

Using Task Recomputation During Application Mapping in Parallel Embedded Architectures*

Suleyman Tosun

Computer Engineering Department
Selcuk University
Konya, Turkey

Mahmut Kandemir

Department of Computer Science
and Engineering
The Pennsylvania State University
University Park, PA, USA

Hakduran Koc

Department of Electrical
Engineering and Computer Science
Syracuse University
Syracuse, NY, USA

Abstract - *Many memory-sensitive embedded applications can tolerate small performance degradations if doing so can reduce the memory space requirements significantly. This paper explores this tradeoff by proposing and evaluating an algorithm for performing recomputations in select program points to reduce memory space occupation of data. Our algorithm targets heterogeneous computing platforms and operates with two user-specified parameters that bound performance degradation of the resulting code and its memory space demand. It explores the solution space, performing recomputations (instead of memory stores) for select tasks to reduce memory space demand.*

Keywords: Heterogeneous Chip Multiprocessors, recomputation, memory minimization.

1 Introduction

Embedded systems operate under very different constraints than their general-purpose counterparts. One of these constraints is the limited memory capacity, which motivates the compiler writers and application programmers for designing techniques that reduce memory space consumption of program code and data. The memory space limitation problem is much more severe in parallel embedded architectures since multiple processors can compete for the same storage space.

We can divide the techniques explored in the past to address this limited memory problem into two main categories. In the first category are techniques [1][1][2] that employ code and data compression. The idea behind these approaches is to represent code and data in as few bits as possible. The main challenge in front of this option is the potential performance degradation if decompressions occur on the critical path of execution at runtime and the extra energy spent during compressions and decompressions. The techniques in the second category [3][4][5], on the other hand, make use of lifetime analysis for program variables. Since the lifetime of a program variable is typically much shorter than the entire lifetime of the program that uses that variable, multiple variables can share the same memory

location if their lifetimes do not overlap. The main problem with these approaches, apart from the complex compiler-based analysis they require, is that the memory space savings achieved are bounded by the maximum total size of the live data at any given point in execution.

This paper explores a third option in reducing data memory space consumption of embedded applications. The idea is to reduce memory space requirements by performing extra *recomputations* instead of indiscriminately storing all intermediate results in memory. In this paper, we propose an algorithm that performs this storage space-performance tradeoff for task graphs mapped onto parallel embedded heterogeneous architectures (e.g., an embedded heterogeneous multiprocessor). Since such a recomputation-based approach incurs performance penalties (because of the fact that recomputing a task each time it is required is typically more costly than storing its output in memory and using it from there when required), an important issue is how to balance this overhead against the storage gains achieved. Since the answer to this question depends strongly on the specific execution environment under consideration, our algorithm is designed for enabling the exploration of this tradeoff/balance by accepting two parameters from the user. The first parameter controls the allowable performance degradation (with respect to the best performance possible), whereas the second one controls memory space consumption of the resulting output code (with respect to the minimum memory consumption possible).

We tested the viability of our approach by implementing it and performing experiments with it using several task graphs mapped onto heterogeneous computing platforms. We also compared our approach to two alternate schemes: one that does not perform any recomputations and one that performs recomputations very aggressively. Our experimental evaluation clearly shows that the proposed algorithm enables efficient exploration of the performance-storage tradeoff. To our knowledge, the only other previous efforts that study this tradeoff between storage and recomputation are [6] and [7], and they exclusively target single processor based environments.

*This work is supported by Selcuk University project number 06701088.

2 System Specification and Example

In today’s technology, it is possible to embed multiple processors in a single chip, which gives an opportunity to the designers to explore multiple design alternatives. Virtex-4 FPGA [8] is such an example, where up to two PowerPC blocks, a 32-bit MicroBlaze soft processor core, and an 8-bit PicoBlaze soft processor core are embedded into it. Our target architecture in this paper is a system that consists of multiple heterogeneous processors and a shared memory. The processor cores are assumed to be embedded in a single die and they are connected using shared bus. The communication between the processors and the memory system is conducted via the memory interface and the bus structure. Fig. 1(a) illustrates example target architecture with two CPUs. In this figure, two heterogeneous processor cores are embedded into a single chip, and they are connected to an off chip memory. The communication between memory and processors is conducted via a memory interface. Optionally, the processors can also have a local, on-chip memory space.

In our framework, we use a graph based model, called the task graph, to describe the system specification. A task graph is a directed-acyclic graph where the vertices represent the tasks and the edges represent the dependencies among these tasks. A directed edge between two tasks captures the fact that there is a data transfer between them. In our scheduling approach, we have to make sure that a task is scheduled only after all its input data are available from its predecessors. To make the scheduling problem easier, we add a source vertex and a terminal vertex to our task graphs. The source and terminal vertices are assumed to have zero delay and no memory consumption. In Fig. 1(b), we give an example task graph with nine vertices, where the vertices zero and eight represent the source and terminal vertices, respectively.

Motivational Example: Before moving to the technical details of our approach, let us give a motivational example to illustrate the trade-off between performance and memory consumption.

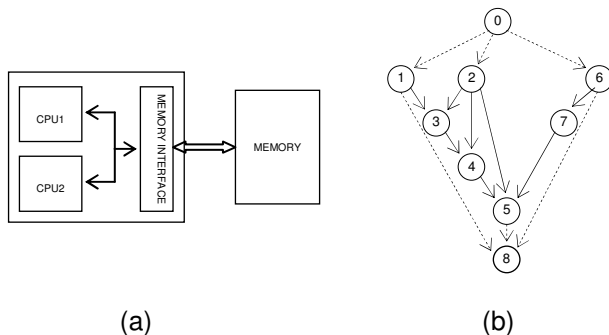


Fig. 1: (a) The target architecture and (b) an example task graph.

In this example, we first schedule the task graph in Fig. 1(b) without considering any recomputations (we give the details of the scheduling algorithm in Section 3). We then show how recomputation(s) can affect the memory space consumption and the overall performance of the design. Assume that we have two processors in our target architecture. For simplicity, assume further that all the tasks consume the same amount of memory space to store their results, namely, 20 units of memory. In Table I, we give the possible worst-case execution times (WCETs) of the tasks on processors CPU1 and CPU2. A possible schedule for the task graph in Fig. 1(b) is given in Fig. 2(a). Using lifetime analysis, the results that do not conflict with each other can share the same memory space. In Fig. 2(b), we give the conflict graph for the lifetimes of the results produced by the tasks. The minimum memory space required for the design can then be found using a graph coloring algorithm [9]. The conflict graph in Fig. 2(b) can be colored with five different colors, indicating that in this schedule, we need a minimum of five memory spaces as a result of memory reuse through lifetime analysis. This means that the schedule in Fig. 2(a) occupies a total of 100 units of memory space, resulting in 29% reduction over the memory usage when the results of all the tasks are stored separately without considering their lifetimes (if we stored the results of the seven tasks separately, this would take a total of 140 units of memory). Note that, in our approach, the schedule that does not consider recomputation of tasks is assumed to return the best performance and worst memory consumption values. The schedule in Fig. 2(a) completes the execution of all the tasks in 29 units of delay, indicating $P_{best}=29$, and its memory consumption is 100 units of memory space, which means $M_{worst}=100$. Our goal is to see whether we can achieve even a lower memory consumption using task recomputations. That is, *we want to achieve further memory savings over a lifetime based approach*.

Table I: WCETs of the tasks for the task graph in Fig. 1(b) on two heterogeneous processors.

	Tasks						
	1	2	3	4	5	6	7
CPU1	5	6	8	5	5	4	10
CPU2	8	8	7	6	5	5	8

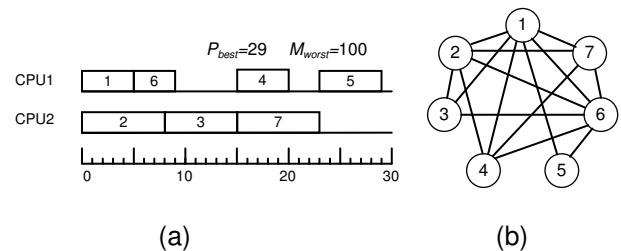


Fig. 2: (a) A schedule for the task graph in Fig. 1(b) without any recomputations, (b) the conflict graph for the lifetimes of the tasks’ results.

Let us now discuss how recomputation affects the performance and memory consumption results for this example. As can be easily observed from the task graph in Fig. 1(b), the minimum memory space we need must hold at least the results of three tasks since two of the tasks, the vertices 5 and 8, have three immediate predecessors. These tasks can start their executions when all the data from their predecessors are available; i.e., the data coming from the predecessors should be stored somewhere in the memory. Based on this minimum memory space requirement, we next determine our schedule, which is shown in Fig. 3. In this schedule, the tasks 1, 2, and 6 are recomputed instead of storing their data in memory. For example, task 2 is recomputed when task 4 needs its data. We should note here that, if we decide to recompute the result of task 3 instead of that of task 2, we may have to recompute task 1 as well since recomputing 3 depends on the availability of data from its predecessors and this may trigger multiple (i.e., a chain of) recomputations. Obviously, this results in a worse performance overhead as opposed to recomputing task 2. Thus, choosing the set of tasks to recompute can be very important in terms of limiting the resulting performance penalty. The schedule in Fig. 3 requires a minimum of three memory spaces. This means a total of 60 units of memory, which is a 57% reduction over the memory usage when the results of all the tasks are stored separately. If we compare the memory requirements of this approach against the lifetime analysis based one, we see that the schedule in Fig. 3 (that uses recomputations) brings 40% reduction over the one in Fig. 2(a). However, it also brings an extra performance penalty by increasing the execution time of the original schedule by 14.7%. In this work, we assume that the performance and memory consumption values of the schedule that uses recomputation as much as possible are the worst performance and the best memory consumption, respectively. As a result, we have $P_{worst}=34$ and $M_{best}=60$ for the schedule shown in Fig. 3.

Note that, in the above schedules, we did not account for the communication times between the tasks for clarity. However, in our implementation, we consider the time of writing/reading the data to/from the memory, and add a fixed delay after each task. To summarize, the analysis of this simple example shows that it is possible to tradeoff performance with memory space consumption through task recomputation.

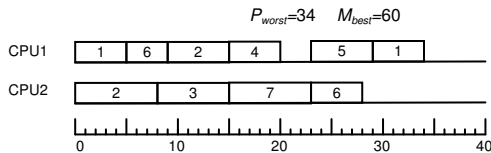


Fig. 3: The schedule of Fig. 1(b) when using recomputations. Tasks 1, 2, and 6 are recomputed.

3 The Proposed Approach

In this section, we present a heuristic approach, which helps designers to explore the memory space consumption/performance tradeoffs during the task mapping on parallel heterogeneous processors.

3.1 Design Space Exploration

As our small example in the previous section demonstrates, the solution we are searching strongly depends on the system requirements under consideration. If the embedded application at hand has severe memory limitations, then one may want to use more recomputations in order to minimize the overall memory consumption of the final design. However, most of the embedded applications require high performance, and they have strict (hard) deadlines to meet. In such a case, one may want to limit the number of recomputations to an acceptable level. Clearly, we have two extremes here. The first one (maximum recomputations) has minimum memory consumption but the worst performance (M_{best} and P_{worst}). The second one (no recomputations), on the other hand, has the best performance, but the maximum memory consumption (M_{worst} and P_{best}). Obviously, the solutions between these two extreme ends are the ones needed in a typical design. In our approach, to achieve a balanced design in terms of performance and memory consumption, we define two variables α and β . These two variables are user defined based on the best and worst performance and memory consumption values. The variable α is used for determining the memory consumption constraint. Specifically, we employ the following expression to determine the allowable memory space consumption:

$$M \leq (1 + \alpha)M_{best} \text{ such that } 0 \leq \alpha \leq \frac{M_{worst}}{M_{best}} - 1, \quad (1)$$

where M_{best} and M_{worst} are the best and worst memory consumption values, respectively, and M represents the memory consumption of the final design. The variable β , on the other hand, is used for bounding the performance of the design. The following expression is used for this purpose:

$$P \leq (1 + \beta)P_{best} \text{ such that } 0 \leq \beta \leq \frac{P_{worst}}{P_{best}} - 1, \quad (2)$$

where P_{best} and P_{worst} represent the best and worst execution times, respectively, and P represents the performance of the final design.

In Fig. 4, we illustrate the effects of the variables α and β on the design space exploration. As can be seen from this figure, the variables α and β limit the exploration region to a region of interest (for a particular design), forcing our

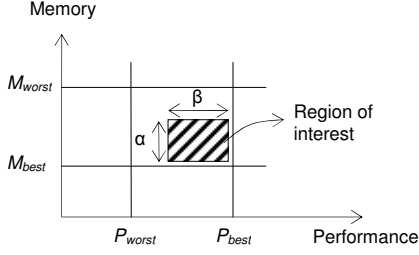


Fig. 4: Boundaries of the design space of interest. Performance in this context can be interpreted as 1/execution time.

scheduler to find a solution within these limits. We have to emphasize at this point that, when we choose the values for the variables α and β , we also have to consider the cases where there might be no solutions. For example, if we choose both the variables to be zero, then this means that we want both the best performance and the best memory consumption values at the same time. However, such a solution can be possible only in one case where both the schedules (i.e., the schedules with and without recomputations) are the same.

3.2 Overview of the Proposed Approach

We show the high level operation of our approach in Fig. 5. The inputs are the embedded system specification as a task graph, the worst case execution times (WCETs) of each task on each processor and the memory consumption of each task (in terms of units). Our approach starts with a preprocessing step, where two extreme schedules are found. The first schedule is the one that outputs the best performance (P_{best}) and the worst memory consumption (M_{worst}) values by scheduling the task graph without considering any recomputations. For determining these values, the variable α is set to zero. The second schedule finds the best memory (M_{best}) and the worst performance (P_{worst}) values by setting the variable β to zero and performing task recomputations as much as possible during the scheduling process. If none of these schedules satisfy the requirements set by the designer, she can input new α and β values in the second step (the exploration step in Fig. 5). The important point is that the variables α and β has to be entered by the designer, and these values shape the final design in terms of memory and performance parameters. Having taken α and β as input, our approach starts with the schedule that uses recomputations very aggressively (i.e., $\beta=0$), and iteratively reduces the number of recomputations by selecting the tasks whose results will be stored in memory, which results in increased storage but brings better performance. This process is repeated until a design point that falls in the region of interest in Fig. 4 is found. If the final schedule does not satisfy the designer, she is asked to enter new α and β values and a new scheduling process is invoked.

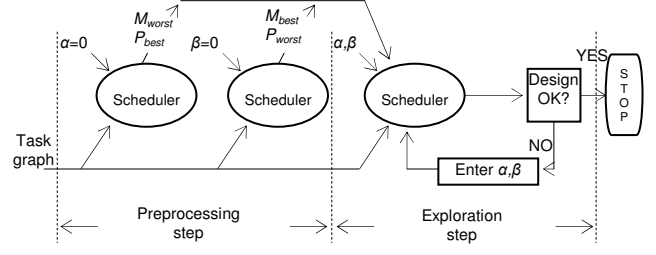


Fig. 5: The high level operation of the approach.

3.3 Preprocessing Step

In this section, we discuss the preprocessing step. The sketch of our recomputation-based algorithm is given in Algorithm 1. The scheduling algorithm we employ is similar to the ones used in high-level synthesis [9]. Before starting our scheduling process, we first map the tasks to processors in such a way that a task's execution time on the mapped processor is the minimum among all possible alternatives. We then find the earliest start time (EST) and the latest start time (LST) for every task in the task graph. The EST of task i is found based on the finish times of the task i 's predecessors. This can be expressed as follows:

$$EST_i = \max\{EST_j + d_j + ct(j, i)\}; \quad \forall i, j : (j, i) \in E, \quad (3)$$

Algorithm 1: Recomputation-based scheduling algorithm. M , P , and m represent the memory consumption, performance of the final design, and the available number of processors, respectively.

1. Find freedom and level of each task using Expressions (6) and (7);
2. Schedule the source vertex at step 1;
3. $current_level=1$;
4. Find the minimum memory requirement;
5. **while** (the terminal vertex is not scheduled) **do**
6. **for** ($p=1$ to m) **do**
7. Select task i , the task with the highest priority;
8. **if** ($\alpha=0$) **then**
9. Schedule task i ;
10. Store task i 's data in the memory;
11. **else if** ($\alpha \neq 0$) **then**
12. **if** (the data of task i from pred. j is not available) **then**
13. $i=recompute(i, j, current_level)$;
14. Schedule task i ;
15. Store task i 's data in the memory;
16. **end if**
17. **end if**
18. **end for**
19. Increment the level of non-scheduled tasks by one;
20. Adjust freedoms of non-scheduled tasks;
21. $current_level=current_level+1$;
22. **end while**
23. **if** ($\alpha=0$) **then**
24. $M_{worst}=graph_coloring()$;
25. **end if**
26. Find the bounds on α and β based on Expressions (1) and (2);
27. Enter α and β ;
28. **if** ($M < (1+\alpha)M_{best}$ && $P < (1+\beta)P_{best}$) **then**
29. Select a resident task to store its data in memory;
30. **else if** ($\beta=0$) **then**
31. **return** $schedule(M_{best}, P_{worst})$;
32. **else if** ($\alpha=0$)
33. **return** $schedule(M_{worst}, P_{best})$;
34. **end if**
35. **return** $schedule(M, P)$;

In this expression, d_j is the delay of task j and $ct(j,i)$ is the communication time (delay) between task i and task j . Specifically, the communication time is the total time for task i to write its output to the memory and for task j to read this data from the memory. (j,i) represents a dependence (the edge) between tasks i and j ; i.e., task j is the predecessor of task i in the task graph. We set the *EST* of the source vertex to one and, subsequently, we find the *ESTs* of the remaining tasks by using Expression (3). This step also returns the initial latency (*IL*) of the schedule, which is found using the following expression:

$$IL = \max\{EST_j + d_j\}; \quad \forall j. \quad (4)$$

The *LST* of each task can be determined by a reverse application of the method used to find the *ESTs*. That is, we can use the following expression:

$$LST_j = \min\{LST_i - d_i - ct(i, j)\}; \quad \forall i, j: (j, i) \in E \quad (5)$$

We set the *LST* of the terminal vertex to *IL* and determine the *LSTs* of the remaining tasks using Expression (5).

After having found the *ESTs* and *LSTs* for our tasks, we calculate their *freedom* (*F*). The freedom of a task, also known as its mobility or slack, can be defined as the time frame during which the task can be scheduled, which is also the difference between the *EST* and *LST* of the task. We determine the freedom of task i using the following expression:

$$F_i = LST_i - EST_i. \quad (6)$$

We also assign a level l to each task assuming that each task has one unit delay (this assumption is just for the level assignment). To do this, we set the level of the source vertex to one and then use the following expression to determine the level of task i :

$$l_i = \max\{l_j + 1\}; \quad \forall i, j: (j, i) \in E. \quad (7)$$

We then set the initial level to one and this also becomes the *current level* of the schedule. We schedule the tasks in the current level based on their priorities. The priorities of the tasks are determined based on their freedoms.

Specifically, a task with a lower freedom has a higher priority over a task with a higher freedom. In the current level, we may schedule some of the tasks based on the available number of processors in our target architecture (i.e., we can schedule only a maximum of m tasks in the current level, where m is the number of processors in our architecture). In this step, we check our two variables α and β , and if $\alpha=0$, we do not check the possibility of recomputations. The tasks that are not scheduled are moved

to the next level by incrementing their levels by one. At this point, we have to note that we may not be able to schedule the tasks at the best possible processors (for them) since there may be more than one task in the current level requiring the same processor. This is because there cannot be two tasks executing on the same processor at the same time. On the other hand, if we schedule the second task after the first one completes, then this can significantly increase the latency of the design. To prevent this, we schedule the task on its next best candidate processor, resulting in a small latency overhead (i.e., the task with the higher priority is mapped to its best processor for it and the next one is mapped to its best available one). After scheduling the tasks in the current level, we adjust the freedom of the remaining tasks. After this, we increment the current level by one. This process is iterated until the terminal task (vertex) is scheduled.

The next step, when $\alpha=0$, is to determine the memory consumption of the design. To do this, we use a graph coloring algorithm to decide which tasks' results can share the same memory space. First, we determine the lifetimes of all the task results based on the first time they are produced and the last time they are used by any of their direct successors. We use the following expressions to find the start (r_i^s) and end (r_i^e) times of the task i 's result:

$$r_i^s = e_i \text{ and } r_i^e = \max\{s_j\}, \quad (8)$$

where e_i represents the time at which task i completes its execution and s_j is the time at which task i 's successor, task j , starts its execution. After this, we find the minimum memory space consumption by employing a graph coloring algorithm, as we have discussed earlier.

Now, we explain the second scheduling performed in the preprocessing step where we introduce recomputations as much as possible. In this step, we set β to zero and use the scheduling algorithm explained above except that we consider recomputing the tasks during scheduling instead of storing their data in memory. We first bind the tasks to the processors such that a task's execution time on its processor is the minimum among all possible alternatives. Then, we assign the freedom and level of each task. To store the data manipulated by the tasks, we need to know the minimum memory bound. This can be found by analyzing the producer/consumer relationships between the tasks. The task that needs the most data from its predecessors determines the minimum memory space requirement (if all the tasks have the same memory requirements, the task with the maximum number of predecessors determines the minimum memory requirements). The tasks are then scheduled based on their priorities when their input data is available in the memory. If a task's input from any of its predecessors is not available, we call a recursive procedure to determine the recomputed tasks for the current task, and

subsequently increment the task's level by one to schedule it in the next level. The sketch of the recursive procedure used for this purpose is given in Algorithm 2. The recomputation of any task may trigger more recomputations since the data of this task may not be available in the current level. Then, our recursive procedure moves the non-scheduled tasks (these tasks may be the ones that have been scheduled in previous levels) to the next level. If any task is recomputed, we increment its *recomputation counter*, a counter that keeps the number of recomputations for a task, by one. We also store the *chain of recomputations* (if there is any) in an array. The recomputation counter for each task and the array in which we keep for the chain of recomputations together help us to select a task, and during scheduling, we store the result of this task instead of recomputing it when needed.

The second scheduling process (in the preprocessing step) is iterated until the terminal vertex is scheduled. We then return the resulting execution time of the schedule as our worst performance value and the resulting memory size as the best memory consumption value. If the returned performance and/or memory consumption values and the corresponding values for the schedule that does not use any recomputations are the same, we conclude that we do not have to investigate the solution space further, since we cannot have any solution which is different from these two. Otherwise, the designer proceeds with the exploration step, which is explained next.

3.4 Exploration Step

We now have the best and worst performance and memory consumption values at this point. By plugging these values into Expressions (1) and (2), we can determine the upper bound for the variables α and β . In the next step, the designer explores the solution space using different α and β values (see Fig. 5). In other words, she studies the tradeoffs between performance and memory space consumption.

After taking α and β values from the designer, our exploration step starts with the schedule where

Algorithm 2: *recompute(i, j, current_level)*. The recursive procedure for determining the set of tasks to be recomputed.

```

1.  if (the data of  $j$  in  $current\_level$  is available) then
2.       $recomputation\_counter_j$ ++;
3.      Store  $j$  in the recomputation array;
4.      return  $j$ ;
5.  else if (the data of predecessor  $k$  for  $j$  is not available) then
6.       $l=j+1$ ;
7.       $recompute(j, k, current\_level)$ ; }
8.  end if

```

recomputation is used very aggressively (i.e., $\beta=0$). At each step, we need to select a task (we refer to this task as the *resident task*) whose results will be stored in memory as long as it is needed by its successors (i.e., when the data of the resident task is stored in the memory, it remains there until we do not need it anymore). Note that, if the result of a task is stored in memory, the total memory consumption is increased. In this way, the exploration step selects a resident task at each step, and updates performance and memory space consumption values. To select the resident task, we can employ two different criteria in this study based on the characteristics of the embedded application under consideration. These selection procedures can be briefly described as follows:

Breaking the longest recomputation chain (called LRC): As we explained above, we store the chain of recomputations in an array. We determine the longest chain from this array, and then pick the task from the middle of this chain as the resident task to break the chain of recomputations. Assuming that the longest chain has k tasks, we can select the resident task r_i using the following expression:

$$r_i = \left\lceil \frac{k}{2} \right\rceil, \quad (9)$$

which gives us the task in the middle (in case where k is an odd number), or higher of the two tasks in the middle (in case where k is an even number) of the chain.

The most frequently recomputed task (called MFR): In this approach, we select the task that has been recomputed (in the preprocessing step) more than any other tasks as the resident task.

Depending on the embedded application at hand, one of these two selection criteria may result in a better performance value than the other. In our experiments, we compared both these schemes.

After the scheduling process completes, based on α and β values, the designer decides if the current schedule is the one she was looking for. If not, we proceed with new α and β values supplied by the designer, and the whole exploration step is repeated.

Now, let us revisit our motivational example in Section 2, and show how the user exploration step works. Recall that earlier we discussed the preprocessing step for this example (in Section 2) and found the best and worst performance and memory consumption values as $M_{best}=60$, $M_{worst}=100$, $P_{best}=29$, and $P_{worst}=34$. Also, the upper bounds for our α and β variables are found to be 0.67 and 0.17, respectively, using Expressions (1) and (2). Now, if we do not want to select any of the two schedules given in Fig. 2(a) and Fig. 3 for our design, we can change the values of

α and β parameters and try a new schedule. Suppose we input $\alpha=0.4$ and $\beta=0.1$. So, our maximum memory consumption (M) and execution time (P) values can be found as 84 and 31.9 using Expressions (1) and (2), respectively.

Recall that, in the schedule in Fig. 3, the tasks 1, 2, and 6 are recomputed. We select task 1 to store its value throughout its lifetime (note that there are two other alternatives). Storing task 1's result reduces the execution time from 34 units to 29 units of delay. However, it also increases the memory consumption by 20 units of memory, resulting in a total of 80 units of memory consumption. In Fig. 6, we give the resultant schedule for this choice. The performance and memory consumption values are within the desired bounds ($P=34$ and $M=80$).

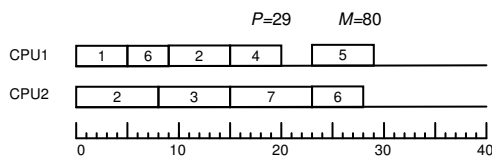


Fig. 6: The final schedule for the graph in Fig. 1(b) with $\alpha=0.4$ and $\beta=0.1$.

4 Experimental Results

In this section, we demonstrate the impact of our recomputation-based approach on task graphs generated from benchmarks. We also made experiments on automatically generated task graphs. Due to space limitations, we only present the results obtained from real benchmarks. For this purpose, we used the benchmarks *G721decode* and *G721encode* from MediaBench [10], *Mismatch_test* from Trimaran GUI [12], and *M-JPEG* from [13]. In Fig. 7, we give the memory consumption achievement and the performance overhead percentages of the scheduled benchmarks. To determine the best and worst performance and memory consumption values, we set α and β to zero. The bar charts in Fig. 7 show that our approach achieves 25.25% memory savings while tolerating 9.75% performance overhead on the average.

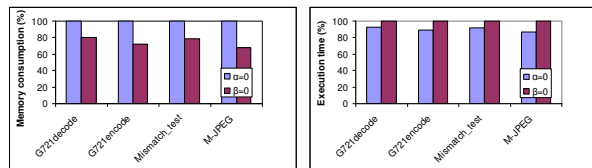


Fig. 7: The best and worst memory consumptions and execution times with the two scheduling alternatives for benchmarks: one with the best performance ($\alpha=0$), and the other with the best memory consumption ($\beta=0$).

5 Conclusions

This paper proposes an approach that uses task recomputation instead of memory storage, for select tasks of a given application, in an attempt to reduce memory space demand of embedded applications. The proposed approach operates under two user-specified parameters that control the tradeoff between performance degradation and storage demand. Our experiments with several task graphs mapped onto a parallel embedded architecture are very promising, and indicate that task recomputation can be a serious alternative to compression and lifetime analysis based approaches.

6 References

- [1] Y. Xie et al., "Code compression using arithmetic coding based variable-to-fixed coding," Proc. Data Compression Conference, pp. 382-391., 2003.
- [2] O. Ozturk et al. "Data compression for improving SPM behavior," Proc. the 41st Design Automation Conference, 2004.
- [3] S. Debray and W. Evans, "Profile-guided code compression," Proc. SIGPLAN '02 Conference on Programming Language Design and Implementation, 2002.
- [4] M. Strout et al., "Schedule-independent storage mapping in loops," Proc. of ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 1998.
- [5] V. Lefebvre and P. Feautrier, "Automatic storage management for parallel programs," Research report PRiSM 97/8, France, 1997.
- [6] F. Catthoor et al., Custom memory management methodology: Exploration of memory organization for embedded multimedia system design, Kluwer Academic Publishers, , 1998.
- [7] M. Kandemir et al., "Studying storage-recomputation tradeoffs in memory-constrained embedded processing," Proc. of Design Automation and Test in Europe Conference, 2005.
- [8] R. Giering and T. Kaminski, "Generating recomputations in reverse mode AD," In G. Corliss et al., editors, Automatic Differentiation of Algorithms: From Simulation to Optimization, Springer Verlag, Heidelberg, 2002.
- [9] <http://www.xilinx.com/products/virtex4/overview/processing.htm>
- [10] G. De Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill, 1994.
- [11] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", International Symposium on Microarchitecture, 1997.
- [12] www.trimaran.org
- [13] C. Erbas, S. C. Erbas, and A. D. Pimentel, "A Multiobjective Optimization Model for Exploring Multiprocessor Mappings of Process Networks", Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Co-design and System Synthesis, 2003.