in this paper include: First, a polynomial-time algorithm for the partitioned strategy is presented. Second, an exhaustive search and a polynomial-time heuristic for the hybrid strategy are presented. The results obtained using synthetic benchmarks show that the heuristic that assigns high-frequency tasks to static regions and low-frequency tasks to the shared region so as to minimize context switching overhead, does not necessarily yield the least memory solution.

This paper is organized as follows: in Section 2, we discuss some related works and in Section 3, we present our partition strategies as well as our prediction algorithm and illustrate it with an example. Section 4 will feature our experimental evaluations with the results. We conclude in Section 6 and outline future challenges in this field.

II. Related Work

The left edge algorithm[4] where scalar variables are allocated to registers is an early method that has been used for memory estimation previously. Grun

et al. in [3], discussed memory size estimation for multimedia applications for algorithmic specifications that contain multidimensional arrays and parallel construct with both instruction level and coarse grain parallelism. These works on memory estimation were mainly about estimating the number of memory locations necessary to satisfy storage requirements of a system. In contrast to our work, the above estimation approaches specifically do not target the scratchpad on-chip memory system size, on which our work is based.

Molnos et al.[6] investigated compositional cache partitioning method for multimedia communicating tasks, where they propose a cache allocation strategy, which assigns sets of the unified cache exclusively to tasks and to communication buffers. Molnos' work focused on retaining the L1 cache and statically partitioning the L2 cache in order to enhance flexibility. Our work differs from theirs due to their focus on the off-chip L2 cache whereas we are focused on the on-chip scratchpad, an equivalent of the L1 cache.

Very little has been done in the area of scratchpad memory allocation for multiprocess applications[2], [7]. In [7] methods are described aimed at sharing the scratchpad memory targeted towards multiprocess embedded systems. Their strategy assigns both code and data elements to the scratchpad but with a focus on reducing total energy consumption. They present a saving, non-saving and hybrid allocation strategies, which when employed, yield different energy reduction levels for the target application. However, while their focus is on reduction in energy consumption, we are more concerned with reducing to the barest minimum the total size (area) of the on-chip scratchpad memory, which invariably will reduce cost, while maintaining predictability.

III. On-chip scratchpad partitions

There are three strategies for assigning tasks to the on-chip memory scratchpad. The strategies are shown in [figure 1].

The Private Partition: In this scenario, the scratchpad is fully partitioned into n single (per task) regions and each task is allocated a region. Each task remains in its assigned partition until the application or entire task set is terminated (see figure 1). Therefore the partition is exclusive to the assigned task. The partitions do not have to be of equal sizes. The total memory size in this strategy is given by the summation of the individual partition sizes.

Shared: Here, all tasks share the scratchpad. This

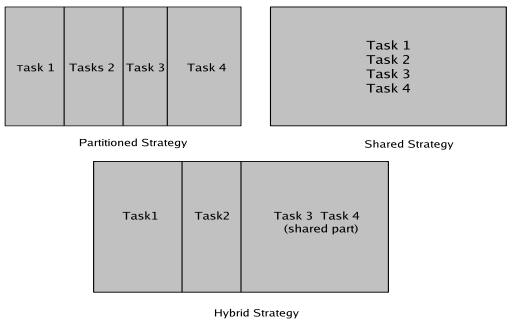

Fig. 1. Scratchpad Partitioning Methods

implies that the contents of the scratchpad should be loaded, saved and/or restored at each context switch. The cost of each context switch is taken into consideration in the algorithm. The total memory size for this strategy is the memory size requested by the highest demanding task. The implication of this is discussed further in Section 3.3.

Hybrid: The hybrid method yields a mixture where some tasks are allocated to a partitioned area and others to a shared region. The total memory size in this strategy is the sum of the memory sizes required by the shared partition and the sizes of the private partitions.

A. Algorithms

We present three algorithms that given the tasks' time consumption $t_i(m)$, as a function of their memory sizes m, and their periods, P_i, determines the minimum scratchpad size and its corresponding partition required to ensure predictability. The first algorithm returns the smallest memory requirement when each task for the partitioning strategy is allocated its own private area (as described in Section 3). The second algorithm is a hybrid that yields the optimal solution, which could be partitioned, shared or both, while the third is a heuristic, which assigns low frequency tasks to the shared region and the high frequency ones to the partitioned region.

Before we further discuss the algorithms, we shall consider some basic schedulability criteria[5].

Schedulability Test 1: A set of tasks $T_1, T_2, \ldots, T_n$ with execution times $t_1, t_2 \ldots t_n$ and periods $p_1, p_2 \ldots p_n$ can be scheduled in real-time if and only if

$$\sum_{i=1}^{n} U_i \leq 1$$

where $U_i = t_i/p_i$.

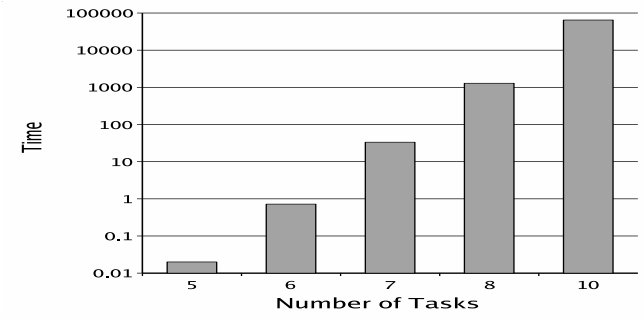

Fig. 2. Plot of Time Vs. Number of Tasks

A full exhaustive search will need $\Theta(M^n)$ time, since there are m possible partition sizes and n tasks. This is as shown in Figure 2. For the y-axis, a logarithmic scale has been used.

In our case, the execution time, $t_i(m)$, is a function of memory size m and so the utilization $U_i(m)$ also. Let m be the number of possible partition sizes. We do not assume that the scratchpad is byte-partitionable. So if the scratchpad could be partitioned into chunks of 1KB, and its total size is 16KB, m is = 17.

The algorithm for the partitioned strategy yields the minimum memory size that would be used to predictably schedule a given task set when each tasks is assigned to a single, private partition. It computes all possible task combinations for all given memory sizes, and selects and returns the least in memory size, and indicating the size of memory each assigned task will use.

The algorithm works as follows.

Let $m_0, m_1, \ldots, m_{\max}$ be the possible scratchpad partition sizes. For example, if the scratchpad can be partitioned into chunks of 1KB and its total size is 16KB, then m_0 = 0KB, m_1 = 1KB, etc., and $m_{\max}$ = 16KB. Furthermore, $u_i(m_j) = t_i(m_j)/p_i$ is the utilization of task T_i when it is allocated m_j bytes of scratchpad. The algorithm computes, for each possible scratchpad size m_j, the minimal utilization. It does this by maintaining a table *util_table*, where *util_table*$[m_j]$ is the minimal utilization for scratchpad size m_j.

Initially, *util_table*$[m_j] = u_1(m_j) = t_1(m_j)/p_1$, that is, the utilization of tasks T_1. Now it adds the tasks one by one. For each possible scratchpad size m_j, it tries every possible combination of allocating x bytes to the new task and $m_j - x$ to the already processed tasks and finds the minimal utilization.

Note that the minimal utilization of the already processed tasks when allocated a scratchpad of size $m_j - x$ is given by *util_table*$[m_j - x]$. Furthermore, the utilization of the new task T_k when given a private scratchpad partition of size x is given by $u_k(x)$. Once the last process has been added, the minimal scratchpad size required to schedule all processes in real-time is given by the least i such that *util_table*$[m_i] \leq 1$.

A more formal description of the algorithm is shown (Algorithm 1). The algorithm takes $\Theta(nM2)$ time, where as before n is the number of tasks and M the number of possible scratchpad partition sizes, since each invocation of the procedure *addTask* takes $\Theta(M2)$ time and it is called n times.

addtask:

Input parameters: *util_table* of tasks processed; i: number of tasks to be added. Results: New *util_table* with the new task added.

```
1:  min = infinity
2:  for m = m_1(= 0), m_2 ... m_max do
3:    for x = m_1, m_2 ... m do
4:      if util_table[m − x] + u_i(x) < min then
5:        min = util_table[m − x] + u_i(x)
6:        util_table[m] = min
7:      end if
8:      Return util_table
9:    end for
10: end for
```

Algorithm 1:

```
1:  for m = m_1, m_2 ... m_max do
2:    util_table[m] = u_1(m) = t_1(m)/p_1(m)
3:  end for
4:  for i = 2 ... m do
5:    util_table = addTask(util_table, i)
6:    for m ≤ m_max do
7:      m = m_1, m_2 ... m_max
8:      if util_table[m] ≤ 1 then
9:        Return m
10:     end if
11:   end for
12: end for
```

A.1 An Example

We shall now illustrate the algorithm for the partitioned strategy using the example task set depicted in Table I. We assume that the tasks cannot employ efficiently more than 4KB of scratchpad. In other words, $t_i(m) = t_i(4\text{KB})$ for all $m \geq 4\text{KB}$.

Table II shows the values of the elements of the utilization table after adding each task. Initially, *util_table*$[m_j]$ is the utilization of task T_1 $(u_1(m_j))$ when it is allocated a scratchpad partition of size m_j. For example, when it is given a partition of 1KB its

Memory size	T_1	T_2	T_3	T_4
0KB	24	20	14	8
1KB	12	10	7	4
2KB	8	6	4	3
3KB	7	5	3	2
4KB	5	4	2	1
Period	20	24	10	8

TABLE I

EXECUTION TIMES OF THE TASKS AS A FUNCTION OF THE SCRATCHPAD PARTITION SIZE AND THEIR PERIODS

utilization is $t_1(1KB)/p_1 = 12/20 = 0.6$. The next column shows the values of the elements of the utilization table after task T_2 has been added. It already shows that using a total scratchpad size of 2KB does not yield a schedulable solution. For a total scratchpad size of 2KB the possible partitions are $(0, 2)$ (allocate 0KB to T_1 and 2KB to T_2), $(1, 1)$, and $(2, 0)$.

The first one can be discarded because the execution time of T_1 when it is allocated a partition of 0KB is larger than its period. The second partition does not yield a schedulable solution because $t_1(1\text{KB})/p_1 + t_2(1\text{KB})/p_2 = 12/20 + 10/24 = ... > 1$. Similarly, the third one can also be discarded. The last column shows that the minimal scratchpad size is 12KB, since it is the least memory size that has a utilization of less than or equal to 1. The partition that achieves this is $(2, 2, 4, 4)$. We finally remark that the algorithm does not have to consider partitions corresponding to infeasible solutions. However, this will not affect the worst-case complexity of our algorithm.

A cross section of the output from the algorithm is shown in Table III

B. Hybrid Algorithm

Our algorithm for the hybrid aims at obtaining the optimal solution for the task assignment strategies and least memory size for the given set of tasks. It computes the tasks' utilizations for all memory sizes, and returns the least value of memory from the list of the schedulable options with utilization less than 1. It first assigns the task, T_1, to the partitioned part of the scratchpad, $(solS[T] == 0)$ - where 0 denotes the partitioned region with an assigned memory size M_j bytes. Let p denote tasks in the private partitions, it computes the utilization of the tasks here using $U_p = t_p/P_p$. The total memory size of all tasks assigned to this partition is given by the sum of the sizes of the single partitions.

Memory size	Tasks' Utilization			
	T_1	T_2added	T_3added	T_4added
0KB	1.20	2.03	3.43	4.43
1KB	0.6	1.43	2.73	3.73
2KB	0.4	1.02	2.13	3.13
3KB	0.35	0.82	1.72	2.63
4KB	0.25	0.65	1.42	2.22
5KB	0.25	0.60	1.22	1.92
6KB	0.25	0.50	1.05	1.72
7KB	0.25	0.46	0.95	1.55
8KB	0.25	0.42	0.85	1.43
9KB	0.25	0.42	0.80	1.30
10KB	0.25	0.42	0.70	1.18
11KB	0.25	0.42	0.66	1.08
12KB	0.25	0.42	0.62	0.97
13KB	0.25	0.42	0.62	0.92
14KB	0.25	0.42	0.62	0.82
15KB	0.25	0.42	0.62	0.78

TABLE II

TASKS' UTILIZATION FOR THE PARTITIONED EXAMPLE

Min Mem size	T1	T2	T3	T4
12KB	2	2	4	4
13KB	3	2	4	4
14KB	4	2	4	4
15KB	4	3	4	4

TABLE III

CROSS SECTION OF MEMORY SIZES FOR THE PARTITIONED EXAMPLE

However, for the tasks assigned to the shared part of the scratchpad, the algorithm defines the size of the shared partition, max. It does this by inspecting all tasks assigned to the shared part and adopting the maximum size of memory allocated to any of the component tasks. For instance if tasks T_2 and T_3 are assigned to the shared part with memory allocations 3KB and 4KB respectively, the memory size that will be adopted for the shared part is 4KB, being the highest assigned memory value. The context switch penalty per scratchpad size for the value of the maximum scratchpad size for the shared partition, max is added to each task's execution time (for every task in the shared partition only). Hence, if s denotes tasks in the shared region, then computation for a task utilization is given by:

$$(t_s + cs * max)/p_s \quad (1)$$

For a complete round of computation for a given memory size, we sum up the total utilization of tasks in both the partitioned and shared parts and return the total value. The total memory size for the combinations is given by the summation of the singly partitioned parts and the max value of the shared partition. To yield an acceptable solution, the summation of the two parts must return utilization less or equal to 1. This is given by:

$$\sum_{i \in p} t_p/P_p + \sum_{i \in s}((t_s + (cs_s * max))/p_s) \leq 1 \quad (2)$$

Each schedulable solution is saved. The algorithm then assign tasks to the shared part of the scratchpad, $(solS[T] == 1)$ - where 1 denotes the shared case. It assigns M_j bytes to task T, for all the possible sizes of memory M_j - $(solM[T] = j)$. When all the task set has been considered, the algorithm returns the least memory size of from the saved solution

A formal description of the algorithm is shown in Figure below. U_s means the utilization for a task T in the shared part.

Algorithm 2

1: **for** $i = 0 \ldots tasksize$ **do**
2: **if** $solS[T] == 0$ **then**
3: $solM[T] = j$
4: $U_p = \sum t_p/P_p$
5: **else**
6: $max = memsize[solM[T]]$
7: $U_s = \sum[t_s + cs * max]/P_s$
8: $U_t = U_p + U_s$
9: **end if**
10: **end for**
11: **if** $U_t \leq 1$ **then**
12: **for** $i = 0 \ldots tasksize$ **do**
13: $savesolS, sol\_min\_S$
14: $savesolM, sol\_min\_M$
15: **if** $sol[new] < sol[old]$ **then**
16: $sol[min] = new\_sol$
17: **end if**
18: **end for**
19: **else**
20: **for** $solS[T] == 1$ **do**
21: $solS[T] = i$
22: **for** $j = 0 \ldots memsize$ **do**
23: $solM[T] = j$
24: $Returnmemsize[sol\_min[i]]$
25: **end for**
26: **end for**
27: **end if**

C. The Heuristic Algorithm

The previous algorithm computes the optimal solution but requires exponential time. In this Section we therefore present a heuristic that separates the task set into two parts based on the task periods, P_i. It determines a threshold boundary for this separation by dividing the highest period in the task set by two (fractional parts not considered). It then categorizes the high frequency tasks (low periods) as those below this threshold value while the low frequency (high periods) tasks are the ones above the threshold period. The heuristic assigns high frequency tasks to the partitioned part of the scratchpad and the low frequency tasks are allocated to the shared part. The idea behind this heuristic is that high frequency tasks need to be scheduled more often, and each time, the context switching overhead is incurred.

The algorithm then computes the possible combinations of the tasks in this strategy for all given memory sizes. For the tasks allocated to the partitioned part, the utilization computations are done simply using:

$$\sum_{i=1}^{n} t_i/p_i \leq 1$$

while for those assigned to the shared part of the scratchpad the computations are done by invoking equation (2), which takes the context switching times into consideration. The sum of the memory sizes needed in each of the parts yields the total needed memory for that round of combination, same as computed in equation 3. The algorithm stores every feasible solution and when all the possible cases have been considered, returns the least memory size from this archive.

The heuristic algorithm assumes the same structure as the partitioned algorithm except for the initial separation of the task set into high and low frequency tasks.

IV. Experiments and Results

A. Experiments

In our experiments, we evaluate the partitioned, hybrid and heuristic algorithms. Eight different task sets were used and Table IV shows a sample of periods and execution times for one of the task sets. We run the experiments using a context switch cost factor of 0.3

Number of Tasks	Hybrid(KB)	Heuristic(KB)	Partitioned(KB)
4	8	11	12
4	3	7	6
5	9	17	20
5	10	14	20
6	12	20	28
6	27	27	29
7	36	34	42
7	25	26	37

TABLE V
MINIMUM MEMORY SIZES RESULTS FOR THE TASK SETS

Tasks	1KB	2KB	3KB	4KB	5KB	Period
T1	8	7	6	3	2	26
T2	8	6	5	4	1	35
T3	9	7	6	4	2	10
T4	10	9	8	6	4	25
T5	9	8	6	4	3	14

TABLE IV
A CROSS SECTION OF EXPERIMENTAL TASK SET PARAMETERS

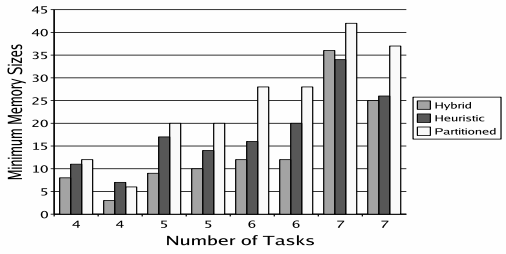

Fig. 3. Results 1 for optimal memory sizes

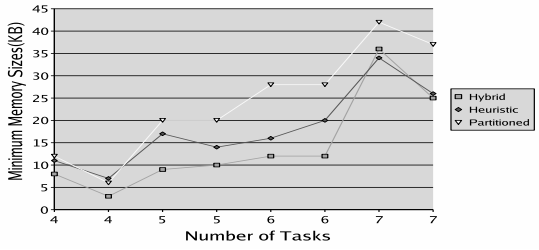

Fig. 4. Results 2 for optimal memory sizes

per KB of scratchpad as the penalty for save and restore operations for tasks in the shared part. The value of the context switch varies with the different sizes of scratchpad.

The real-time tasks used are rate monotonic, preemptable, and statically scheduled. Our tasks sets were obtained through a random number generator program for values between 1, 10 and 12 for the number of memory sizes needed. The generation of the random numbers was in such a way that as the scratchpad memory size increases, the generated execution time of the task decreases. In the same way, the tasks' periods are generated but are not affected by the scratchpad size. The memory sizes used for each case varied with the number of tasks.

B. Results

The results obtained from the experiments are as shown in Table V. The figures show that the optimal (minimum) memory size in at least 75% of the cases were given by the hybrid strategy. The heuristic algorithm only yielded an optimal solution in less than 15% of all the explored cases. Among the 75% it is interesting to note the range of deviation from optimality of both the heuristic and partitioned cases from the hybrid. For instance in the second instance for the application with five tasks, the hybrid against the heuristics, saves a total of 8KB (from 9KB to 17KB) and also saves 11KB as against the partitioned case (from 9KB to 20KB). This pattern is also repeated in the instance of the application with six tasks. Again the hybrid finishes with a minimum memory size of 12KB, while the heuristic and partitioned cases trail behind with 20KB and 28KB respectively, which saves 8KB and 16KB in each case.

A plot of the graph of this recorded pattern, as shown in Figures 3 and 4, captures the results while the plot in Figure 5 expresses the range of deviation of the heuristic and partitioned case algorithms from the optimal solution yielded by the hybrid. Thus, the

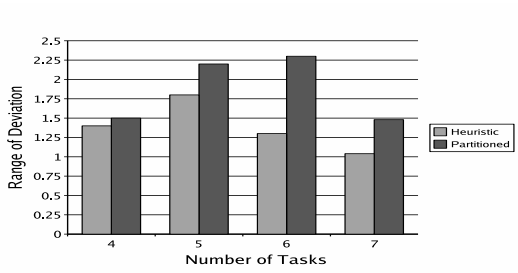

Fig. 5. Range of deviation from optimal solution

result of the hybrid strategy is seen to give the most benefit in terms of area cost for the on-chip scratchpad memory. The differences in sizes as seen from the deviation graph, that is, the actual deviation from optimality in terms of size as exhibited by the heuristic and private partitioned strategies, can be deployed as a guide in estimating area-cost trade-offs by systems' designers.

V. Conclusion

In this paper, we have evaluated three algorithms - the private partition, hybrid and heuristic strategies. The hybrid proves to be the best option for on-chip memory area reduction over the heuristic method that assigns high frequency tasks to an all partitioned part and low frequency ones to the shared part as well as against the private partitioned case. 75% of all the tested cases resulted in the hybrid strategy yielding the optimal (minimum) memory sizes, while the heuristic strategy led in less than 15% of all cases. We have also provided an estimate of an area-size deviation analyses for the partitioned, hybrid and heuristic strategies that could guide embedded real-time systems' designers in making informed choices. Our future work will seek to extend the allocation algorithms for dynamically scheduled tasks while still preserving predictability.

References

[1] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. *In Proceedings of International Symposium on Hardware/Software Codesign (CODES)*, 2002.

[2] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and M. Mendias. An integrated hardware/software approach for run-time scratchpad management. *In Proceedings of Design Automation Conference, San Diego California, USA*, 2004.

[3] Peter Grun, Florin Balasa, and Nikil Dutt. Memory size estimation for multimedia applications. *In Proceedings of International Conference on Hardware Software Codesign Proceedings of the 6th Intenational workshop*, pages 145–149, 1998.

[4] F. Kurdahi and A. Parker. Real: A program for register allocation. *Proc. 24th ACM/IEEE Design Automation Conference*, pages 210–215, 1987.

[5] Jane W.S Liu. *Real-Time Systems.* Prentise Hall Upper Saddle River, New Jersey, 2000.

[6] A. Molnos, M.J.M Heijligers, S.D. Cotofana, and J.T.L. Van Eijndhoven. Compositional memory system for multimedia communicating tasks. *Design Automation and Test in Europe Conference*, 2:932–937, 2005.

[7] M. Verma, Klauss Petzold, Lars Wehmeyer, Heiko Falk, and Peter Marwedel. Scratchpad sharing strategies for multiprocess embedded systems: A first approach. *In Proceedings of IEEE 3rd Workshop on Embedded System for Real-Time Multimedia Jersey City, USA.*, 2005.