Improving Memory Space Utilization in Multi-core Embedded Systems using Task Recomputation

[1]Hakduran Koc,[2]Suleyman Tosun,[3]Ozcan Ozturk,[4]Mahmut Kandemir

[1]University of Houston – Clear Lake, Houston, TX 77058, USA

[2] Ankara University, Ankara, 06500,Turkey

[3]Bilkent University, Ankara, 06800, Turkey

[4]Pennsylvania State University, University Park, PA, 16802, USA

Abstract

As embedded applications are processing increasingly larger data sets,keeping their memory space consumptions under control is becoming a verypressing issue. Observing this, several prior efforts have considered memoryspace reduction techniques (in both hardware and software) based on data compression and lifetime-based memory recycling. In this work,we propose and evaluate an alternate approach to memory space saving in multi-core embedded architectures such as chip multiprocessors. The uniquecharacteristic of our approach is that it recomputes the results of selecttasks in a given task graph (which represents the application), instead of storing these results in memory andaccessing them from there as needed. Our approach can work under a givenperformance degradation bound and reduces memory space requirements underthis bound. Our experimental resultsare very encouraging and show that the proposed approachreduces memory space requirements of our task graphs by as much as 19.5%,the average savings being around 11.3%.

Keywords:*Memory Space Reduction, Task Recomputation, Multi-core Architecture, Embedded Systems.*

1. Introduction

Memory space consumption is an important metric to optimize for many embedded designs with tight memory constraints. While this is certainly true for both code and data memory, the rate at which the data sizes of embedded applications increase far exceeds the rate at which their code sizes increase. As a result, optimizing for data memory size is becoming increasingly more important than optimizing for code memory size. Several research papers aimed at reducing the data space requirements of embedded designs and proposed different techniques that can be adopted by optimizing compilers and design synthesis tools. These techniques range from compressing data [2, 28] to lifetime based memory reuse analysis [1, 16, 24] to code restructuring for memory reuse [13, 14, 15, 25].

This paper proposes a novel approach for reducing memory space consumption based on task recomputation. The basic idea is to reduce memory space demand by recomputing select tasks (in a task graph representation of the program) whenever their results are needed, instead of storing those results in memory (after their first computation) and accessing them from memory. While this approach can reduce memory demand, performing frequent recomputations can also lead to an increase in overall execution latency. In other words, there is a clear tradeoff between performance and memory space consumption. Consequently, this approach should be applied with care to select tasks only. Working on a task graph representation of a given embedded application, we propose a fully-automated scheme that identifies the tasks to recompute in such a way that the potential negative impact on execution time is minimized.

Focusing on an embedded multi-core architecture, the proposed approach first identifies thecritical paths in the task graph under consideration. It then marks all the tasks that sit in the critical paths as non-recomputable, meaning that these tasks are computed only once andtheir results are stored in memory for future use as long as they are needed. The remaining tasks, i.e., those that are not in the critical path, are marked as recomputable. The restof our approach traverses the tasks marked as recomputable and selects a subset of them (to be recomputed) such that the overall increase in execution latency is bounded by apreset value (typically, a designer-specified parameter). A particularly interestingoptimization problem that can be instantiated from our general problem description is to minimize the memory space requirements (by maximizing task recomputation) without increasing the original execution latency (i.e., the latency that would be obtained when no taskrecomputation is used). This can be made

possible by not allowing any path in the taskgraph to have a latency which is larger than that of the critical path.

We implemented our approach and tested it using eleven different task graphs (both automatically generated and extracted from applications). Our experimental analysis shows that the proposed approach can be used as a practical tool for studying the performance/memory space consumption tradeoffs in embedded designsthat accommodate a multi-core architecture. Specifically, for the example task graphs in our experimental suite, we found that our approach can reduce memory requirements by about 11.3% on average without any increase in original execution latency. Also, with a 20% allowable increase in original execution latency, we were able to increase our memory savings by up to 24.5%. Our experimental analysis also shows that this taskrecomputation based approach is effective with both unoptimized designs and designs that have already been optimized (based on data lifetime analysis).

The remainder of this paper is organized as follows. We revise the previous work on memoryspace optimization in Section 2. In Section 3, we illustrate, through an example, how task recomputation can save memory space. A formal description of our algorithm that recomputes select tasks every time their results are needed is given in Section 4. Our experimental results obtained using eleven task graphs are presented in Section 5. Finally, Section 6 concludes the paper and points out the future research directions onthis topic.

2. Related Work

Prior research considered data reuse and data lifetime analysis as potential solutions to the memory space optimization problem. Most of these approaches are based on loop level transformations [13, 14, 15, 25, 26] that modify the order of execution of loop iterations to use the memory hierarchy effectively. McKinley et al [19] present an approach to perform necessary loop transformations that exploit the inherent spatial and temporal reuse available in the program. Liu et al [18] present a loop fusion algorithm based on the loop dependency graph model. Several approaches have been proposed for reducing data space requirements of embedded applications by analyzing the lifetimes of variables [3, 4, 29]. Catthoor et al [3] showed how loop fusion can be used for minimizing data space requirements. An algorithm to accurately estimate the minimum memory size for array intensive applications is proposed in [29]. Based on live variable analysis, Zhao and Malik [29] transform the memory size estimation into an equivalent mathematical problem which can be solved by integer point counting.Hicks [9] proposed a compiler-directed storage reclamation scheme using object lifetime analysis which performs garbage collection by having the compiler insert deallocation code.

Data space optimizations have also been investigated by many researchers. Palem et al [22] proposed data remapping for pointer-intensive dynamic applications to decrease the memory requirements along with energy consumption. In their MDO work [12], Kulkarni et al aim at obtaining a data layout which has the least possible conflict misses. A combination of both loop and data transformations has also been explored by different research groups [10, 21].Kandemir et al [10] describe a compiler algorithm that considers loop and data layout transformations in a unified framework for optimizing cache locality on uniprocessor and multiprocessor machines. On the other hand, O'Boyle and Knijnenburg [21] propose an extended algebraic transformation framework to eliminate the temporary copies.

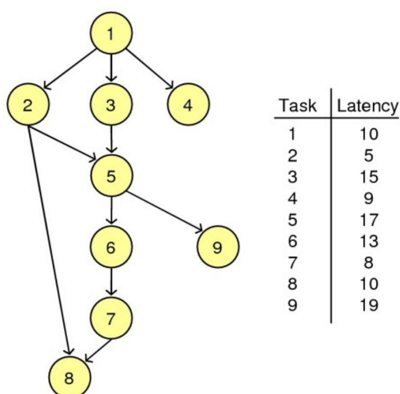
Compression techniques have also been used to reduce the memory footprint of both program code and application data.Cooper and McIntosh [5] uses pattern-matching to coalesce the instruction sequences to reduce the size of the compiled code.In VLIW architectures, Ros and Sutton [23] applied code compression algorithms to instruction words. Methods that use profile information have been proposed as well [7]. Code compression has also beenapplied to VLIW architectures that use Variable-to-fixed (V2F) coding [27]. An extension to this approach, called the variable-sized-block method, is presented by Lin et al [17]. Prior data compression related efforts include both hardware and software approaches. Benini et al [2] propose a hardware-assisted data compression that uses on-the-fly data compression and decompression. A first level cache compression technique is proposed in Yang et al [28] to reduce the number of cache miss as well as miss penalty.

Data recomputation is utilized in [11] for saving memory space. As compared to this prior effort, our work focuses on a multi-core architecture and proposes a fast heuristic solution. In comparison, the work in [11] considers a single CPU based system and uses integer linear programming (ILP). Also, recomputation is used in [30] to improve performance of chip multi-processors and in [31] to minimizing write activities to non-volatile memories.

3. Task Recomputation

In our approach, we use the task graph representation of a given embedded application. A task graph is a directed acyclic graph where nodes are tasks and edges represent

the dependencies among these tasks. An edge from task t_i to task t_j indicates that the data computed by t_i is used by t_j. Since the execution latency of such a task graph is determined by the critical path(s), depending on the properties of the task graph, it might be possible to perform recomputations without incurring performance overhead. Let us consider the example task graph given in Figure 1 with 9 tasks running on an embedded multi-CPU (chip multiprocessor) architecture with two homogeneous processor cores. The execution latency of each task is also shown on theright hand side of the same figure. Based on these latencies, the critical path in this example is 1-3-5-6-7-8, which has a total execution latency of 73.



Task	Latency
1	10
2	5
3	15
4	9
5	17
6	13
7	8
8	10
9	19

Fig. 1: An example task graph with 9 tasks. The source and the sink nodes are not shown.

The corresponding schedule for this task graph (without any recomputations) is shown in Figure 2(a).As can be seen, CPU_2 is idle more than half of the execution, which indicates a possible recomputation opportunity without increasing the original execution latency. Assuming that each task consumes 10 units of memory space to store its results and we do not employ any lifetime based memory space recycling (i.e., no automatic garbage collection), the total memory space required for storing the data manipulated will be 90 (10 × 9) units. By exploiting lifetime analysis, on the other hand, one can come up with a better memory behavior. This can be achieved by using a conflict graph [20] and applying graph coloring algorithm [20, 6] to this conflict graph to identify the tasks whose lifetimes do not overlap.Note that this problem is slightly different from the conventional register allocation problem since each task can have different memory requirements. Then, the number of colors returned gives the minimum number of tasks that need to store their results. For the above example, four memory spaces will be needed if such a lifetime analysis is employed, reducing the total memory space requirement from 90 to 40. However, further reductions in the memory space requirements can be achieved using recomputations by exploiting the idle periods that appear in the schedule of CPU_2. Consider, for instance, the idle period (t=24-42) between tasks t_4 and t_9 scheduled on this CPU. Since t_4 is the only task that requires the output of task t_1 after t=10, the t_1's output will not be needed if we recompute task t_1 right before computing t_4. This way we can further reduce the memory space requirement from 40 to 30.The resulting schedule is shown in Figure 2(b). As can be seen, the total execution latency is not affected by this recomputation. That is, it is possible to reduce memory space requirements of the task graph by carefully recomputing the select tasks without incurring any performance penalty.

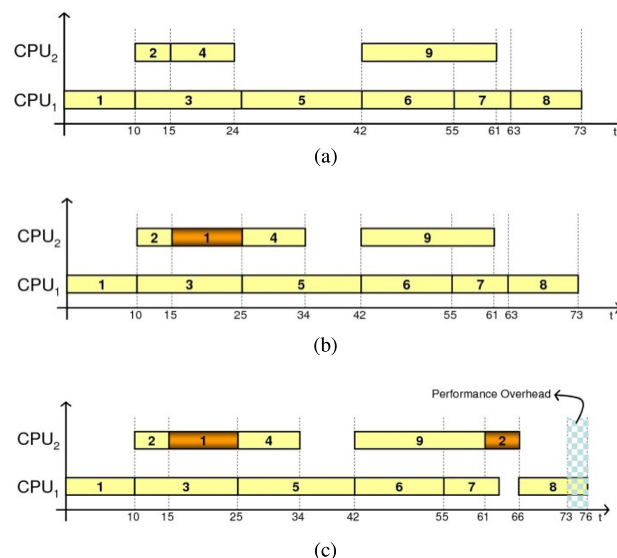
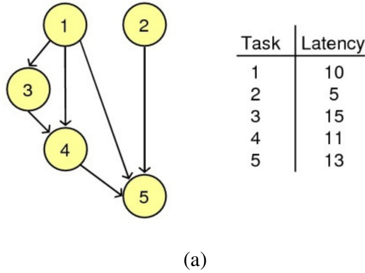


Fig. 2: An example scheduling scenario for a two CPU system. The x-axis corresponds to the execution time. (a) Schedule without any recomputations. (b) Schedule when recomputation is employed without any increase in the original execution latency. (c) Schedule when recomputation is employed with a maximum of 5% allowable increase in the original execution latency.

Although there might be different recomputation possibilities in a given schedule, not all of them can reduce the memory space requirements and not all of them come without performance overheads. On the other hand, in some cases, one can tolerate an increase in execution latency up to a certain level which can be captured by a preset value, a designer-specified parameter. Figure2(c) illustrates how we can achieve further savings in memory space requirements of our example task graph when a maximum of 5% increase in execution latency is allowed. In this case, it is sufficient to keep the outputs of only two tasks in memory at the same time, reducing the memory space requirement from 30 units to 20 units. The performance overhead incurred due to recomputing task t_2 before task t_8 is 3, i.e., the total execution latency is increased from 73 to 76, as depicted in Figure 2(c).

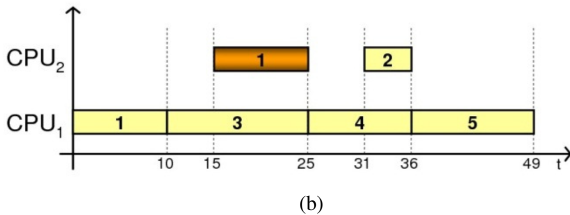
It is important to note that a task t_i does not need to be recomputed every time its output is needed by some other task t_j. Instead, it is possible to recompute task t_i initially a

couple of times, and then storing its result in memory when recomputation is no longer beneficial. Let us consider the example task graph given in Figure 3(a). Tasks t_3, t_4, and t_5 in this graph all depend on the output of task t_1. Based on the corresponding schedule shown in Figure 3(b), task t_3 immediately uses the output generated by t_1. However, tasks t_3 and t_5 will receive the output of t_1



Task	Latency
1	10
2	5
3	15
4	11
5	13

(a)



(b)

Fig. 3: An example task graph and task execution latencies (a) and the corresponding schedule with recomputations (b).

either through recomputation (of task t_1) or from the memory. For task t_4, recomputation without any increase in the total execution latency is possible. On the other hand, this is not the case for task t_5. As this example illustrates, some of the tasks can use recomputation to obtain their inputs, whereas some others can obtain the same inputs through the memory at different points in the execution of the task graph.Overall, this discussion shows that recomputation can be an effective means of reducing memory space requirements of applications executing on multi-CPU embedded systems. The next section presents and discusses our recomputation algorithm.

4. Details of Our Approach

While the previous section explains our approach at a high level, in this section, we discuss the details of the proposed approach. Algorithm 1 gives a sketch of our approach. This program takes five inputs, namely, the task graph in question (*TG(V,E)*) , the number of processors (*P*), the original execution latency (*L*) without any recomputation, the number of recomputation levels (*R-Level*), and the performance overhead allowed (*OA*), and it returns, as output, the new schedule with recomputation. It is important to note that we start by a schedule which has been obtained from a *performance-oriented* task scheduling algorithm. The reason for this design choice is to minimize the impact of our approach on performance. In other words, we would like to keep our modifications to the schedule with the best performance at minimum. *TG(V,E)* denotes a task graph scheduled with respect to performance constraints, where $V=\{v_1,v_2, \ldots, v_n\}$ is the vertex set representing the tasks and $E=\{e_1, e_2, \ldots, e_k\}$ is the edge set representing the dependencies among these tasks. Notice that the tasks in *V* are ordered starting from the closest task to the sink. Based on this order, the slack value for each task is calculated. Note that the slack for a task indicates the amount of extra latency it can tolerate without affecting the overall execution latency of the task graph being analyzed.*R-Level* indicates the maximum number of subsequent recomputations allowed to execute a task. Also, the value of global benefit (denoted *G-Benefit* in the algorithms) obtained by exploiting recomputations is initialized to 0.

Algorithm 1*Memory Optimization*
1. **Input:** TG(V,E), P, L, R-Level, and OA
2. **Output:** Schedule with recomputation
3. G-Benefit = 0
4. **for all**$i \in \|V\|$**do**
5. Order task v_i
6. **end for**
7. **for all**$i \in \|V\|$**do**
8. Calculate slack for v_i
9. **end for**
10. **for all**$i \in \|V\|$**do**
11. Determine all the paths to the source originating from v_i
12. Construct Recompute Set for each path if possible
13. Search Best (v_i, R-Level, OA, 0)
14. **end for**

For each node of the given task graph, the *Memory Optimization* algorithm looks for the potential recomputation patterns.This is achieved by a call to a function named *Search Best*, which recursively tries to perform recomputations based on the slacks in the task graph and the specified performance overhead. Algorithm2 gives the sketch of this function.This function takes the task (*V*), the number of recomputation levels (*Level*), the performance overhead allowed (*OA*), and the memory benefit brought by the current recomputation path so far (*MB*). When *Search Best* is invoked from the *Memory Optimization* function, it is passed the maximum number of recomputations allowed and the maximum overhead possible. The initial memory benefit is passed as 0. Then, *Search Best* traverses all the predecessors of the given node (*V*). First, it computes t_{diff}, which is the difference between the end of the lifetimes of the current node (*V*) and its predecessor (v_i). This indicates whether recomputing v_i is beneficial or not. If another task (other than *V*) is using the same output, (i.e., the results of v_i will be kept in the memory in any case), the lifetime of v_i may go beyond that of *V*, which suggests that recomputing v_i is

not beneficial in terms of saving additional memory space. As can be seen, if t_{diff} is less than 0, the recursion does not go any further. The second constraint checked by this function is whether the slack of the current task is long enough to accommodate a recomputation. This is checked by $V_{slack} \geq v_i.exec$. If it is possible to recompute v_i within the slack time of *V*, the memory space saving (*MB-new*)

Algorithm 2 *Search Best*

```
1.  Input: V, Level, OA, MB
2.  Output: Optimum Recomputation
3.  for all i ∈ |Pred{V}| do
4.  t_diff ← V.end - v_i.end
5.  if t_diff > 0 then
6.  if (V_slack ≥ v_i.exec) or (OA permits v_i.exec - V_slack) then
7.      MB-new ← MB + v_i.exec × v_i.memory
8.      OA-new ← update OA
9.      if MB-new > G-Benefit then
10.             G-Benefit ← MB-new
11.             Add v to the recomputation list
12.             Update scheduling accordingly
13.     end if
14. end if
15. end if
16. if (t_diff > 0) and (Level > 0) then
17. if (V_slack ≥ v_i.exec) or (OA permits v_i.exec - V_slack) then
18.     Search Best(v_i, Level-1, OA-new, MB-new)
19.     end if
20. end if
21. end for
```

brought by this recomputation (in addition to the possible previous recomputation(s)) is calculated. This value is obtained based on the parameter passed to the function (*MB*), that is, the memory savings brought by the previous recomputations. If this is the first recomputation in the path, this value is 0. Although there might be different criteria to select the recomputations to perform (from among the set of all possible recomputations), we use *time* × *memory* as the metric for memory space savings, where *time* is the reduction in lifetime of the task's output and *memory* is the corresponding task's memory consumption. While we prefer not to increase the overall execution latency, in some execution environments it might be possible to tolerate up to a certain performance overhead, which is given as *OA* in this algorithm. If it is possible to recompute the task under consideration within the tolerated performance overhead bound, it is recomputed and the overhead allowed (*OA*) is updated accordingly. This value is then passed to the next *Search Best* function call. In order to decide whether we can continue performing recomputations in the current path, we check whether the maximum number of subsequentrecomputations allowed has been reached or not (in addition to the conditions discussed above).

It is important to emphasize that, using this algorithm, all of the legal paths from a lower level node to a higher level node in the task graph are evaluated for possible recomputations. If performing one or more recomputations on a path reduces the memory consumption, this reduction is stored in *G-Benefit* and the corresponding path is recorded as well. Among these paths, the one that brings the maximum memory space savings is chosen.

Let us now discuss how this algorithm operates on the task graph shown in Figure 1. Given the performance-scheduled graph, our algorithm calculates the time slack for each task. For example, since CPU_2 is idle for t=24-42 and there is no task succeeding node 4 in the task graph, the slack for task v_4 is 18. Then, checking every predecessor of each node, a *Recompute Set* is constructed for each path to source. In this example, since v_4 has a path to source through v_1 and has enough slack available, (4, 1) is the recompute set for v_4. On the other hand, v_5 has no element in its recompute set. If it had time slack available, it could have (5, 2), (5, 3), (5, 2, 1) and (5, 3,1) in its recompute set. Next, the algorithm calls the recursive function *Search Best* to determine if the recomputation brings any memory saving. The function also determines the most efficient recompute set among all sets if there is more than one. In our example, it takes into account the recomputation for the set (4,1) and updates the schedule and required parameters for memory.

5. Experimental Evaluation

Our goal in this section is to present an experimental evaluation of the proposed recomputation based approach. For this purpose, we used both automatically-generated task graphs and task graphs extracted from benchmarks. For the first part, we used the TGFF tool [8] and generated several task graphs. Unless otherwise stated, we assumed 10 processors in our experiments.Also, in our experiments, each task has a latency value between 7-20 units, and uses a memory size of 4-15 units.We made experiments with two groups of automatically-generated task graphs. Our first group of task graphs (tg1 through tg4) have the same edges/node ratio, but each with different number of nodes and edges. Thesecond group of task graphs (tg5 through tg7) on the other hand is comprised of graphs with different edges/noderatios. The reason that we make experiments with these two different sets oftask graphs is to evaluate the behavior of our approach under the differentscenarios. The important characteristics of our task graphs are given inTable1. The first column of this table gives the name of the task graph and the next two columns give the number of nodes and edges in the graph. The fourth column of the table shows the total sizeof the data manipulated by the nodes of the task graph when no

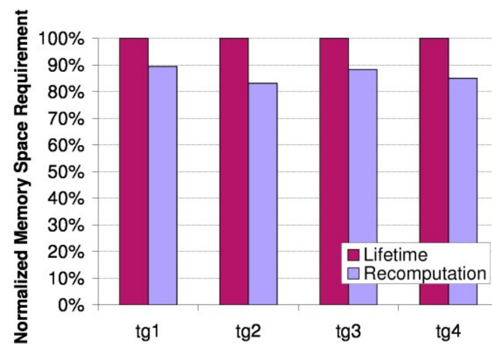
memory spacesaving technique is employed. The next column of Table 1 onthe other hand gives the amount of data space requirements when a *lifetime based* memory space recycling is used. In this scheme, the memory spaceallocated for storing the results of a task is recycled when the data storedare no longer needed. When comparing these two columns of this table, wesee that a lifetime analysis based memory space recycling cuts the memoryspace requirements by 49.5% on the average. Our goal is to further increasememory space savings through task recomputation. Finally, the last column of the Table 1 shows the execution latency of each task graph when *no*recomputation is used. In the rest of thissection, all memory saving results are given as *normalized values* withrespect to the corresponding values in the fifth column ofTable 1 (i.e., over the lifetime based approach). Similarly, the performance overheads (if any) incurred by our approach are given as *normalized values* with respect to the corresponding valueslisted in last column of this table.

Table 1: Task Graphs and their important characteristics.

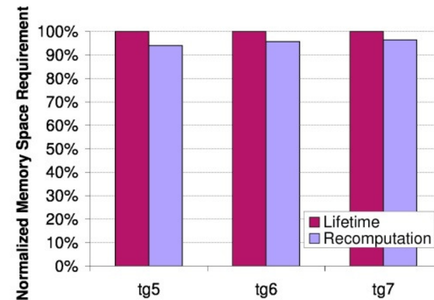
Task Graph Label	Number of Nodes	Number of Edges	Data Size (No Opt.)	Data Size (Lifetime Ana.)	Latency
tg1	11	16	86	47	51
tg2	14	19	136	59	37
tg3	20	30	184	94	73
tg4	31	45	306	139	71
tg5	20	40	192	99	75
tg6	21	50	180	92	83
tg7	20	60	195	110	131

The bar-chart in Figure 4 shows the normalized memory spacesavings obtained by our recomputation-based approach for our task graphs. Inthese experiments, we use the version of our approach (whose pseudo-code isgiven in Algorithm1) that does not increase the originalexecution latency. We seethat our approach reduces the memory space requirements by 13.6% for thefirst groupof task graphs and 6.5% for the second group of task graphs.These results clearly show the effectiveness of our approach in reducingmemory space requirements and the important point we want to emphasize here is that these savings come at no performance cost.

In our next set of experiments, we study the tradeoff between memory spacesaving and performance overhead by allowing our approach to tolerate certain(specified) increase in original execution latency. That is, we test ourapproach whose pseudo-code is given inAlgorithm 2. We see from the results given inFigure 5 (which are given for two of our task graphs) that, by tolerating 20% increase in original execution latency, we can save 24.5%memory space. This gives the designer to perform a tradeoff analysis betweenmemory space savings and performance overheads.



(a)



(b)

Fig. 4: Normalized memory requirements of our approach for the task graphs in Table 1.

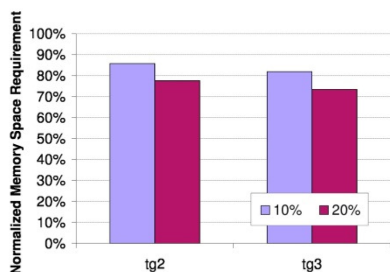


Fig. 5: Normalized memory requirements with varying performance overheads.

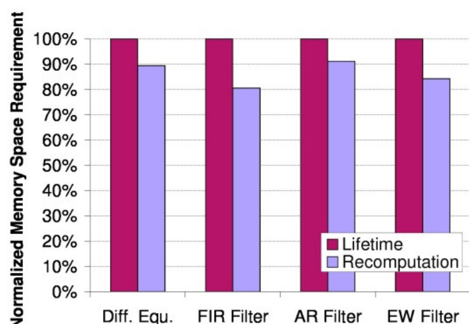


Fig. 6: Normalized memory requirements for the task graphs extracted from benchmarks.

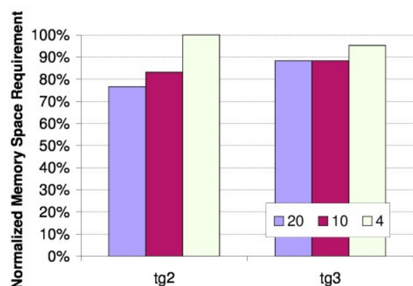


Fig. 7: Normalized memory requirements with different number of CPUs.

In addition to these task graphs generated by the TGFF tool, we alsoperformed experiments with the task graphs extracted from several embedded applications. The normalized memory requirements for this set of task graphs are given in Figure 6 for the case when no increase in the original execution latencies is tolerated. We see that our recomputation based approach is very effective in reducing memory space requirements of these task graphs as well, achieving an average memory saving of 13.9%.

We next evaluate the behavior of our approach when the number of CPUs is varied. Recall that the default number of CPUs used so far in our experiments was 10. As before, we focus only on task graphs *tg2* and *tg3*. Allowing no increase in original executionlatency, Figure 7 gives the memory space savings (over the versions that uselifetime based analysis) under different number of CPUs. We observe from these results that, as the number of CPUs increases, the normalized memory requirement of the application decreases. However, after reaching a CPU count that handles all the concurrent paths in the task graph, increasing the number of CPUs further will not affect the memory requirement, as seen in the figure for *tg3*.

6. Conclusion and Future Work

The main contribution of this paper is a novel memory space saving schemefor embedded multi-CPU systems. Starting with a task graph scheduled for thebest performance, the proposed approach identifies a set of tasks andrecomputes their results every time they are needed (instead of computingthem once, storing their results in memory and accessing those results from memory whenever needed). Weperformed experiments with several task graphs (that represent different execution scenarios) and the results obtained sofar show the effectiveness of this recomputation based approach. Specifically, our experimental analysis shows that we can save significant memory space (over a lifetime based approach) without incurring any performance penalty. Our approach also works under the cases where a certain performance penalty can be tolerated. Our futurework involves extending this idea to a dynamic compilation environment wherethe recomputation decisions are taken at runtime by a dynamic compiler.

References

[1] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memoryallocation performance. In Proceedings of the ACM Conference onProgramming Language Design and Implementation, pages 187–196, 1993.

[2] L. Benini, D. Bruni, A. Macii, and E. Macii. Hardware-assisted datacompression for energy minimization in systems with embedded processors. InProceedings of the Conference on Design, Automation and Test in Europe, page449, 2002.

[3] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. Kjeldsberg, T. V.Achteren, and T. Omnes. Data Access and Storage Management for Embedded Programmable Processors.Kluwer Academic Publishers, Boston, MA, USA,2002.

[4] F. Catthoor, E. de Greef, and S. Suytack. Custom Memory ManagementMethodology: Exploration of Memory Organization for Embedded MultimediaSystem Design. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[5] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In Proceedings of the ACM Conference on Programming LanguageDesign and Implementation, pages 139–149, 1999.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms.The MIT Press/McGraw-Hill, Cambridge, MA, USA, 2001.

[7] S. Debray and W. Evans. Profile-guided code compression. In Proceedings ofthe ACM Conference on Programming Language Design and Implementation,pages 95–105, 2002.

[8] R. P. Dick, D. L. Rhodes, and W.Wolf. TGFF: Task Graphs For Free. InProceedings of the International Workshop on Hardware/Software Codesign,pages 97–101, 1998.

[9] J. Hicks. Experiences with compiler-directed storage reclamation. InProceedings of the Conference on Functional Programming Languages andComputer Architecture, pages 95–105, 1993.

[10] M. Kandemir, J. Ramanujam, and A. Choudhary.Improving cache locality by a combination of loop and data transformations. IEEE Transactions onComputers, 48(2):159–167, 1999.

[11] M. T. Kandemir, F. Li, G. Chen, G. Chen, and O. Ozturk.Studyingstorage-recomputation tradeoffs in memory-constrained embedded processing.In Design, Automation and Test in Europe Conference, pages 1026–1031, 2005.

[12] C. Kulkarni, F. Catthoor, and H. D. Man. Advanced data layout optimization formultimedia applications. In Proceedings of the IPDPS Workshops on Paralleland Distributed Processing, pages 186–193, London, UK, 2000.Springer-Verlag.

[13] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance andoptimizations of blocked algorithms. In Proceedings of the InternationalConference on Architectural Support for Programming Languages andOperating Systems, pages 63–74, 1991.

[14] W. Li. Compiling for Numa Parallel Machines. PhD thesis, Ithaca, NY, USA,1993.

[15] W. Li and K. Pingali. A singular loop transformation framework based onnon-singular matrices. International Journal Parallel Program, 22(2):183–205,1994.

[16] H. Lieberman and C. Hewitt. A real-time garbage collector based on thelifetimes of objects. Commun. ACM, 26(6):419–429, 1983.

[17] C. H. Lin, Y. Xie, and W.Wolf. LZW-based code compression for VLIW embedded systems. In Proceedings of the Conference on Design, Automationand Test in Europe, page 30076, 2004.

[18] M. Liu, Q. Zhuge, Z. Shao, and E. H.-M.Sha.General loop fusion techniquefor nested loops considering timing and code size. In Proceedings of theInternational Conference on Compilers, Architecture, and Synthesis forEmbedded Systems, pages 190–201, 2004.

[19] K. S. McKinley, S. Carr, and C.-W.Tseng.Improving data locality with looptransformations. ACM Transactions on Programming Languages and Systems,18(4):424–453, 1996.

[20] G. D. Micheli. Synthesis and Optimization of Digital Circuits.McGraw-HillHigher Education, 1994.

[21] M. F. P. O'Boyle and P. M. W. Knijnenburg.Integrating loop and datatransformations for global optimization. Journal of Parallel and DistributedComputing, 62(4):563–590, 2002.

[22] K. V. Palem, R. M. Rabbah, I. Vincent J. Mooney, P. Korkmaz, andK. Puttaswamy. Design space optimization of embedded memory systems viadata remapping. In Proceedings of the Joint Conference on Languages,Compilers and Tools for Embedded Systems, pages 28–37, 2002.

[23] M. Ros and P. Sutton. Code compression based on operand-factorization forvliw processors. In Proceedings of the Conference on Data Compression, page559, 2004.

[24] C. Ruggieri and T. P. Murtagh.Lifetime analysis of dynamically allocatedobjects. In Proceedings of the ACM SIGPLAN-SIGACT Symposium onPrinciples of Programming Languages, pages 285–293, 1988.

[25] L. Wang, W. Tembe, and S. Pande.Optimizing on-chip memory usage throughloop restructuring for embedded processors. In Proceedings of the InternationalConference on Compiler Construction, pages 141–156, 2000.

[26] M. J. Wolfe. High Performance Compilers for Parallel Computing.Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[27] Y. Xie, W.Wolf, and H. Lekatsas.Code compression for VLIW processors usingvariable-to-fixed coding. In Proceedings of the 15th International Symposiumon System Synthesis, pages 138–143, 2002.

[28] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. InProceedings of the 33rd Annual ACM/IEEE International Symposium onMicroarchitecture, pages 258–265, 2000.

[29] Y. Zhao and S. Malik. Exact memory size estimation for array computationswithout loop unrolling. In Proceedings of the 36th ACM/IEEE Conference onDesign Automation, pages 811–816, 1999.

[30] H. Koc, M.Kandemir, E. Ercanli, and O. Ozturk.Reducing Off-Chip Memory Access Costs Using Data Recomputation in Embedded Chip Multi-processors.In Proceedings of the 44th ACM/IEEE Design Automation Conference, pp.224-229, 2007.

[31] J. Hu et al. Minimizing write activities to non-volatile memory via scheduling and recomputation. In proceedings of the 8th Symposium on Application Specific Processors (SASP),pp.101-106, 2010.

Hakduran Koc is an assistant professor in Computer Engineering at University of Houston - Clear Lake, Houston, TX. He received his B.Sc. degree in Electronics Engineering from Ankara University, Ankara, Turkey in 1997 and his M.Sc. and Ph.D. degrees from Syracuse University, Syracuse, NY in 2001 and 2008, respectively. His research interests include embedded systems, computer architecture, and high level synthesis.

Suleyman Tosun received his B.Sc. in Electrical and Electronics Engineering from Selcuk University, Turkey, in 1997 and his M.Sc. and Ph.D. degrees in Computer Engineering from Syracuse University, NY, in 2001 and 2005, respectively. His research interests are embedded system design, reliability, design automation, and high-level synthesis of digital circuits.

Ozcan Ozturk received the Bachelor's degree from Bogazici University, Istanbul, Turkey, in 2000, the M.Sc. degree from the University of Florida, Gainesville, in 2002, and the Ph.D. degree from Pennsylvania State University, University Park, in 2007. His research interests are in the areas of multicore and manycore architectures, power-aware architectures, and compiler optimizations.

MahmutKandemiris received the B.Sc. and M.Sc. degrees in control and computer engineering from Istanbul Technical University, Istanbul, Turkey, in 1988 and 1992, respectively. He

received the PhD degree from Syracuse University, Syracuse, New York, in electrical engineering and computer science in 1999. His main research interests are optimizing compilers, I/O-intensive applications, and power-aware computing.