Studying Storage-Recomputation Tradeoffs in Memory-Constrained Embedded Processing *

Mahmut Kandemir, Feihui Li, Guilin Chen, Guangyu Chen, and Ozcan Ozturk
Computer Science and Engineering Department
The Pennsylvania State University, University Park, PA 16802, USA
{kandemir, feli, guilchen, gchen, ozturk}@cse.psu.edu

Abstract

Fueled by an unprecedented desire for convenience and self-service, consumers are embracing embedded technology solutions that enhance their mobile lifestyles. Consequently, we witness an unprecedented proliferation of embedded/mobile applications. Most of the environments that execute these applications have severe power, performance, and memory space constraints that need to be accounted for. In particular, memory limitations can present serious challenges to embedded software designers. The current solutions to this problem include sophisticated packaging techniques and code optimizations for effective memory utilization. While the first solution is not scalable, the second one is restricted by intrinsic data dependences in the code that prevent code restructuring. In this paper, we explore an alternate approach for reducing memory space requirements of embedded applications. The idea is to re-compute the result of a code block (potentially multiple times) instead of storing it in memory and performing a memory operation whenever needed. The main benefit of this approach is that it reduces memory space requirements, that is, no memory space is reserved for storing the result of the code block in question.

1. Introduction and Motivation

One of the major challenges for embedded software writers is memory space limitation. While memory capacities attached to embedded systems keep increasing, the rate at which complexity and dataset sizes of embedded applications is increasing is much faster. Consequently, making best use of available memory space is becoming increasingly critical. This is true for diverse embedded application domains ranging from automobile control to disk processing. There are at least two current solutions to this memory problem. The first is to use sophisticated circuit/packaging techniques to cram as much memory as possible in as little area as possible. This solution is not scalable and is typically costly from both design and implementation perspectives. The second solution is to use software techniques to reduce memory space requirements by maximizing data reuse and exploiting data lifetime information. Recent studies explored this option in different directions that include both computation space and data space optimizations [8, 4, 5]. While the techniques in this group are effective in many embedded application domains as has been demonstrated by prior work, they are inherently limited by the intrinsic data reuse in the program and by potential overlap between lifetimes of different data structures. Consequently, in cases where there is not much data reuse to exploit and lifetimes of a large majority of data structures overlap with each other, such techniques cannot be very successful.

In this paper, we explore an alternate approach for reducing memory space requirements of embedded applications. The idea is to *re-compute* the result of a code block (potentially multiple times) instead of storing it in memory and performing a memory access whenever needed. In other words, we study the cases where re-computation can substitute for memory accesses. The main potential benefit of this approach is to reduce memory space requirements, that is, no space is reserved for storing the result of the code block in question. A potential disadvantage is the increase in execution cycles as re-computing a result each time it is required (in particular when the result has a high degree of locality) can be costly from the performance overhead viewpoint. However, this may not always be so since sometimes accessing a data generates misses in the on-chip storage, and visiting off-chip memory can cost tens of cycles (which also keeps increasing). In such cases, re-computation can bring performance benefits as well. Therefore, the performance impact of re-computation should be experimentally studied and quantified.

Focusing on array-intensive embedded applications that execute in memory-constrained systems, this paper makes

* This work was supported in part by NSF Career Award #0093082.

the following contributions:

• We propose two integer linear programming (ILP) based techniques to study maximum savings that could come from re-computation. The first ILP strategy is a static one in which the status of a code block (i.e., whether its results should be stored or re-computed when needed) is decided at exactly one point during optimization. In contrast, the dynamic scheme re-evaluates the storage versus re-computation decision multiple times.

• We propose and evaluate a heuristic approach, and compare it to the ILP-based solution. Our experimental results reveal that the proposed heuristic performs very well for most of the time.

• We explain how our approach can be extended to handle multi-level memories, to account for data lifetimes, and to reduce memory space requirements under performance constraints.

Our experimental evaluation with six embedded applications demonstrates that re-computation based ILP and heuristic solutions are beneficial from both memory space and performance perspectives.

The remainder of this paper is organized as follows. Section 2 presents the formulation for the problem of minimizing the number of execution cycles under memory space constraints. Section 3 explains the solutions we propose. Section 4 discusses several extensions to our baseline approach. Finally, Section 5 concludes the paper with a summary.

2. Problem Formulation

We study our memory space optimization problem at two levels: *static* and *dynamic*. In the static approach, we decide for each computation whether its results should be stored in the memory or not at exactly one point during optimization, and this decision is maintained throughout the execution. More specifically, once we decide that the results of a computation will not be stored, we stick to this decision until the end of execution; i.e., each time we need the result of that computation, we re-compute it. In contrast, in the dynamic scheme, we consider the possibility of storing the results of a computation several times. Consequently, a computation whose results are not stored at the first time it is encountered can still have its result stored at a later step during execution (when it is re-computed). Clearly, this potentially brings additional benefits at the cost of additional implementation complexity. This paper evaluates both these schemes.

Our approach works on a graph called *flow graph*, which captures the *producer-consumer* relationship among different code blocks in the application. Each node of this graph is a code fragment whose granularity can be tuned. For example, each node can be an entire procedure, a loop nest, or a sub-nest (which is a small loop that contains a subset of the iterations of a loop nest in the application). The edges between the nodes indicate potential data flow between them. Specifically, a directed edge (arc) from a node to another indicates that the former generates a result which is subsequently used by the latter. It should be noted, though, this is a conservative approximation of real execution (with a particular input) since the data transfer implied by an edge may not be materialized due to control flow. We use $\mathcal{T}$ to denote the set of nodes in the flow graph, and assume that there is a unique terminal node in the flow graph. If this last assumption fails, we create a terminal node and connect it to all the other nodes in the graph that have no successors. We use N to denote the number of nodes in the flow graph being optimized. We also assign a *level* (starting with 1) to each node in the graph. Specifically, the level of node i is set to L+1, where L is the largest level among all the nodes from which we have an edge to node i. We use M to indicate the number of levels in the flow graph. In processing the flow graph for optimization purposes, we also use the term *step* to denote the processing of a level, i.e., each step corresponds to processing of a particular level in the graph.

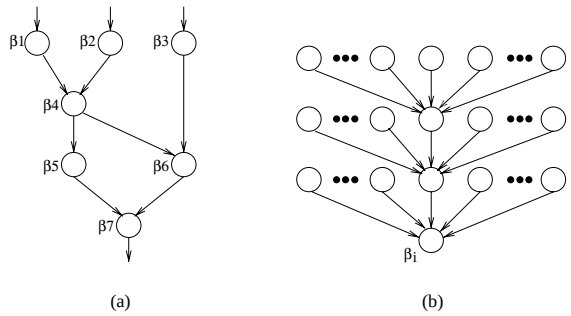

Figure 1. Two example flow graphs.

Before going into our discussion of the static and dynamic schemes, let us study an example scenario that illustrates the tradeoffs between storage requirements and re-computation overheads. We consider the flow graph shown in Figure 1(a), and focus on two simple scenarios for illustrative purposes. In the first scenario, we assume that only the results of β_1, β_2, and β_3 are stored in memory, and results of all other nodes in the graph are re-computed when they are needed. In the second scenario, the results of nodes β_1, β_2, β_3, β_4, and β_6 are stored in memory, whereas nodes β_5 are β_7 are re-computed. In both the scenarios, it is assumed that the results of nodes β_4, β_5, β_6, and β_7 are needed as output (i.e., none of them is temporary or intermediate data). In the first scenario, computing β_4 requires memory accesses for the results of β_1 and β_2. Then, computing β_5 requires re-computation of β_4 since the result of the latter is not stored in memory.[1] Note that this

	β_1	β_2	β_3	β_4	β_5	β_6	β_7
β_1	C, M_w	-	-	-	-	-	-
β_2	-	C, M_w	-	-	-	-	-
β_3	-	-	C, M_w	-	-	-	-
β_4	M_r	M_r	-	C	-	-	-
β_5	M_r	M_r	-	R	C	-	-
β_6	M_r	M_r	M_r	R	-	C	-
β_7	M_r	M_r	M_r	R	R	R	C

	β_1	β_2	β_3	β_4	β_5	β_6	β_7
β_1	C, M_w	-	-	-	-	-	-
β_2	-	C, M_w	-	-	-	-	-
β_3	-	-	C, M_w	-	-	-	-
β_4	M_r	M_r	-	C, M_w	-	-	-
β_5	-	-	-	M_r	C	-	-
β_6	-	-	M_r	M_r	-	C, M_w	-
β_7	-	-	M_r	R	R	M_r	C

Figure 2. Two different scenarios for an example data-flow graph. The entry in a cell (β_x, β_y) indicates the necessary computation (C), re-computation (R), memory read (M_r), and memory write (M_w) of/for β_y in order to compute the result of β_x.

re-computation requires fresh accesses to memory for the results of β_1 and β_2. Similarly, computing β_6 requires re-computation of β_4. The remaining computations for this scenario as well as the computations/memory accesses for the second scenario can be identified using a similar strategy. The tables in Figure 2 give, for both the scenarios, the necessary memory accesses and re-computations. The entry in a cell (β_x, β_y) in these tables indicates the necessary computation (C), re-computation (R), memory read (M_r), and memory write (M_w) of/for β_y in order to compute the result of β_x. Note that, while not storing the result of a computation can save memory space, it can also trigger lots of re-computations, whose performance implications should be accounted for. An important question then is to reduce execution cycles (improve performance) under a memory capacity constraint. The tables in Figure 2 also reveal an interesting behavior of re-computation. Suppose that a particular node β_i has p predecessor (β_{j_1} through β_{j_p}). Whenever we want to compute β_i, we need to re-compute all β_{j_k}s whose results have not been saved in memory. This in turn requires re-computation of all predecessors of β_{j_k}s whose results have not been saved, and so on. For example, computation of β_i in Figure 1(b) can trigger a lot of re-computations depending on how many of the preceding nodes have their results stored in memory. Obviously, this chain of re-computations can potentially increase overall execution cycles.

1 For example, if β_4 contains a statement such as `b=a+3` and β_5 contains a statement such as `c=b-2`, re-computing β_4 (during the evaluation of β_5) would generate a computation like `c=(a+3)-2` as `b` is not stored in memory.

2.1. Static Scheme

Let β_i represent the ith computation in the flow graph of the application, where $1 \leq i \leq N$. The memory space occupied by storing the results of β_i is denoted by $S(\beta_i)$, and the computation (or re-computation) time (in cycles) for β_i is expressed using $C(\beta_i)$. We assume that writing the result of β_i into memory and accessing the stored result from memory take $M_w(\beta_i)$ and $M_r(\beta_i)$ cycles, respectively. We use the following 0-1 variable[2] to indicate whether β_i is (decided to be) stored in memory or not (when we process the point where it needs to be computed the first time):

$$\alpha_i = \begin{cases} 1, & \text{if } \beta_i \text{ is stored in memory} \\ 0, & \text{otherwise} \end{cases}$$

Based on these 0-1 variables, the total computation (or re-computation) cost of β_i can be computed as follows:

$$\mathcal{C}(\beta_i) = \alpha_i M_w(\beta_i) + C(\beta_i) + \sum_{\beta_j \in pred(\beta_i)} \mathcal{D}(\beta_j),$$

where $pred(\beta_i)$ denotes the set of immediate predecessors of node β_i. The first term in this expression indicates the total cost of *accessing* the results of all the predecessors of β_i. The second component gives cost of writing its result to the memory (if it is to be stored in memory), and the last term gives the cost of computing β_i itself (note that, irrespective of whether the result of β_i is stored in memory or not, it needs to be computed). For a given node β_i, $\mathcal{D}(\beta_i)$ gives the cost of accessing its result for the computation of some other node. It should be noticed that such an access can involve re-computation (depending on whether its result has already been stored in memory or not), which can in turn trigger cascade re-computations. Consequently, we have the following expression for $\mathcal{D}(\beta_i)$:

$$\mathcal{D}(\beta_i) = \alpha_i M_r(\beta_i) + (1 - \alpha_i)\mathcal{C}(\beta_i).$$

One can then formulate the overall cost of executing the entire application as the computation cost of the terminal node (denoted β_s). In mathematical terms:

$$\mathcal{C}_{total} = \mathcal{C}(\beta_s),$$

On the other hand, the total memory space consumption ($\mathcal{S}_{total}$) can be calculated as:

$$\mathcal{S}_{total} = \sum_{\beta_i \in \mathcal{T}} \alpha_i S(\beta_i).$$

It is to be observed that this calculation implicitly assumes that once a memory space is allocated for the result of β_i, this allocation is retained until the end of execution. While this certainly simplifies the formulation of the problem, it would be more realistic to assume that, once the result of β_i is not needed by any other node in the flow graph, its memory is deallocated and (potentially) recycled for storing

2 A 0-1 variable is one that takes a value of 0 or 1.

the results for other nodes. Constructing the expression for the memory space consumption in such a case requires taking into account lifetimes of (the results of) the flow graph nodes, and is discussed in Section 4.

Assuming that $\mathcal{S}_{max}$ is the maximum allowable memory space consumption (i.e., storage capacity), we are now ready to formulate our optimization problem in formal terms:

"Minimize $\mathcal{C}_{total}$ under $\mathcal{S}_{total} \leq \mathcal{S}_{max}$."

The 0-1 variables are the only unknowns in this formulation, and the objective of any exact or heuristic solution would be to determine the values of these variables. In Section 3, we will present both exact and heuristic solutions to this optimization problem. Note that, it is also possible to define the dual of this problem, that is, minimizing storage occupancy under performance constraints.

2.2. Dynamic Scheme

In this scheme, it is possible to store the result of β_i at each time step k, where $1 \leq k \leq M$. Consequently, α_i variables employed in the static scheme are not sufficient. Instead, we parameterize these variables with both i and k. In mathematical terms:

$$\alpha_{i,k} = \begin{cases} 1, & \text{if } \beta_i \text{ is stored in memory at step } k \\ 0, & \text{otherwise} \end{cases}$$

We need to change our cost formulation as well. Specifically, $\mathcal{C}(\beta_i, k)$ denotes the cost of computing (or re-computing) β_i at step k. There are two reasons for such a parameterization. First, it is possible that β_i is not stored in memory at step k' but later is decided to be stored (say, at step k'' where $1 \leq k' < k'' \leq M$), when for example the available memory capacity changes dynamically. Second, due to the same reason, it is possible that we first decide to store it in memory but later delete it (and re-compute it whenever necessary following this deletion point). This second case will be detailed in Section 4. Our formulation is:

$$\mathcal{C}(\beta_i, k) = \alpha_{i,k} M_w(\beta_i) + C(\beta_i) + \sum_{\beta_j \in pred(\beta_i)} \mathcal{D}(\beta_j, k-1).$$

$$\mathcal{D}(\beta_i, k) = \alpha_{i,k} M_r(\beta_i) + (1 - \alpha_{i,k}) \mathcal{C}(\beta_i, k).$$

Note that, in computing $\mathcal{C}(\beta_i, k)$, we use $\mathcal{D}(\beta_j, k-1)$ since whatever we decide for β_j in the previous step (step $k-1$) will dictate the cost of accessing it at the current step (step k). In addition to these formulations, we need an additional constraint if we assume that once the result of a node is decided to be stored, the memory space allocated for it is never deallocated (i.e., once stored, always stored):

$$\alpha_{i,k} - \alpha_{i,k-1} \geq 0.$$

Based on these constraints, we can formulate the overall cost of executing the entire application as:

$$\mathcal{C}_{total} = \mathcal{C}(\beta_s, M),$$

where β_s is the terminal node in the flow graph. On the other hand, the total memory space consumption at step k ($\mathcal{S}_{total,k}$) can be calculated as:

$$\mathcal{S}_{total,k} = \sum_{\beta_i \in \mathcal{T}} \alpha_{i,k} S(\beta_i).$$

An important advantage of the dynamic scheme is that it can adapt itself well to the dynamic changes in available memory space. For example, in a multi-programmed embedded environment, depending on the relative criticalities of simultaneously running applications, the memory space allocated to a particular application can change from one execution point to another. As a result, a node (β_i) that has been decided not to be stored in memory (due to space constraints) can later be stored in memory when more space becomes available. Consequently, in this dynamic scenario, it might make more sense to talk about time step-wise memory capacity limit, that is, $\mathcal{S}_{max,k}$, which denotes the maximum allowable memory consumption at step k. Obviously, the storage consumption at each step (that is, $\mathcal{S}_{total,k}$) must be smaller than or equal to $\mathcal{S}_{max,k}$. Therefore, our performance optimization problem under memory constraint can be expressed as follows:

"Minimize $\mathcal{C}_{total}$ under $\mathcal{S}_{total,k} \leq \mathcal{S}_{max,k}$, where $1 \leq k \leq M$."

3. Proposed Solutions

In this section, we propose two different solutions to our memory space management problem. The first solution is based on zero-one integer linear programming (ILP). ILP is suitable for our particular memory utilization problem since our formulation of the problem described above lends itself to ILP computation very well. In this formulation, our objective is to determine α_i (in the static scheme), or determine $\alpha_{i,k}$ for each β_i and time step k (in the dynamic scheme) – where $1 \leq i \leq N$ and $1 \leq k \leq M$. However, one point needs to be clarified here. While the static scheme operates on a one-dimensional space (iterated by i), the dynamic scheme operates on a two-dimensional scheme (iterated by both i and k). This can potentially increase the number of 0-1 variables dramatically, making an ILP-based solution infeasible in some cases. However, it must be observed that, many of the (i, k) pairs are not feasible in practice anyway (for a given application), and the costs expressions involving them do not need to be constructed at all. This is because, for a given flow graph node represented by β_i, the possible time steps that need to be considered are between the step where it is first encountered in processing the graph and the last step of the execution. In addition, one might be able to further shrink the feasible search space by taking into account the lifetimes of (the results of) the nodes, as will be further elaborated in Section 4. This ILP-based solution can be useful in its own right, but it can

also be used for evaluating the quality of the heuristic solutions. That is, once a heuristic solution has been developed, the results it generates can be compared to those obtained through the ILP-based method. This can help one evaluate the quality of the heuristic in question, and refine it as necessary. Still, for very large problem sizes/applications, one may need to resort to fast heuristic solutions.

Our second solution to this optimization problem is a heuristic based on a greedy approach. In this approach, we consider all the nodes of the flow graph, and decide whether the result of each node should be stored in memory or not. To guide this decision, one obviously needs a metric or a cost function. Our approach to this problem can be explained as follows. For each node, we compute the number of *potential consumers*, i.e., the set of nodes that will use the result generated (produced) by this node directly or indirectly (again note that a potential consumer may not be an actual consumer at runtime due to dynamic flow of control). After this, we select the node with the *maximum* number of such consumers and decide to store its result in memory provided that doing so does not lead to consuming more memory space than the available capacity. In doing so, we also keep track of the memory space consumed (occupied) so far. In the next step, we select, from the remaining nodes, the one with the maximum number of consumers and schedule it if the memory capacity constraint allows doing so. We continue this way until all the nodes have been processed or the available memory capacity has already been reached. If this latter case occurs, the results of the remaining (unprocessed) nodes are decided not to be stored in memory; that is, they are marked for re-computation. The goal behind this heuristic is to store the most frequently used results as much as possible, thereby utilizing the available memory space in the most effective way. Notice that this heuristic tends to store the results of the nodes that reside in the upper portions of the flow graph as they are typically the ones whose results are used by many nodes (consumers).

Figure 3 gives a sketch of this heuristic solution. Considering the flow graph in Figure 1 once more and assuming that all the nodes in this graph require a memory space of the same size to store their results, and that the on-chip memory can hold the result of only one node, this heuristic would select β_4 among β_4, β_5, and β_6 (under the assumption that the results of β_1, β_2, and β_3 are already in memory), and store its result in memory. Note that, while this heuristic is a static one, it is easy to convert it to a dynamic one. In the dynamic version of the heuristic algorithm, one can take lifetimes of individual graph nodes into account, and also performs transfers between $\mathcal{G}$ and $\mathcal{T}$, depending on the dynamic variations in available memory size. This can in turn lead to better utilization of available memory space.

NOTATION:
$\mathcal{T}$: the set of nodes in the flow graph
$\mathcal{G}$: the set of nodes whose results are decided to be stored
$\mathcal{S}_{total}$: the current memory space occupation
$\mathcal{S}_{total}$: the maximum allowable memory occupation

ALGORITHM:
compute the consumer set CS_i for each node β_i
$\mathcal{G} = \emptyset$
$\mathcal{S}_{total} = 0$
order the nodes according to non-increasing values of $|CS_i|$
while (there is still available space in memory and $\mathcal{T} \neq \emptyset$)
{
select the node β_i from $\mathcal{T}$ with the largest $|CS_i|$
if ($\mathcal{S}_{total} + S(\beta_i) \leq S_{max}$) then
{
remove β_i from $\mathcal{T}$
add β_i to $\mathcal{G}$
$\mathcal{S}_{total} = \mathcal{S}_{total} + S(\beta_i)$
}
}

Figure 3. The sketch of the heuristic algorithm (the static approach).

4. Extensions

In this section, we discuss three important issues. The first issue is about making our approach more effective by taking into account lifetimes of data. Next, we discuss how our baseline approach can be extended to work with multi-level memories.

So far, we have assumed that when the results of a node is stored in memory, it remains there until the end of execution. In many cases, this may not be necessary as we can deallocate the memory space as soon as the result of that node is not needed anymore by any of the remaining nodes, i.e., it is *dead*. The time frame between the generation of the result and the point at which it is dead (i.e., it will not be needed in the rest of execution) is called its *lifetime*. It is to be noted that the lifetime for a given data (the result of a node) can be identified (conservatively) by analyzing the flow-graph. Identifying lifetimes means that, for each node β_i, we need to determine the valid k values (where $1 \leq k \leq M$). It does not make sense to reserve a space for the result(s) of β_i before we reach it during evaluation. And similarly, it is possible to save memory space by deallocating the memory space allocated to β_i once it is last used. Therefore, for each β_i, one can have the following bounds for k: $k_{min,i} \leq k \leq k_{max,i}$. Note that, $k_{max,i} - k_{min,i}$ gives the region in which the result of β_i is live. Let $\mathcal{L}_i$ refer to the lifetime of node β_i. We can change the memory space consumption expression in the static scheme as follows:

$$\mathcal{S}_{total} = max\{ \sum_{\beta_i \in \mathcal{T}'} \alpha_i S(\beta_i)\}.$$

In this expression, $\mathcal{T}'$ refers to a subset of $\mathcal{T}$ (which represents all nodes in the flow graph) with the property that life-

times of all the nodes in $\mathcal{T}'$ overlap with each other. The *max* expression finds the set $\mathcal{T}'$ such that the cumulative memory space occupied by the results of all the nodes in it is the largest among all possible alternative $\mathcal{T}'$s. This can be incorporated into our ILP formulation by constraining the range of k for each β_i and by dropping the constraints $\alpha_{i,k} - \alpha_{i,k-1} \geq 0$ from consideration.

We now discuss how our approach can be extended to work with multi-level memories. While our discussion mainly focuses on a two-level memory, it is straightforward to extend it to memory hierarchies with larger number of levels. We denote the first level and the second level memory using L1 and L2, respectively. Typically, the first level memory is smaller, faster, more power-efficient, and more expensive (in terms of dollars/bit) than the second level memory. We mainly focus on the static scheme here but it is not difficult to extend this description to the dynamic scheme. We first need some modifications to the definition of our 0-1 variables. Specifically, the new variables should distinguish between the two memory levels:

$$\alpha_{i,L1} = \begin{cases} 1, & \text{if } \beta_i \text{ is stored in L1 memory} \\ 0, & \text{otherwise} \end{cases}$$

$$\alpha_{i,L2} = \begin{cases} 1, & \text{if } \beta_i \text{ is stored in L2 memory} \\ 0, & \text{otherwise} \end{cases}$$

An important design decision needs to be made about whether two copies of the same data (one in L1 and the other in L2) are allowed or not. If multiple copies are not allowed, we need the following additional constraint:

$$\forall \beta_i : \quad \alpha_{i,L1} + \alpha_{i,L2} \leq 1.$$

This constraint can easily be included into the ILP-based formulation. Based on these $\alpha_{i,L1}$ and $\alpha_{i,L2}$ variables, new cost formulations are as follows:

$$\begin{aligned} \mathcal{C}(\beta_i) &= \alpha_{i,L1} M_{wL1}(\beta_i) + \alpha_{i,L2} M_{wL2}(\beta_i) + C(\beta_i) \\ &\quad + \sum_{\beta_j \in pred(\beta_i)} \mathcal{D}(\beta_j). \\ \mathcal{D}(\beta_i) &= \alpha_{i,L1} M_{rL1}(\beta_i) + \alpha_{i,L2} M_{rL2}(\beta_i) \\ &\quad + (1 - (\alpha_{i,L1} + \alpha_{i,L2}))\mathcal{C}(\beta_i). \end{aligned}$$

In this expression, M_{wL1} and M_{wL2} refer to memory write costs (in terms of execution cycles) for L1 and L2, respectively. Similarly, M_{rL1} and M_{rL2} denote the corresponding memory read costs. Then, the overall cost can be expressed as:

$$\mathcal{C}_{total} = \mathcal{C}(\beta_s),$$

where β_s is the terminal node in the graph. Finally, the total memory space consumptions in memories L1 and L2 can be computed as:

$$\mathcal{S}_{total,L1} = \sum_{\beta_i \in \mathcal{T}} \alpha_{i,L1} S(\beta_i).$$

$$\mathcal{S}_{total,L2} = \sum_{\beta_i \in \mathcal{T}} \alpha_{i,L2} S(\beta_i).$$

Consequently, assuming that $\mathcal{S}_{max,L1}$ (resp. $\mathcal{S}_{max,L2}$) is the maximum allowable memory space consumption (i.e., storage capacity) for memory L1 (resp. L2), we can formulate our optimization problem for two-level memory in formal terms:

> "Minimize $\mathcal{C}_{total}$ under $\mathcal{S}_{total,L1} \leq \mathcal{S}_{max,L1}$ and $\mathcal{S}_{total,L2} \leq \mathcal{S}_{max,L2}$."

We also want to mention that there are other possible memory constraints when a multi-level memory hierarchy is considered. For example, instead of the one above, one could try to minimize $\mathcal{C}_{total}$ under $\sigma_1 \mathcal{S}_{total,L1}$ + $\sigma_2 \mathcal{S}_{total,L1}$, $\leq S_{max}$ where σ_1 and σ_2 are positive constants that satisfy $\sigma_1 + \sigma_2 = 1$.

5. Conclusions

Most embedded systems operate under tight memory and power constraints. Therefore, their performance requirements must be carefully balanced against their memory space requirements and power consumption. While state-of-the-art code/data optimization techniques try to make best use of available memory space, they are inherently restricted by available data reuse. The work presented in this paper takes an alternate approach where it employs recomputation instead of memory accesses if doing so is beneficial from performance and/or memory space occupancy viewpoints.

References

[1] T. M. Austin. The SimpleScalar/ARM Toolset. http://www.eecs.umich.edu/∼taustin/simplescalar

[2] F. Catthoor et al. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design.* Kluwer Academic Publishers, June 1998.

[3] J. Huang and D. Lilja. Exploiting Basic Block Value Locality with Block Reuse. In *Proc. the Fifth International Symposium on High Performance Computer Architecture*, Orlando, Florida, 1999.

[4] M. Kandemir et al. Dynamic Management of Scratch-Pad Memory Space. In *Proc. the 38th Design Automation Conference,* Las Vegas, NV, June 2001.

[5] P. R. Panda et al. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *Proc. European Conference on Design and Test*, 1997.

[6] A. Sodani and G. Sohi. Dynamic Instruction Reuse. In *Proc. the 24th International Symposium on Computer Architecture,* Denver, Colorado, 1997.

[7] R. P. Wilson et al. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers, *ACM SIGPLAN Notices,* 29(12), December 1994, pp. 31–37.

[8] L. Wang et al. Optimizing on-chip memory usage through loop restructuring for embedded processors. In *Proc. 9th International Conference on Compiler Construction*, March 30–31 2000, pp. 141–156, Berlin, Germany.