Task Response Time Optimization Using Cost-Based Operation Motion

Abdallah Tabbara  
EECS Department  
University of California at Berkeley  
Berkeley, CA 94720  
+1-510-643-5187  
atabbara@eecs.berkeley.edu

Bassam Tabbara  
EECS Department  
University of California at Berkeley  
Berkeley, CA 94720  
+1-510-643-5187  
tbassam@eecs.berkeley.edu

Alberto Sangiovanni-Vincentelli  
EECS Department  
University of California at Berkeley  
Berkeley, CA 94720  
+1-510-642-4882  
alberto@eecs.berkeley.edu

ABSTRACT

We present a technique for task response time improvement based on the concept of code hoisting from the high-level synthesis domain. Relaxed operation motion is a simple yet powerful approach for performing safe and useful operation motion from heavily executed portions of a design task to less visited segments. Here we introduce our algorithm, then we show how it differs from other code motion approaches, and it can be applied to the embedded systems domain. Results of our investigation indicate that cost-guided operation motion has the potential to improve task response time significantly.

Keywords
Response time, optimization, (cost-based) code motion.

1. INTRODUCTION

Embedded controllers are found in many of the components we interact with daily for example: cell phones, pagers, security devices, and the like. Power, size, and efficiency are a few of the design considerations that must be taken into account when developing such complex systems, to say nothing of cost and reliability of these devices. We in the “hardware/software co-design” community have been working in recent years towards methodologies and approaches aimed at addressing this design complexity in order to make better design tools for developers who in turn are attempting to deliver a better product to the consumer.

Increasingly synthesis is playing a crucial role in our methodologies that increasingly advocate abstraction, decomposition, and refinement [12]. If synthesis starting from the high levels of abstraction is to be accepted by designers, its quality must be adequate for the task. To that end, optimization of the design before and after synthesis becomes essential for the success of these design paradigms.

In this work we focus primarily on improving task runtime by performing pre-synthesis optimizations. We deal with computation or operation motion as one such valuable technique that must permit rapid optimization that needs to be reflected in better output quality with respect to the runtime metric after synthesis.

We assume a heterogeneous control-dominated embedded system target application, and an initial functional decomposition and adequate front-end that capture the design as a network of Extended Finite State Machines (EFSM’s) as in [13]. Since we will be dealing with single task optimization we make no assumptions on the model of computation that governs the composition of tasks in the system as a whole.

2. Related Work

There is a body of work on code motion (hoisting) from the software (high-level synthesis) domain(s). The goal of code motion is to avoid unnecessary re-computations at runtime [1]. The creation of temporary variables (i.e. registers) to hold these computations typically improves runtime for most target architectures. Code must be relocated to valid program points and this movement must be safe, in the sense that it must not change what the program flow is intended to compute. The main strategy for code motion is that of moving operations as early as possible in the program as in [9] and [5].

In practice code movement to the earliest program points can create pressure on the target architecture resources for e.g. because of register “spills”. A more practical approach involves also performing temporary lifetime minimization as in [7].

Knoop’s approach is the best-in-class approach for code motion since it involves unidirectional approaches in the program flow where reducible programs can be dealt with in $O(n \log (n))$ bit-vector steps (see [1]) where $n$ is the number of statements in the program in contrast to $O(n^2)$ complexity for previously known approaches. Hailperin in [3] extended Knoop’s approach to incorporate cost into the code motion process. However, the cost metric is based on individual operations (i.e. $*, +, \ldots$) and does not account for the frequency of execution of program portions. The approach’s goal is to position the instructions in (safe) positions in the program where the context possibly permits simplification of the particular operation through constant folding, or operation strength reduction for example.

Castelluccia et. al. [3] used runtime cost to optimize protocols but the techniques they applied were based mainly on node reordering at the CDFG for synthesis level, we use similar cost guidance to our operation motion technique.

3. Our Contribution and Overview

Our work incorporates cost into operation motion. The cost is obtained from a task level static analysis to identify the most
frequently visited segments of the task’s behavioral description. Our code motion step itself is also fast; it has a time complexity of \(O(n)\) in the number of statements in the description. To achieve this simplicity, however, we give up slightly on size where for a brief period after this step the space needed is \(O(n^2)\).

We also rely on code motion being part of a general optimization framework; in particular we take advantage of previous computations within the flow of reachable variable definitions, and reached uses of variables which has a time complexity of \(O(n^2)\). So, the framework’s complexity (i.e. \(O(n^2)\)) is what dominates the overall complexity. Of course, for this increase in complexity we can get much better optimization results than [7] since code motion is applied to all candidate operations at once and is tempered by other data flow and control analysis and optimization steps.

As will be described in the sequel, the approach is therefore much simpler (conceptually and in practice) than other approaches as it tackles operation motion indirectly, and still performs the job adequately as part of a comprehensive multi-step data flow and control optimization approach [13]. Our approach, dubbed Relaxed Operation Motion (RCM) because it accomplishes its objective indirectly, is also specialized to the embedded system domain where we are constrained by I/O schedule preservation, but where we benefit from the user’s insight by soliciting assistance in the cost estimation mechanism since embedded systems have a predictable (or pre-conceived typical) behavior.

We have implemented our approach in the POLIS public domain co-design tool [2], and have used the synthesis engines therein for output generation. Our results are reported for software synthesis.

4. Intermediate Design Representation

We briefly describe our intermediate design representation that permits us to perform task optimization at the high level (adapted from [13] to make this paper self-contained). We use an implementation-independent I/O-scheduled task representation referred to as Function Flow Graph (FFG) equivalent to the EFSM representation, and quite similar to the classical CFG from the software domain.

4.1 Function Flow Graph (FFG)

FFG is the task representation used for design analysis and optimization. Each EFSM state is represented as a collection of nodes, and edges represent control flow. This flow graph is the data structure on which the task control flow analysis is performed, and data flow information is gathered.

**Definition 1:** A Function Flow Graph (FFG) is a triple \(G = (V, E, N_0)\) where

i. \(V\) is a finite set of nodes

ii. \(E = (x, y)\), a subset of \(V \times V\), is an edge from \(x\) to \(y\) where \(x \in \text{Pred}(y)\), the set of predecessor nodes of \(y\).

iii. \(N_0 \in N\) is the start node corresponding to the EFSM initial state.

iv. Operations are associated with each node \(N\).

Operations consist of TEST’s performed on the EFSM inputs and internal variables, and ASSIGN’s on the EFSM outputs and internal variables. Operations are “un-ordered” per se as long as data dependency and execution semantics are preserved.

4.2 Task Optimization Flow

We perform data flow and control optimization at the design representation level as shown in Figure 1. The purpose of the approach is two-fold:

a) Raise the abstraction level, and allow optimization to be reflected in both hardware and software synthesis, and

b) Incorporate powerful classical data flow and control optimizations that have a considerable potential for improving the quality of the synthesized output.

**Figure 1. Data Flow and Control Optimization Flow**

In order to implement task optimizations we have developed an optimizer that examines the FFG in order to statically collect data flow and control information of the task under analysis using an underlying data flow analysis framework [5].

The EFSM in FFG form is shown in Figure 2 in state tree form for a simple example. A DAG state form is also available where FFG nodes are “shared” between states, but we will limit our presentation in this paper to the former. Note that the FFG unlike a classical CFG is I/O-scheduled (performed by the “front-end” input language e.g. Esterel [2]); the FFG is aware of the states, and the optimization itself is broken into 2 phases:

a) Architecture Independent phase: The FFG is analyzed and optimized as a sequence of operations as in the classical software optimization approaches except that I/O is preserved (operations with inputs and outputs have specialized handling)

b) Architecture Dependent phase: The I/O schedule and the state assignment are taken into account, and operations within states are optimized followed by an allocation of registers and computations step.

**Figure 2. EFSM in FFG Form: A Simple Example**
5. Illustrative Example
In order to illustrate our RCM approach we have adapted Knoop’s “motivating example” from [7] as shown in Figure 3, and made it reactive by adding inputs and outputs, and a loop from the final node $S_{10}$ back to $S_1$ so that the system is running continuously. Variables $a$, $b$, $c$ are declared internal, and initialized in $S_1$ to a sampled input value, $x$, $y$, and $z$ are declared as outputs and therefore “fixed” to their respective states since we always preserve I/O traces before and after the optimization. As in [7], our goal is to eliminate the redundant needless runtime re-evaluation of the $a + b$ operation.

We will focus our discussion on the nodes $S_8$ and $S_9$ since as we will see in the sequel they have the most cost, so we will try to relocate the aforementioned addition operation to other less costly nodes.

Figure 3. Illustrative Example (from [7])

6. (Cost-guided) Relaxed Operation Motion
Our operation motion approach is part of a comprehensive optimization approach (overall complexity is $O(n^2)$ in the FFG nodes), which consists of 4 steps performed in sequence:

1) Data Flow and Control Optimization: is a sequence of steps that optimize the FFG representation [13].

2) Reverse Sweep: is the optimization step that we are mainly addressing in this paper where code is relocated from one or more FFG nodes to others. This step can either follow the as early as possible approach, or be cost-guided. It consists of “indirect” operation motion through:

a) Dead operation addition: where operations are added to selected (all or based on cost) FFG nodes.

b) Normalization and Available Operation Elimination: This optimization step effectively replaces the code motion candidates from the targeted FFG nodes to other less costly nodes as a result of step (a).

c) Dead Operation Elimination: removes the useless additions performed in (a).

3) Forward Sweep: is optional. It tries to minimize the lifetime of temporaries by pushing them as close as possible to their use. It is similar in concept to step 2 but is based on available operation addition.

4) Final Optimization Pass: performs the final clean up.

It can therefore be seen that our approach comes “naturally” in an optimization framework [1], which permits us to use relatively simple techniques to accomplish the task. The result after the first step (1) and (2a) are shown in Figure 4. Figure 5 shows the result after (2), and (4), step (3) is not applied here. Note that in the final result the redundant computations in $S_8$ and $S_9$ are indeed relocated up to $S_1$ (earliest position, forward sweep not applied).

In this example, the final result is 60% better than Knoop’s Lazy Code Motion of [7] if we count the remaining addition operations after RCM.

Figure 4. Result After Dead Addition

Figure 5. Result after Available Elimination

7. Cost Estimation
7.1 Background
The exact number of times a certain part of a program is executed can be determined once each branch probability in the program is known [8]. It can be shown that the number of times each basic block (and by analogy each FFG node) is executed can be calculated by solving a system of n linear equations, where n is the number of basic blocks ([11]), if the probabilities of control passing from one block to the next is given [14]. This of course is
a generalization of branch prediction, which only determines the most probable outcome of a branch [3]. The probabilities of all the TEST’s outcomes in the task are requested from the designer in an interactive fashion before the estimation and subsequent optimization takes place.

A Markov chain can be used to model and then compute statically the probabilistic control flow execution as described in [14] where it is also shown that this method is quite close to extensive profiling (assumed to be the “exact” metric). Of course, the advantage is that this estimation approach involves much less effort than profiling and is quite applicable in the embedded system domain where tasks are expected to perform a specific functionality and typically the designer has a good idea of where most of the execution takes place.

7.2 Our Approach: Bayesian Belief Networks

In order to identify the most frequently visited portions of the task’s FFG we use an approach similar to Markov chains but based on Bayesian Belief Networks using the MSBN inference engine from Microsoft Research [9]. The MSBN tool uses a version of the proposed Bayes Net Interchange Format for representing belief networks. In order to compute the probabilities we represented the state transition relation consisting of current state, next state, and conditionals as shown in the screenshot of Figure 6. We initially assign equal probabilities to all the reachable states and then iterate the probability computation until a fixpoint is reached.

Table 1. Frequency of Execution Distribution for Uniform Conditionals

<table>
<thead>
<tr>
<th>State</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>S6</td>
<td>0.046</td>
</tr>
<tr>
<td>S7</td>
<td>0.024</td>
</tr>
</tbody>
</table>

8. Synthesis

In order to perform synthesis, the FFG is mapped into the Software Hardware Intermediate FormaT (SHIFT) representation of the POLIS co-design toolset. SHIFT is a representation format for describing a network of EFSM’s. It is a hierarchical netlist consisting of:

- Co-design Finite State Machines (CFSM’s): finite state machines with reactive behavior
- Functions: state-less arithmetic, Boolean, or user-defined operations.

A CFSM execution consists of four phases:
1. Idle awaiting trigger inputs
2. Sample inputs when invoked
3. Compute chain of operations
4. Emit outputs, return to Idle mode

A CFSM in SHIFT is therefore composed of input, output, state or feedback signals with initial values, as well as a transition relation (TREL) that describes the reactive behavior. Functions are used in the TREL to assign computation results to valued outputs. A function can be thought of as a combinational circuit in hardware or a function (with no side effects) in software.

We therefore decompose the CLIF representation of each task into a single reactive control part, and a set of data path functions consistent with the current default SHIFT macro-architecture. We then use the POLIS engine to build the CDFG for hardware and software co-synthesis [13].

The complete optimization and synthesis flow with RCM is shown in Figure 7.

9. Results

We collected results for the 68hc11e9 target architecture and the ARM920T (from the ARMulator) using the macro-modeling estimation method of [11].
The results collected for the reactive Knoop example are shown in Table 2. To be fair to the cost guided approach we report the results for the worst-case i.e. the longest computation path, since we cannot conceivably get adequate profile coverage to the degree done in the static frequency of execution analysis. It can be seen that the number of nodes in the CDFG for synthesis increases because of the addition of the registers. It should be noted that code size decreases as well because of the redundancy removal. So the operation motion benefit is reflected in both code size and runtime.

<table>
<thead>
<tr>
<th>Method</th>
<th>CDFG (nodes)</th>
<th>cc11 (bytes)</th>
<th>cc11 (cycles)</th>
<th>armcc (bytes)</th>
<th>armcc (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/out RCM</td>
<td>132</td>
<td>994</td>
<td>454</td>
<td>1328</td>
<td>1897</td>
</tr>
<tr>
<td>w/ RCM</td>
<td>133</td>
<td>964</td>
<td>440</td>
<td>1244</td>
<td>1799</td>
</tr>
</tbody>
</table>

% improved - 3.0 % 3.1 % 6.3 % 5.2 %

Table 2. Results of RCM for Reactive Knoop’s Example

It can be seen from the table that the benefit of RCM is more apparent in the register rich ARM9 (with THUMB extension) architecture.

10. Conclusions and Future Work

We presented a novel approach for task response time optimization that borrows the concept of code hoisting from the software and high-level synthesis domains and applies it to embedded systems. We showed that a simple “indirect” operation motion technique specialized to the embedded system domain and guided by user input, dubbed Relaxed Code Motion (RCM), can be used efficiently to optimize task runtime before the synthesis step. Investigation results on software synthesis are very encouraging. In the future, in addition to collecting synthesis results on an extensive set of real application benchmarks, we’d like to explore applying additional cost metrics to guide operation motion such as the “context-dependent” costs used in [3] so as to further improve our function/architecture optimization and co-design framework [12].

ACKNOWLEDGMENTS

Our thanks to Richard Newton, and Bob Brayton from UC Berkeley for their insightful comments on and direction of our work. Gratitude is also paid to the SRC for funding this research under the Graduate Fellowship Program.

REFERENCES


