Code placement in Hardware Software Co synthesis to improve performance and reduce cost

Sri Parameswaran
Department of Computer Science and Electrical Engineering
The University of Queensland
Qld 4072, Australia
sridevan@csee.uq.edu.au

Abstract

This paper introduces an algorithm for code placement in cache, and maps it to memory using a second algorithm. The target architecture is a multiprocessor system with 1st level cache and a common main memory. These algorithms guarantee that as many instruction codewords as possible of the high priority tasks remain in cache all of the time so that other tasks do not overwrite them. This method improves the overall performance, and might result in cheaper systems if more powerful processors are not needed. Amount of memory increase necessary to facilitate this scheme is in the order of 13%. The average percentage of highest priority tasks always in memory can vary from 3% to 100% depending upon how many tasks (and their sizes) are allocated to each processor.

1. Introduction

As very large scale integration (VLSI) systems become larger and larger, and feature sizes become smaller, chips become more complex. This complexity leads to extremely long development times using traditional design techniques. One of the method by which designers are overcoming this problem is to use pre-deigned, pre-verified cores. Several cores (also sometimes referred to as processing elements or PEs) are connected together to execute a single application. An example of such an application can be a video processing decoder or a sophisticated aircraft engine management system. Hardware Software Co-Synthesis methodologies attempt to automate the system development. Several algorithms have been developed for Hardware Software Co-Synthesis.

Caches are small high-speed memories placed between the processor and the main memory. They function as the main memory but work at close to the speed of the processor. If instructions are in the cache then the task can be executed faster than if it were to bring it from the memory.

In HW/SW Co-design environments, we use several processors, which have to meet strict deadlines. Failure to meet these deadlines can have severe consequences. The usual pattern has been to change the processor and meet the deadlines with more expensive processors. There has been considerable effort to also look at cache by Li and Wolf [1], one by reserving some cache for high priority tasks and the other by doubling caches until dead lines are met.

In this work we do not reserve cache for high priority tasks, but we look at methods to not overwrite as much as possible of the highest priority tasks as possible, so that they do not have to be brought into the cache again.

1.1 Related Work

Most of the early work in Hardware Software Co-Synthesis was constrained to a single processor and ASIC. Work by Henkel et. al [2, 3], Gupta et. al [4, 5] and Parameswaran et al [6, 7]. Prakash and Parker published a mixed integer Linear Programming (MILP) approach [8] to synthesize multiple processor architectures for a specific application. This work was enhanced by the authors in [9] to include memory costs into the final implementation. Due to the use of MILP, the scheme proposed by Prakash and Parker had an exponential computation time. Heuristic solutions [10] and evolutionary algorithms [11] have been proposed to reduce the total computation time and to increase the size of problems, which can be dealt with.

Memory optimisations for Hardware Software co-design have been researched extensively in the recent past. Danckert et. al [12] looks at mapping data to memory for efficient processing of tasks. Verkest et. al [13] have created a tool named *Matisse* which looks at several of the memory management problems before partitioning occurs. Wuytack et. al [14] presented an approach to reduce memory bandwidth. Panda et. al. [15-20] has comprehensively researched the topic of data memory in caches and main memory.

Instruction placement in cache for multi-processor synthesis was published by Li and Wolf in [1]. The system divides the tasks into high-priority and low-priority tasks. The work discussed in this paper allocates approximately half the cache for high priority tasks and in the other half allocates the rest of the tasks, to reduce the total number of compulsory misses in a similar manner to the work described in [21, 22].

The allocation of these tasks to memory was left to the compiler for completion. Li and Wolf found the above scheme to be inefficient and used an arbitrary mapping scheme in [23, 24] where the tasks could map to any position in cache. To improve performance of the overall system, they doubled their cache memory until the system met deadlines (if it could not then they moved on to a different partition).

1.2 Relevance of this work

In this paper we describe a method by which high-priority tasks try to capture as much of the cache as possible without reserving half the cache for them. This work is a part of a whole Hardware Software Co-design scheduling portioning allocation system. Once tasks have been scheduled to separate processors with specific cache memory, this system then looks at task allocation in caches and then maps these tasks to the memory so that maximum performance can be obtained with the given resources. The system also minimizes the total memory used by the system.

1.3 Assumptions

For the sake of simplicity, the following assumptions are made about the tasks and the cache. The assumptions are somewhat similar to those made by Li and Wolf in [24].

Assumption 1: The size of the task is no bigger than the size of the cache. This assumption is quite valid in embedded systems where the tasks are usually small enough to fit into small cache sizes. If the task is too large for the cache it is possible to break up the task into smaller granules such that each granule will fit into the cache. Due to this assumption, we will only have compulsory misses, which does not change with the size of cache.

Assumption 2: Only Level-1 caches are available for use. Once again in an embedded system, where frequently there is no cache at all, it is unlikely that more than a single level of cache is going to be available for use.

Assumption 3: The caches are direct mapped. This assumption makes easier analysis due to the direct mapping and therefore deterministic mapping to cache from memory.

Assumption 4: The instructions of a task are allocated to a continuous region of memory. This states that a task mapped to cache will be also in a continuous region in that cache (except when it overflows the cache, which then will continue from the top of the cache). If the memory is going to be in two parts then they will be considered as two separate tasks.

Assumption 5: All addressing performed within each task use relative addressing (i.e. branch) only. Tasks can be mapped to arbitrary locations within the memory map. This does not allow absolute addressing. If absolute addressing is used then instruction code cannot be loaded into any section of the code. If absolute addressing is necessary then we need to have a more sophisticated compiler is necessary.

1.4 Motivational Example

In systems where cache is present, the instruction is first brought to the cache before the instruction is executed. In Hardware Software Co-Synthesis system, a task graph is used as input, where each of the nodes is a task to be executed. The task graph is repeatedly executed with separate data sets. In general purpose computer systems tasks are compiled and stored into memory in a first in first out (FIFO) scheme, which means that when a task is mapped into the cache, we have little control as to where the task will end up in cache. However, by placing tasks carefully in memory, we could make critical task at least partially stay in cache at all time so that the instructions (after the first time when compulsory misses occur) will always be in cache at all times. For example, let us assume that the system is composed of the following architecture given in Figure 1. In this architecture, there are several processors and a single main memory unit, which contain all of the instructions and data, which are necessary for execution of tasks.

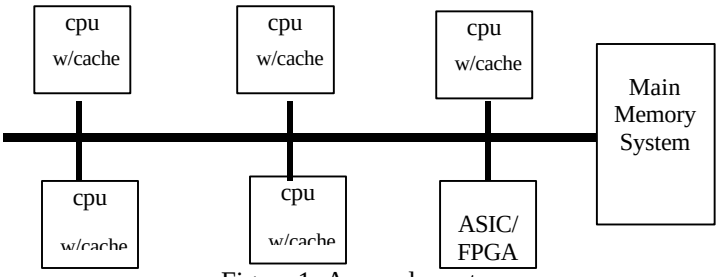

Figure 1: A sample system

Let us assume that the following tasks were compiled and placed in memory. The number of Processing Elements in this example is just two (numbered 1 and 2). Processor 1 has 5 processes allocated to it numbered from 0 – 4, and Processor 2 has 4 processes allocated to it numbered from 0-3. Let us assume that both the processors have a cache size of 1024 bytes. Then the I-cache of Processor 1 and 2 will be mapped as shown in column 5.

PE	Process no	Size	Mem. Map	Cache Map
1	0	900	0-899	0-899
2	0	800	900-1699	900-675
1	1	800	1700-2499	676-451
2	1	700	2500-3199	452-127
1	2	700	3200-3899	128-827
2	2	500	3900-4399	828-303
1	3	600	4400-4999	304-903
2	3	500	5000-5499	904-379
1	4	400	5500-5899	380-779

Table 1: An example system

As can be seen from this table, if we only look at processor 1, it can be seen that the associated cache is filled from 0-899 (task 0), then from 676-1023 and then from 0-451 (task 1). Task 2 in processor 1 will then fill 128-827. Task 3 will cover from 304 to 903 and finally task 4 will cover the cache from 380-779. The only part of the cache that does not need to be replaced again when a new task arrives will be the part of the

cache from 904 to 1023. As the number of tasks increase, this too will disappear and there will be virtually no part of the task that will remain in the cache when the task graph starts to repeat once again. Thus there are only 120 locations that will never have to be filled once again.

If another scenario is considered, where the cache is filled in the following manner. Task 0 is filled from 0-899. Task 1 is filled from 224-1023. The space from 224-923 is filled by task 2. Task 3 occupies 224-823 and finally task 4 occupies 224-623. If the cache were to be filled in such a manner, then the cache area from 0-223 and 924-1023, will never need to be replaced (totalling 323 locations). Assuming that it takes an extra 10 clock cycles to bring in data from memory to cache the total time saved will be 2030 clock cycles for just one processor. With both processors taken into account, the total savings will be 5020 clock cycles.

The negative side to this is the extra memory needed to make certain that the data maps to the particular locations that have been allocated to them. For Task 0 to map to 0-899 in cache, let us say it is in memory location 0-899. Task 1, which maps from 224-1023, will have to be in locations 1248 (1248 mod 1024 equals 224) to 2047, since that is the first address which is free where the data will map to cache at 224. In this example there is a gap in memory from locations 899 to 1248. Unless there is a later task that can utilize that space, this gap in memory will remain, which will result in wasted space. Therefore the total memory needed will be greater than for the FIFO scheme, but the scheme will result in faster execution of certain tasks resulting in an overall speedup on the task graph.

1.5 Organization of this paper

This paper is organized as follows: the first section introduced the topic and discussed the relevant works. The next section mathematically defines the problem. Section three discusses two heuristic algorithms and section four gives the method by which these algorithms were validated and then the results are presented. Section five concludes the paper.

2. Problem Statement

The problem can be mathematically stated as follows:
Let there be m processors named $P_1, P_2, P_3 \ldots P_m$, each with associated cache sizes $S_1, S_2, S_3 \ldots S_m$. The j_{th} processor has n_j tasks associated with it. The tasks have an urgency factor associated with each called $x_{1j}, x_{2j}, x_{3j} \ldots x_{nj}$ and a size associated with it called $s_{1j}, s_{2j}, s_{3j} \ldots s_{nj}$. Let $x_1 \geq x_2 \geq x_3 \ldots x_n$. That is the tasks are ordered in descending order of priority.

For a cache be tightly packed, and that as many a high priority tasks be given space in the cache which is not replaced, let task k_j satisfy, $\sum_{i=1}^{k_j} s_{ij} + \max_{k_j+1 \leq i \leq n_j} \{ s_{ij} \} \geq S_j \geq \sum_{i=1}^{k_j-1} s_{ij} + \max_{k_j \leq i \leq n_j} \{ s_{ij} \}$. Thus it can be seen that $k - 1$ tasks can fit completely in the cache above the largest of the remaining tasks. A part of the k^{th} task will also fit above the largest of the remaining task but all of the others will be below the largest of the remaining tasks. For the sake of representation we shall break task k into two tasks (thus now there are $n_j + 1$ tasks in each cache). The k^{th} task has now become k and $k+1^{th}$ task, where k is in the section, which will never be overwritten by the remaining largest task, and $k+1^{th}$ task is in the section, which will be over written by the remaining largest task.

Let t_{ij} be the starting address in cache j of task i and let e_{ij} be the ending address of task i in cache j. Therefore

$t_{1j} = 0$ and $e_{1j} = s_{1j} - 1$,

$t_{lj} = e_{lj-1} + 1$ and $e_{lj} = t_{lj} + s_{lj} - 1$ for $l = 2, \ldots, k+1_j$ and $j = 1, \ldots, m$.

$t_{lj} \geq e_{k_j j} + 1$ and $e_{lj} \leq S_j$ for $l = k_{j+2}, \ldots, n_j$ and $j = 1, \ldots, m$

Our aim is to preserve the above task allocation in cache, while reducing the overall memory consumption. In order to do that, we must satisfy the following inequalities in memory so that two separate tasks will not overlap in memory.

$t_{lj} + i_{lj}S_j + Z_j \geq e_{nm} + i_{nm}S_m + Z_m$

$t_{nj} + i_{nj}S_j + Z_j \geq e_{lm} + i_{lm}S_m + Z_m$, for $m = 1, \ldots, m$ and $j = 1, \ldots, m$.

Since the broken up task kj and k+1j must be together in memory, the tasks must also meet the condition that

$t_{kj} + i_{kj}S_j + Z_j + s_{kj} - 1 = t_{k+1j}$.

If Y is the largest memory location then,

$e_{lj} + i_{lj}S_j + Z_j \leq Y$

Our aim is to <u>*minimize Y.*</u>

2.1 Complexity of the Problem

It can be shown that a special case of the problem where, we set all Z's to 0 can be shown to be a bin-packing problem. Therefore this problem is at least as hard as the bin-packing problem which is NP-Hard. To solve this problem in reasonable run-time it is necessary to use some type of heuristics. The following section gives two separate algorithms, one algorithm to allocate tasks in cache and another to map the tasks in memory to reduce overall memory size.

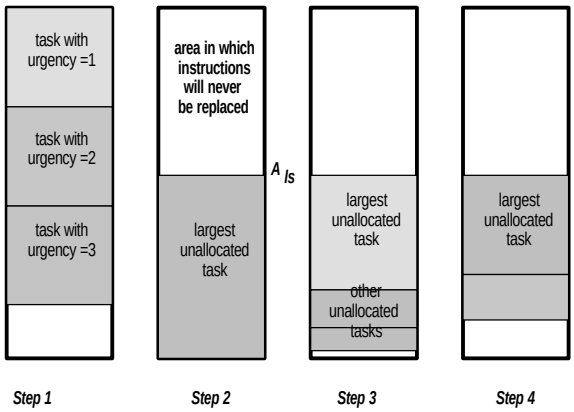

Fig 2. An Illustration to show task allocation methodology in cache

3. Algorithm for allocating tasks in Cache and Memory

3.1 Cache allocation

The methodology used for ordering tasks in cache is as follows (see Figure2). First the tasks are ordered for each of the processors in their order of urgency. Until the cache is completely filled, the tasks (only whole tasks are allowed) are allocated to the cache from the lowest address to the highest. Once this step is finished, we find the largest task from the remaining tasks. This large task is allocated to the bottom of the cache say with starting address A_{ls} and ending at the end of the cache (step 2). After this we take the next largest task and allocate it's starting address in cache to $A_{ls.}$ (step3). The ending address will be less than the final address of the cache. Thus if another unallocated task can be found which can go into the space (below the task we just allocated, and above the last cache address), we allocate that task into the available space (step 3 contd). We keep doing this until we reach the end of the cache. We take the next largest unallocated task (step 4), and start it at address A_{ls} and we repeat the process until all tasks are allocated.

Algorithm

Let us assume that the processors used in the system are p_1, p_2, p_3… p_k. The associated caches are c_1, c_2, c_3, … c_k. For each processor *r*, the tasks to be executed in that processor are as follows: T_{r1}, T_{r2}, T_{r3} …..… T_{rn} where n is the number of tasks, and the tasks are ordered in descending order of urgency. As the Urgency increases, the associated task is expected to be executed in as short a time as possible. Urgency can be defined in many differing ways. We have used the definition given in [1].

```
ForEach Processor {
      Until Cache is filled
            Allocate tasks to the cache in
            descending order of urgency;
       Reorder Unallocated tasks in order of
       size and place in list Trl  (Trl = Trl1,
       Trl2, Trl3 … Trlm, where y is the number of
       unallocated tasks for that processor and
       y <n);
       Allocate Trl1 from address Als to end of
       cache (where tr11  = sizeof (cache r)  -
       sizeof (Trl1));
       Remove Trl1 from Trl
       Repeat Until all tasks are allocated {
             Find the next largest unallocated
            Task Trlp from the list Trl;
            Allocate Trlp from address Als to
            Ale where Ale = Als + sizeof (Trlp);
            Mark  Trlp  from  list  Trl  as
            allocated;
             Move along the list Trl and place
            as many tasks as possible between
            Ale  and sizeof (cache r);
            Mark placed tasks as allocated;
      }
}
```

3.2 Memory Allocation

The memory allocation algorithm takes the placed tasks in cache and directly maps them to the memory. Thus if a task is mapped to location from t_x to e_x in cache of processor *r*, then the task can be placed in memory in any one of the address ranges from addresses $t_x + i^*$ *sizeof* (cache *r*) to $e_x + i *$ *sizeof* (cache *r*), where *i* is a positive integer . However, since tasks in cache will wrap around the cache, an offset $Z_{r,}$ can be added to each task allocated to processor *r*, and the task can be placed from memory location $t_x + Z_r + I *$ *sizeof* (cache *r*) to memory location $t_x + Z_r + i *$ *sizeof* (cache *r*). This introduction of the offset allows the reduction in size of the total memory needed for the system.

The algorithm works as follows as described here. The processors are ordered in the descending order of total task size and put in a processor list. The processor with the highest sum of all task sizes will be the first in the processor list. The processor with the lowest sum will be the last element in the processor list. The tasks allocated in the first processor are mapped to the main memory as follows: the first task allocated in cache is directly mapped to the memory (in this case, task $t_x = 0$ and $e_x =$ *sizeof* (task 1) – 1); task and upwards will be mapped by moving *i* along the non-negative integer range and trying to find an area in memory which is free from locations $t_x + i^*$ *sizeof* (cache *1*) to $e_x + i *$ *sizeof* (cache *1*). The tasks allocated to the subsequent processors are allocated thus: the largest task allocated to the processor is put in the first available memory block where the task will fit (say M_x to M_y); from this the offset Z_r is calculated, and $Z_r =$ (M_x mod *sizeof* (cache *r*)) – t_x. Using this $Z_{r,}$ all of the other tasks allocated to the processor are mapped to the memory in the order of size from largest to smallest using the mapping each task from memory location $t_x + Z_r + i^*$ *sizeof* (cache *r*) to memory location $t_x + Z_r + i *$ *sizeof* (cache *r*).

Algorithm

Let us assume that the processors used in the system are p_1, p_2, p_3… p_k. The associated caches are c_1, c_2, c_3, … c_k. For each processor *r*, the tasks to be executed in that processor are as follows: T_{r1}, T_{r2}, T_{r3} …..… T_{rn} where n is the number of tasks.

Foreach processor do

$$\sum_{n} \text{sizeof}\,(T_{rx})\ ;$$

Reorder processors from largest sum to smallest sum of total task size and name them p_a, p_b, p_c …;

```
For all tasks ordered in the cache allocation
order in pa do {
  i =  0;
  While task not allocated do {
       If  memory  locations  tx  +  i*  sizeof
       (cache a) to tx + i * sizeof (cache a)
       is free then
             Map  task  to  address  tx  +  i*
             sizeof  (cache  a)  to  tx  +  i  *
             sizeof (cache a);
```

```
        Else
                        i++;
        }
}

For all tasks ordered in descending order of
size in each subsequent processor p_b, p_c etc do
{
        Allocate largest task in the first
        available contiguous memory block  (M_x
        to M_y) which will hold the task

        Calculate Z_r =  (M_x mod sizeof (cache
        r)) – t_x, where t_x is the address in
        which the task being allocated starts in
        the cache at address 0 and r is the
        present processor under consideration;

        While task not allocated do {
                If memory locations t_x + Z_r + i*
                sizeof (cache a) to e_x +Z_r + i *
                sizeof (cache a) is free then
                        Map task to address t_x + Z_r
                        +  i* sizeof (cache a) to
                        e_x + Z_r + i * sizeof (cache
                        a);
                Else
                        i++;
        }
}
```

4.Validation of cache performance & Results.

4.1 Validation method

To validate the performance of the cache we looked at two schemes: one the scheme described in this paper and the other an arbitrary scheme, where we assumed memory was written in a first in first out manner (ie the one that was compiled first was written to the memory first and then the second and so on). In order to check the performance of the cache we wrote a task level cache simulator, which only tracks compulsory misses. Since we are only expecting to have compulsory misses, a cache simulator at this level is sufficient.

To show the power of this technique, we found it useful to randomly create cache sizes, the size of tasks and vary the number of processors and the cache sizes. We created systems with 2, 5, 10 and 20 processors and varied the number of tasks from twice the number of processors to one hundred times the number of processors. Therefore for two processors we varied the number of tasks from 20 to 200. The cache sizes were randomly created and varied from 1K to 1M. The caches were associated with processors. All task sizes were randomly created and each task was associated with a particular processor. Since there was a condition that the task size had to be smaller than the size of the cache, all tasks larger than the cache size was rejected from the tasks created randomly. The largest sized task in each processor was given the highest priority since that would give us the worst case performances.

We recorded the total number of words, which did not have to be brought into the cache for the second time. We also recorded the percentage of each of the first three highest priority tasks, which were never pre-empted. We tabulated these for the two cases: one our proposed algorithm and the other the FIFO case. We further collected the amount of memory needed for the tasks in both of the cases.

4.2 Results

The results are given for the average percentage of each of the first two highest priority tasks, which does not have to be restored to memory, when other tasks pre-empt the task. Figure 3 shows the average percentage for the first task in processors for varying number of tasks for our allocation scheme and the FIFO allocation scheme. As can be seen from the graph when there are just a few tasks for each processor, the percentages for the FIFO case is reasonably close to the figures given using our method. As the number of tasks increase, the amount of words left in the cache is significantly more in comparison to the FIFO case. Even when there is only a 10% difference this could mean 10,000 words may be in cache all the time. So for every execution of the task graph, the amount of time save would be in the order of 80,000 clock cycles (assuming that it takes 8 more clock cycles to bring a word from memory), which is quite significant.

In Figure 4 the average percentage of 2nd highest priority tasks, which is always in memory, is shown for differing processor and task numbers. This shows that the scheme works for not only the highest priority task, but it also continues to work for lower priority tasks as well (as long as it has not been overwhelmed by large tasks).

The average memory necessary for the proposed scheme and the FIFO scheme are shown in Figure 5. The extra memory required was on average 13.2 % and varied from 3% to 37% excess memory.

5. Conclusions

This paper presented a method by which to allocate tasks in cache in such a way that the total execution time could be reduced. The scheme requires the user to place the highest priority tasks in an area where the lower priority tasks will never overwrite. In order for the tasks to map to the correct position in cache, the task has to be mapped to memory in a particular way.

Acknowledgments

The author would like to thank Dr Joerg Henkel for the interesting discussions and Dr Bhaskar Sengupta for helping to formulate the mathematical description of the problem. The author is also immensely grateful for the support given by the management and members of the Computers and Communications Research Laboratories of NEC USA in Princeton, where this work was conducted while on sabbatical.

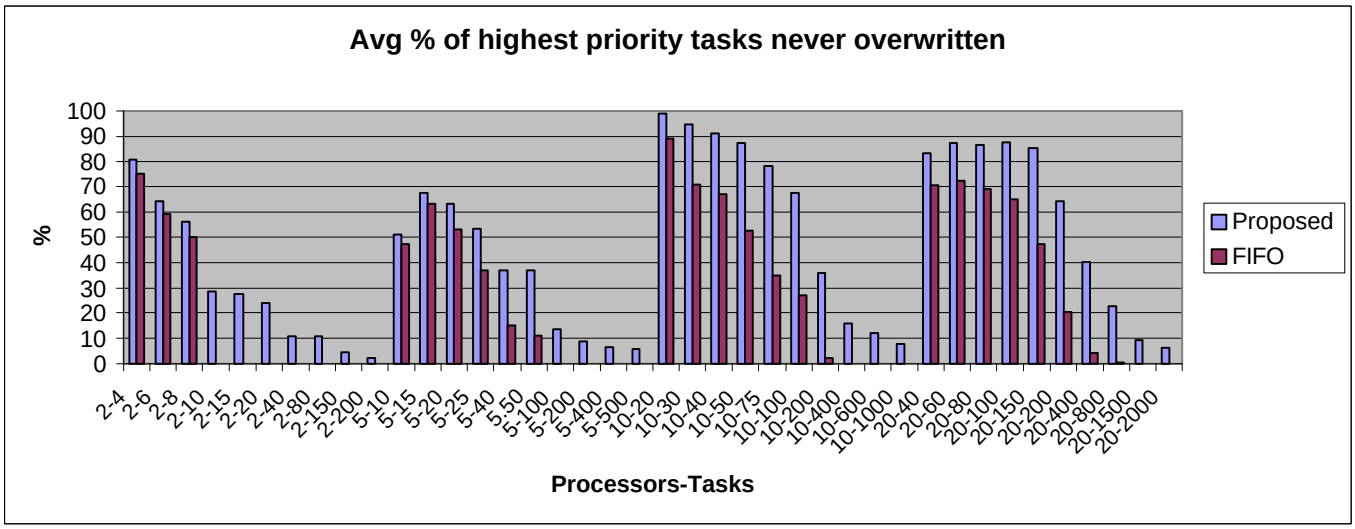

Figure 3: Average percentage of highest priority tasks never overwritten

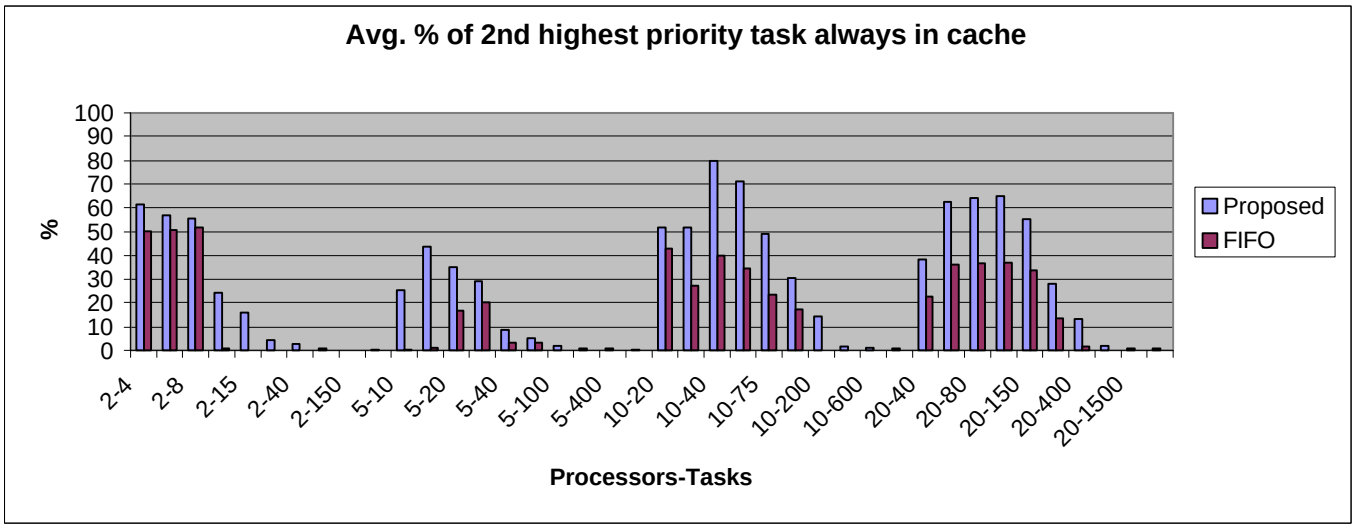

Figure 4: Average percentage o f 2nd highest priority tasks never overwritten

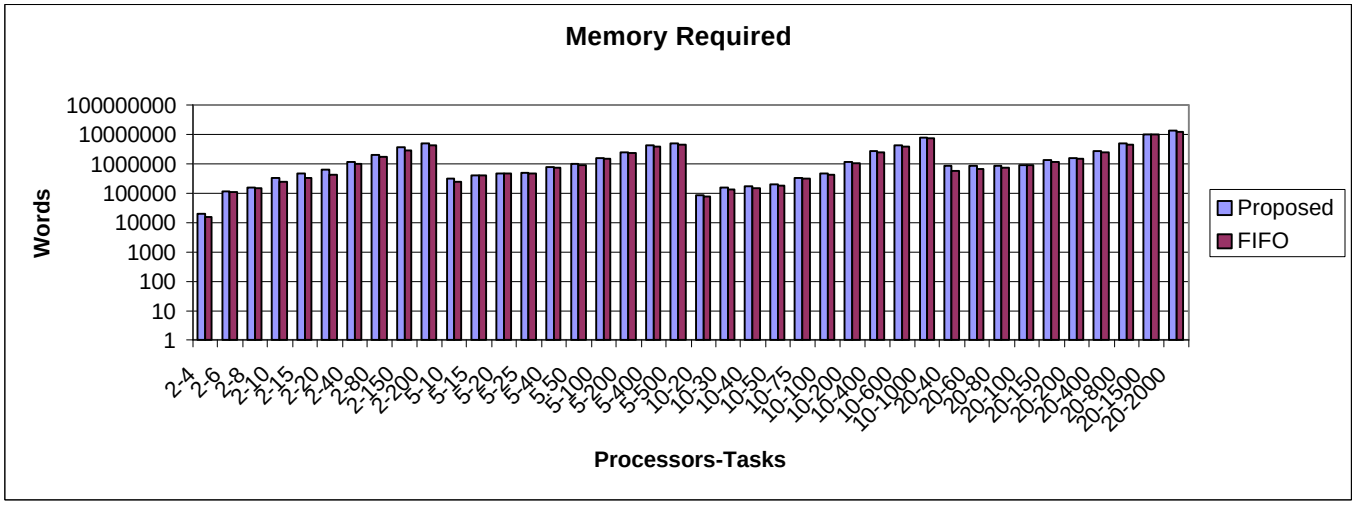

Figure 5: Memeory requirements for our scheme and the FIFO scheme

References

[1] L. Yanbing and W. Wolf, "A task-level hierarchical memory model for system synthesis of multiprocessors," in *Proceedings 1997. Design Automation Conference, 34th DAC. ACM, New York, NY, USA; 1997; xxix+788 pp. p.153 6*, 1997.

[2] J. Henkel, T. Benner, R. Ernst, W. Ye, N. Serafimov, and G. Glawe, "COSYMA: a software-oriented approach to hardware/software codesign," *Journal of Computer and Software Engineering*, vol. 2, pp. 293-314, 1994.

[3] J. Henkel, R. Ernst, U. Holtmann, and T. Benner, "Adaptation of partitioning and high-level synthesis in hardware/software co-synthesis," in *1994 IEEE/ACM International Conference on Computer Aided Design. Digest of Technical Papers (IEEE Cat. No.94CH3455 3). ACM, New York, NY, USA; 1994; xxx+771 pp. p.96 100*, 1994.

[4] R. K. Gupta and M.-G. De, "Hardware-software cosynthesis for digital systems," *IEEE Design & Test of Computers*, vol. 10, pp. 29-41 Phys Sci & Engin Journal holding 21(1984)-1912(1995);1913(1996)-, 1993.

[5] R. K. Gupta and M.-G. De, "Constrained software generation for hardware-software systems," , 1994.

[6] M. F. Parkinson, P. M. Taylor, and S. Parameswaren, "An Automated Hardware/Software Codesign (HSC) using VHDL," *Proceedings of First Asian Pacific Conference on Hardware Description Languages, Standards and Applications*, pp. 267-280, Dec. , 1993.

[7] M. F. Parkinson and S. Parameswaran, "Profiling in the ASP codesign environment," in *Proceedings of the Eighth International Symposium on System Synthesis (IEEE Cat. No.95TH8050). IEEE Comput. Soc. Press, Los Alamitos, CA, USA; 1995; xiii+175 pp. p.128 33*, 1995.

[8] S. Prakash and A. C. Parker, "Synthesis of application-specific multiprocessor systems including memory components," in *Proceedings of the International Conference on Application Specific Array Processors (Cat. No.92TH0453 1). IEEE Comput. Soc. Press, Los Alamitos, CA, USA; 1992; xii+698 pp. p.118 32*, J. Fortes, E. Lee, and T. Meng, Eds., 1992.

[9] S. -*Prakash and A. C. Parker, "Synthesis of application-specific multiprocessor systems including memory components," *Journal of VLSI Signal Processing*, vol. 8, pp. 97-116, 1994.

[10] W. H. Wolf, "An architectural co-synthesis algorithm for distributed, embedded computing systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 5, pp. 218-229, 1997.

[11] A. Rae and S. Parameswaran, "Application-specific heterogeneous multiprocessor synthesis using differential-evolution," in *Proceedings. 11th International Symposium on System Synthesis (Cat. No.98EX210). IEEE Comput. Soc, Los Alamitos, CA, USA; 1998; xii+164 pp. p.83 8*, 1998.

[12] K. Danckaert, F. Catthoor, and M.-H. De, "System level memory optimization for hardware-software co-design," , 1997.

[13] D. Verkest, S.-J. L. Da, Jr., C. Ykman, K. Croes, M. Miranda, S. Wuytack, F. Catthoor, J.-G. De, and M.-H. De, "Matisse: a system-on-chip design methodology emphasizing dynamic memory management," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 21, pp. 185-194, 1999.

[14] S. Wuytack, F. Catthoor, J.-G. de, B. Lin, and M.-H. De, "Flow graph balancing for minimizing the required memory bandwidth," in *Proceedings. 9th International Symposium on System Synthesis (Cat. No.96TB100061). IEEE Comput. Soc. Press, Los Alamitos, CA, USA; 1996; xii+145 pp.*, 1996.

[15] P. R. Panda, N. D. Dutt, and A. Nicolau, "Local memory exploration and optimization in embedded systems," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 3-13, 1999.

[16] P. R. Panda and N. D. Dutt, "Low-power memory mapping through reducing address bus activity," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, pp. 309-320, 1999.

[17] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, "Augmenting loop tiling with data alignment for improved cache performance," *IEEE Transactions on Computers*, vol. 48, pp. 142-149 Phys Sci & Engin Journal holding 117(1968)-1944(1995);1945(1996)-, 1999.

[18] P. R. Panda and N. D. Dutt, "Behavioral array mapping into multiport memories targeting low power," in *Proceedings. Tenth International Conference on VLSI Design (Cat. No.97TB100095). IEEE Comput. Soc. Press, Los Alamitos, CA, USA; 1997; xxxvii+566 pp. p.268 72*, 1997.

[19] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient utilization of scratch-pad memory in embedded processor applications," in *Proceedings. European Design and Test Conference. ED & TC 97 (Cat. No.97TB100102). IEEE Comput. Soc. Press, Los Alamitos, CA, USA; 1997; xxxvi+634 pp. p.7 11*, 1997.

[20] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, "Improving cache performance through tiling and data alignment," in *Solving Irregularly Structured Problems in Parallel. 4th International Symposium, IRREGULAR '97 Proceedings. Springer Verlag, Berlin, Germany; 1997; x+288 pp. p.167 85*, G. Bilardi, A. Ferreira, R. Luling, and J. Rolim, Eds., 1997.

[21] D. B. Kirk, J. K. Strosnider, and J. E. Sasinowski, "Allocating SMART cache segments for schedulability," in *Proceedings. EUROMICRO '91. Workshop on Real Time Systems. IEEE Comput. Soc. Press, Los Alamitos, CA, USA; 1991; ix+231 pp. p.41 50*, 1991.

[22] D. B. Kirk and J. K. Strosnider, "SMART (strategic memory allocation for real-time) cache design using the MIPS R3000," in *Proceedings. 11th Real Time Systems Symposium (Cat. No.90CH2933 0). IEEE Comput. Soc. Press, Los Alamitos, CA, USA; 1990; xi+341 pp. p.322 30*, 1990.

[23] L. Yanbing and W. Wolf, "Hardware/software co-synthesis with memory hierarchies,ICCAD, 1998.

[24] L. Yanbing and W. H. Wolf, "Hardware/software co-synthesis with memory hierarchies," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 1405-1417, 1999.