Task Response Time Optimization Using Cost-Based Operation Motion

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

Abdallah Tabbara
EECS Department
University of California at Berkeley
Berkeley, CA 94720
+1-510-643-5187
atabbara@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 motion from the software domain. Relaxed Operation Motion (ROM) 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. We introduce here our algorithm, how it differs from other code motion approaches, and its application 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, e.g., cell phones, pagers, security devices. Increasingly synthesis is playing a crucial role in design methodologies for embedded systems that advocate abstraction, decomposition, and refinement [10]. 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 paper, 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 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 [11]. 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 [6].

In practice code movement to the earliest program points can create pressure on the target architecture resources e.g. because of register "spills". A more practical approach involves also performing *temporary lifetime minimization* as in the work on Knoop [7]. Knoop's approach is the best-in-class approach for code motion since it involves unidirectional analysis techniques 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 techniques. Hailperin in [4] extended Knoop's method to incorporate cost into the code motion process. However, the cost metric is based on individual operations (i.e. `*, +, ...`) and does not account for the frequency of execution of program portions. The goal of the latter is to place the instructions in (safe) positions in the program where the context possibly permits simplification of the particular operation through, for example, constant folding, or operation strength reduction.

Castelluccia et. al. [3] used runtime cost to optimize protocols but their techniques were based mainly on node re-ordering at the CDFG for synthesis level.

3. Our Contribution and Overview

As in [3], 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 behavioral description of the task. Our operation 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 operation motion being part of a *general optimization flow and framework*; in particular we take advantage of two analysis and information gathering steps within the flow: reachable variable definitions, and reached uses of variables. So, the framework's complexity of *$O(n^2)$* is what dominates the overall complexity. Of course, for this increase in complexity we can get much better optimization results than [7] since operation 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 [11]. Our approach, dubbed *Relaxed Operation Motion (ROM)* is also specialized to the embedded system domain. In this domain, we are constrained by *I/O schedule preservation*, but 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 software synthesis engines therein for output generation.

4. Intermediate Design Representation

We briefly describe our intermediate design representation that permits us to perform constrained task optimization at a high level of abstraction. We use an implementation-independent state-scheduled task representation referred to as *Attributed Function Flow Graph (AFFG)* equivalent to the initial EFSM representation. The AFFG is a *refinement* of the *Function Flow Graph* (FFG) we have introduced in [11]. The FFG is quite similar to the classical Control Flow Graph from the software domain with the distinction of being concerned with *safe* function optimizations that preserve the I/O semantics of the task. The AFFG incorporates more information into the representation by adding *attributes* to the FFG nodes, and associated operations [10]. These attributes can either be solicited from the user or obtained by inference or profiling. An example of the former is the state schedule obtained from the "front-end" (e.g. Esterel [2]) and used to qualify the AFFG nodes. An example of the latter is the cost of operations like addition and multiplication, and the visit probability of AFFG node collections. The visit probability attribute is the "cost" that guides the ROM algorithm, as we will see shortly.

4.1 Attributed Function Flow Graph (AFFG)

AFFG is the task representation used for guided design analysis, optimization, and trade-off between the *function* the task describes, and the *architecture* that implements this function. Each EFSM state is captured by a collection of nodes while edges in the graph 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: An Attributed Function Flow Graph (AFFG) 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*x*V*, where (x,y) is an edge from *x* to *y* such that $x \in Pred(y)$, the set of predecessor nodes of *y*.

iii. $N_0 \in V$ is the start node (header that leads to the node(s) corresponding to the EFSM initial state(s)).

iv. Operations are *associated* with each node *N*.

v. *Attributes* are allied with nodes and operations. We assume that at least a state attribute is obtained from the font-end.

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. We assume w.l.o.g. that the model of computation dictates that input and output semantics must be preserved at the *state boundary*.

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

b) Raise the abstraction level, and allow function and architecture optimization and trade-off to be reflected in both hardware and software synthesis.

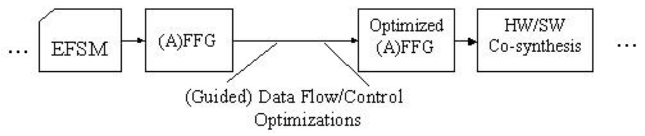


Figure 1. Data Flow and Control Optimization Flow

In order to implement (guided) task optimizations we have developed an *optimizer* that examines the (A)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 AFFG structure is shown in Figure 2 in *Tree* form for a simple example. Nodes denoted with an F are AFFG nodes that are labeled with the state assignment shown with an S. A *DAG* form is also available where AFFG nodes (comprising a DAG) are "shared" within and between states, but we will limit our presentation in this paper to the former. The optimization itself (shown in Figure 1) is broken into two phases (we neglect the *micro*-architecture optimization here):

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

b) *Macro*-Architecture Dependent phase: The AFFG is considered where the state schedule is taken into account, and operations within states are optimized followed by an allocation of registers and computations step.

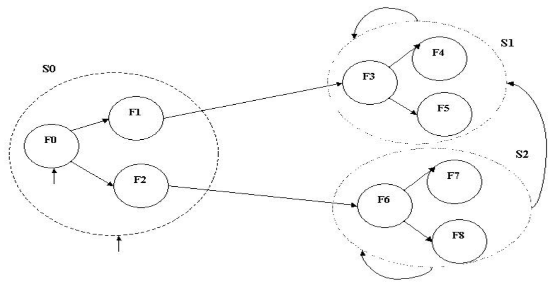


Figure 2. EFSM in AFFG Tree Form: A Simple Example

5. Illustrative Example

In order to illustrate our ROM 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 `S10` back to `S1` so that the system is running continuously. Variables `a`, `b`, `c` are declared internal, and initialized in `S1` 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 focus our discussion on nodes `S8` and `S9` since they are *costly*, as we will see in Section 7.2, and try to relocate the aforementioned addition operation to other less expensive nodes.

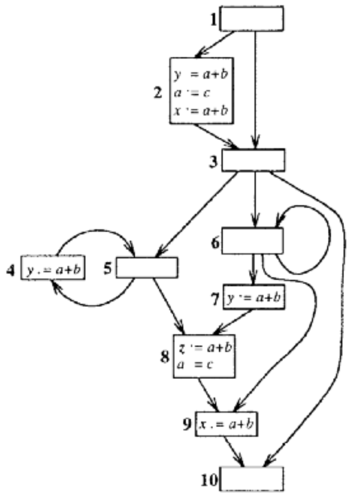


Figure 3. Illustrative Example (from [7])

6. (Cost-guided) Relaxed Operation Motion

Our operation motion approach consists of 4 steps performed *in sequence*:

1) **Data Flow and Control Optimization**: is an order of steps that optimize the (A)FFG representation as stated in Section 4.2.

2) **Reverse Sweep**: is the optimization step that we are mainly addressing in this paper where code is relocated from one or more AFFG 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 all or selected AFFG nodes based on cost.

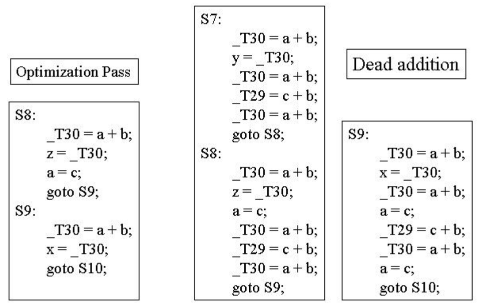
   b) *Normalization and Available Operation Elimination*: This optimization step effectively replaces the operation motion candidates from the targeted AFFG nodes to other less costly nodes as a result of step (a).

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

3) **Forward Sweep**: 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*. This step is optional.

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

Our approach comes "naturally" in an optimization framework [1], which allows us the use of relatively simple techniques to accomplish our goal. The results after steps (1) and (2a) are shown in Figure 4. Figure 5 shows the result after (2b), and (4), step (3) is not applied here. Note that in the final result the redundant computations in `S8` and `S9` are indeed relocated up to `S1` (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 ROM. The improvement comes about because ROM is part of a *comprehensive* optimization framework (i.e. flow has normalization, copy propagation and dead elimination to name a few useful steps, see [11]).

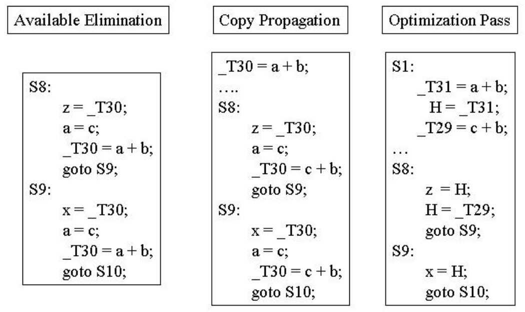

Optimization Pass
Dead addition

```
S8:
        _T30 = a + b;
        z = _T30;
        a = c;
        goto S9;
S9:
        _T30 = a + b;
        x = _T30;
        goto S10;
```

```
S7:
        _T30 = a + b;
        y = _T30;
        _T30 = a + b;
        _T29 = c + b;
        _T30 = a + b;
        goto S8;
S8:
        _T30 = a + b;
        z = _T30;
        a = c;
        _T30 = a + b;
        _T29 = c + b;
        _T30 = a + b;
        goto S9;
```

```
S9:
        _T30 = a + b;
        x = _T30;
        _T30 = a + b;
        a = c;
        _T29 = c + b;
        _T30 = a + b;
        a = c;
        goto S10;
```

Figure 4. Result After Dead Addition


Available Elimination
Copy Propagation
Optimization Pass

```
S8:
        z = _T30;
        a = c;
        _T30 = a + b;
        goto S9;
S9:
        x = _T30;
        a = c;
        _T30 = a + b;
        goto S10;
```

```
_T30 = a + b;
….
S8:
        z = _T30;
        a = c;
        _T30 = c + b;
        goto S9;
S9:
        x = _T30;
        a = c;
        _T30 = c + b;
        goto S10;
```

```
S1:
        _T31 = a + b;
         H = _T31;
        _T29 = c + b;
…
S8:
        z  = H;
        H = _T29;
        goto S9;
S9:
        x = H;
        goto S10;
```

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 [3]. It can be shown that the number of times each basic block (and, by analogy, each AFFG node) is executed, can be calculated by solving a system of *n* linear equations, where *n* is the number of basic blocks ([1]), assuming the probabilities of control passing from one block to the next is given [12]. 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 take place.

A Markov chain can be used to model and then compute *statically* the probabilistic control flow execution as described in [12] where it is also shown that this method is quite close to extensive profiling (assumed to be the "exact" metric). Of course, this estimation approach has the advantage of requiring much less effort than profiling, which has to be *exhaustive*. Hence, the method is quite applicable in the embedded system domain where tasks are expected to perform a specific functionality and the designer has typically a good idea of where most of the execution occurs.

7.2 Our Approach: Bayesian Belief Networks

In order to identify the most frequently visited portions of the task's AFFG, we apply an approach similar to Markov chains but based on Bayesian Belief Networks using the MSBN inference engine from Microsoft Research [8]. The MSBN tool uses a version of the proposed Bayes Net Interchange Format for representing belief networks. In order to compute the probabilities, we represent 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 fix-point is reached.

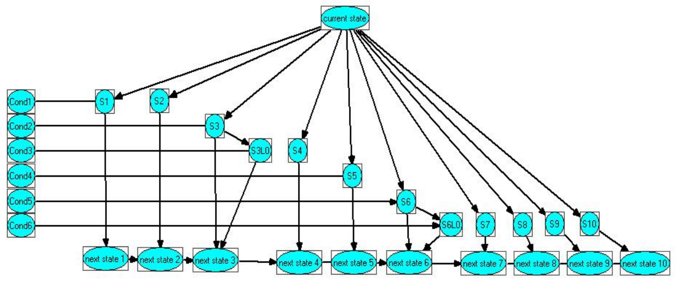


Figure 6. Belief Network for Knoop's Example *(Courtesy Microsoft Research)*

The frequency of execution results for the reactive version of Knoop's example (i.e. loop from `S10` back to `S1` added) with uniform probability of the conditionals (i.e. *P(True) = P(False) = 0.5*) is shown in Table 1 including costs of S8, S9, (operation motion candidates) and S7 (target of operation motion).

State	Probability
S1	0.15
S3	0.15
S5	0.15
S10	0.15
S9	**0.1**
S8	**0.09**
S2	0.07
S4	0.07
S6	0.046
S7	**0.024**

Table 1. Frequency of Execution Distribution for Uniform Conditionals

8. Synthesis

In order to perform synthesis, the AFFG is *mapped* into the Software Hardware Intermediate FormaT (SHIFT) representation of the POLIS co-design tool-set. SHIFT is a representation format for describing a network of EFSMs. It is a hierarchical netlist [2] of:

- Co-design Finite State Machines (CFSMs): 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 AFFG 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 engines to build the synthesis CDFG and perform hardware and software co-synthesis [11].

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

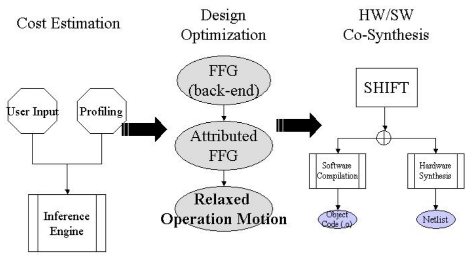


Figure 7. Optimization with ROM and Synthesis Flow

9. Results

We report here some results for yet another cost metric that can be used in guided ROM: the task *worst-case execution time (WCET)*. The WCET corresponds to the longest computation path if there is no task pre-emption [2]. This measure is very useful in schedule validation to check if the system meets its timing constraints, and also in processor resource utilization analysis, and schedule optimization. We use ROM to optimize the WCET path, which belongs to targeted state S2, and the next to worst path, which belongs to targeted state S8. We collected results for the *68hc11* and the *ARM920T* (from the ARMulator) using the *macro-modeling* estimation method of [2]. The results collected for the reactive Knoop example are shown in Table 2.

It can be seen from Table 2 that the number of nodes in the CDFG for synthesis increases because of register addition. ROM indeed improves the WCET; code size decreases as well because of the redundancy removal. Thus, the operation motion benefit is reflected in *both* code size and runtime.

Method	*CDFG (nodes)*	68hc11 (bytes)	68hc11 (cycles)	ARM9 (bytes)	ARM9 (cycles)
w/out ROM	*125*	934	454	1332	377
w/ ROM	*126*	904	432	1308	353
% Improved	-	**3.2 %**	**4.8 %**	**1.8 %**	**6.3 %**

Table 2. Worst-Case Response Time Results of Reactive Knoop's Example

The table shows that the benefit of ROM 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 motion 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 or profiling knowledge, dubbed Relaxed Operation motion (ROM), can be used efficiently to optimize task runtime before the synthesis step. Experimental results on software synthesis are very encouraging. Future work will be in the area of more extensive experimentation with our technique on real embedded applications. Furthermore, we would like to explore applying additional cost metrics to guide operation motion such as the "context-dependent" costs used in [3] in order to further improve our function/architecture optimization and co-design framework [10].

ACKNOWLEDGMENTS

Our thanks to Luciano Lavagno, and Felice Balarin from Cadence Berkeley Labs, 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

[1] Aho, A. V.; Sethi, R.; Ullman, J.D., "Compilers: Principles, Techniques, and Tools", *Addison-Wesley*, 1988.

[2] Balarin F.; Chiodo M.; Giusto P.; Hsieh H.; Jurecska A.; Lavagno L.; Passerone C.; Sangiovanni-Vincentelli A. L.; Sentovich E.; Suzuki K.; and Tabbara B., "Hardware-Software Co-Design of Embedded Systems: The POLIS Approach", *Kluwer Academic Publishers*, May 1997.

[3] Castelluccia, C.; Dabbous, W., "Generating Efficient Protocol Code from an Abstract Specification", *ACM/SIGCOMM*, 1996.

[4] Hailperin, M., "Cost-Optimal Code Motion", ACM Transactions on Programming Languages and Systems, Vol. 20, No. 6, pp. 1297-1322, November 1998.

[5] Kam, J.B.; Ullman, J.D., "Monotone Data Flow Analysis Frameworks", *Acta Informatica*, 1977, pp. 305-307.

[6] Knoop, J.; Rüthing, O.; Steffen, B., "Lazy Code Motion", *ACM SIGPLAN*, Vo. 27, No. 7, pp. 224-234, 1992.

[7] Knoop, J.; Rüthing, O., "Optimal Code Motion: Theory, and Practice", ACM Transactions on Programming Languages and Systems, Vol. 16, No. 4, pp. 1117-1155, July 1994.

[8] Microsoft Research: Decision Theory & Adaptive Systems Group at: http://research.microsoft.com/msbn/default.htm

[9] Morel, E.; Renvoise C., "Global Optimization by Suppression of Partial Redundancies", *Commun. ACM*, Vol. 22, No. 2, pp. 96-103, 1979.

[10] Tabbara, B., "Function Architecture Optimization and Co-design of Embedded Systems", Ph.D. Dissertation, *University of California at Berkeley*, in progress, May 2000.

[11] Tabbara, B.; Tabbara, A.; Sangiovanni-Vincentelli, A., "Hardware and Software Representation, Optimization, and Co-synthesis for Embedded Systems", Technical Report *UCB/ERL M00/7*, January 2000.

[12] Trivedi, K., "Probability and Statistics with Reliability, Queuing and Computer Science Applications", Englewood Cliffs; *Prentice Hall*, 1982.

[13] Wagner, T.; Maverick, V.; Graham, S.; Harrison, M., "Accurate Static Estimators for Program Optimization", ACM SIGPLAN, 1994.