Generation of Minimal Size Code for Schedule Graphs

Claudio Passerone
Politecnico di Torino, Italy

Yosinori Watanabe
Cadence Design Systems, USA

Luciano Lavagno
Università di Udine, Italy

Abstract

This paper proposes a procedure for minimizing the code size of sequential programs for reactive systems. It identifies repeated code segments (a generalization of basic blocks to directed rooted trees) and finds a minimal covering of the input control flow graphs with code segments. The segments are disjunct, i.e. no two segments have the same code in common. The program is minimal in the sense that the number of code segments is minimum under the property of disjunction for the given control flow specification.

The procedure makes no assumption on the target processor architecture, and is meant to be used between task synthesis algorithms from a concurrent specification and a standard compiler for the target architecture. It is aimed at optimizing the size of very large, automatically generated flat code, and extends dramatically the scope of classical common subexpression identification techniques.

The potential effectiveness of the proposed approach is demonstrated through preliminary experiments.

1 Introduction

Recent technology advances push the performance of current electronic systems to new heights. However, the design practice does not allow an increase of the productivity of designers to take full advantage of the available performance; this is particularly true in the embedded system world, where products must be designed in a very short time and specification changes very often. One way to solve this problem is to make extensive use of programmable devices, which are more flexible than hardware implementations and provide a faster design cycle. However, to get enough performance, the software written for the application should be of good quality.

Current design methodologies in software development try to increase the productivity by encouraging extensive use of IP components. Even though such a methodology is not fully adopted in design practice, programmers often copy (possibly large) pieces of code from previous projects or libraries. This practice leads to a fairly high degree of redundancy in the final code, and hence an increase in the total cost due to memory requirements. This redundancy is further increased if the software is not of very high quality, as it often contains duplicated code, especially in ill-structured loops, which are difficult to optimize by hand.

Code synthesis techniques from concurrent specifications based on "code stitching", such as [5, 8, 3], may also produce code with very high redundancy.

In this paper we propose a technique to address these problems. We start from a high level abstract view of the program, namely a graph which represents the flow of control and data, annotated with information on the flow of the program and actions to be performed; we then generate a new program with the same behavior of the initial one, modified to reduce the memory required to store the code. This is achieved by partitioning the input representation of the program into a set of segments in such a way that the code generated from those segment is maximally shared. Our procedure is intended as a technology independent step, and standard compilation techniques can be used afterwards to apply architecture dependent optimizations.

We mostly target reactive real-time embedded systems: software in this domain is often partitioned into a set of concurrent modules which communicate through primary inputs and primary outputs; a common characteristic is that modules wait for a set of external events, perform some computation producing some outputs, and then go to sleep waiting for a new event to come. Usually they monitor several different inputs, and react to their occurrence according to the application goals. For each primary input, our technique generate a task, which is invoked whenever an event occurs at that input. To do this, we assume that a special annotation is available in the initial graph representation of the program, that clearly identifies where that input is waited for and read.

The paper is organized as follows: Section 2 formally describes our starting point and the problem that we want to solve. Section 3 describes in details the algorithm that we have implemented. Section 4 shows some preliminary example to show the effectiveness of our technique. Section 5 finally concludes the paper.

2 Problem definition and previous work

The input instance of our procedure is a *Schedule Graph*. It is a finite directed graph, where each node has a unique *name* and each arc corresponds to an *action*. In the following we may refer to a Schedule Graph also by calling it a schedule. A Schedule Graph has a distinguished node, called the *initial node*. Some of the nodes may be tagged as *await nodes*, which are nodes where the system stops, waiting for a primary input event; there should be at least one await node in the graph. Await nodes are used to model reactive behavior of a system.

Each outgoing arc of an await node corresponds to a primary input. Each action has a name and an associated code; there may exist multiple arcs with the same action. Without loss of generality, actions with the same associated code should always have the same name. Each node also has an associated label, called *Enabled Actions* (EA), which lists the names of all actions associated with its outgoing arcs. Two nodes are said to be equivalent if they have the same Enabled Actions. A node having more than one outgoing arc is a choice; at this time, only data dependent choices are supported, where values of the data available at the run-time will uniquely determine an arc to be taken for the succeeding action[1]. A choice node has an associated condition to be evaluated to resolve the choice.

A Schedule Graph models the behavior of a sequential program where the execution starts with the initial node and traverses the graph to execute actions associated with visited arcs. The execution proceeds to one direction at a choice, based on values of data, and stops at an await node until an event is provided at a primary input. The execution then resumes by taking the action of the outgoing arc of the await node corresponding to the input.

An example of a Schedule Graph is a Control Flow Graph (CFG) of a program. Branches such as an `if-then-else` construct constitute a choice (node with more than one outgoing arcs), and loops correspond to a cycle in the graph representing the schedule. The entry point of the program is the initial node, which is also the only await node in the graph, i.e. the arrival of an input event causes the execution of the program. More complicated examples arise from a specification using multiple modules, whose concurrency is resolved by means of scheduling techniques. E.g., a Schedule Graph can be constructed as a path in the Reachability Tree of a Petri Net, or from a Finite State Machine, where states have no associated actions [8, 3].

An example of a Schedule Graph is shown in Figure 1. A node has a name, followed in parenthesis by the Enabled Actions. Arcs are labeled with the names of the actions. The initial node is r, which is also tagged as an await node. This means that the computation begins when an event is received at the input. Node v_3 is also an await node: if computation ever reaches this node, then the execution is suspended until another event is received at the input. Nodes v_1 and v_5 are choices: at run-time, a condition specified for these nodes is evaluated to determine which branch to take. Actions should have an associated code, which is executed whenever a transition from one node to the next occurs.

Our goal is to process a Schedule Graph, generating code for each primary input while minimizing its size. The code is called a task for the input. This is achieved by sharing code for the same action occurring in different arcs of the graph. This technique is technology independent, as it does not assume any particular underline architecture; a compiler shall be used to perform technology dependent optimizations after applying our procedure.

Our problem is a generalized version of a well-known step of compiler optimization: common subexpression extraction [1], generalized in order to expand the class of code sequences that can be handled (arbitrary sub-trees of a control flow graph), and simplified in that only maximal common sub-graphs are considered, rather than those leading to maximum sharing.

[1] As opposed to choice depending on the availability of inputs to multiply enabled *await*s.

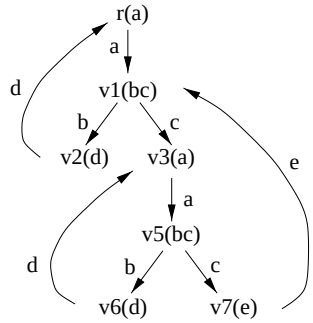

Figure 1. Example of a Schedule Graph

Code minimization has been addressed a number of times in embedded software compilation. Bhattacharyya *et al.* in [2] give two heuristic algorithms that are specialized for Schedule Graphs composed of multiple nested loops, and that also guarantee non-duplication of code segments (loop bodies). The authors of [6, 7], on the other hand, work at the binary code level, and look for common code sequences to be inserted in a "dictionary" of lightweight subroutines to be called.

Our approach deals with more general Schedule Graphs, including data-dependent branching, and works at the source level on non-linear tree-structured fragments. It is a simplified version of general graph matching, limited to trees to keep complexity low.

3 Code Generation

A very simple way of generating a program starting from a Schedule Graph is to traverse it, and stitch together code for each action, providing `if-then-else` constructs to reflect the flow of control when a run-time data dependent choice is encountered. This solution requires code for some actions to be duplicated, due to the serial-parallel form of structured code. In our approach, we try to preserve performance as much as possible, and we minimize the memory required for the program.

For sake of simplicity in the description of the algorithm, we assume that the module has only one primary input. This means that an *await node* corresponds to waiting for an event on that particular input. Since we want to generate one task for each input, this condition means that we will always generate one single task. The algorithm can be easily extended to handle multiple inputs, generating one task for each one of them.

3.1 Initialization: cutting loops

The code generation procedure that we have implemented assumes that the graph it operates on is acyclic. Since this is not in general the case for a given Schedule Graph, a first initialization step is needed to analyze the schedule and cut the

loops. This is used to provide a simple termination criterion for successive transformations. The analysis is performed using a depth first traversal of the schedule from the initial node r, and cutting loops when already visited nodes are encountered; a new leaf node is then created, with a different name but the same EA label of the destination node of the loop. The resulting graph is still a Schedule Graph, with the property of being acyclic.

Cutting of loops preserves the property that if a node v is reachable from the initial node in the Schedule Graph, then it is also reachable in the new acyclic graph. Since we add a node for each loop that is cut, different cuts may yield a graph of different size: while this does not change the results of our algorithm, it may slightly change its running time.

3.2 Algorithm for schedule traversal

The next step traverses the newly generated acyclic graph to extract sequences of actions that are candidates to be shared in the generated code. In particular, the graph is divided into a set of *code segments*:

Code Segment A *code segment* is a directed rooted tree that associates an action with each edge, and a state to each node. A code segment is a Schedule Graph, in which await nodes can only be at the root, or at the leaves. It is not necessary for a code segment to have an await node. During code generation, code segments isomorphic to subtrees of the schedule are created.

As we shall see later, code segments represent uninterruptable sequences of actions. Since await nodes require the execution to be suspended, they are forbidden within a code segment. The goal of code generation is to find the minimum set of disjunct code segments such that:

1. an action in the Schedule Graph belongs to one and only one code segment,
2. each code segment is isomorphic to a set of subtrees of the Schedule Graph, such that each arc of each subtree has the same action with that of the corresponding arc of the segment,
3. the set covers the entire Schedule Graph, i.e. each node of the Schedule Graph is in a subtree, for which an isomorphic code segment exists.

The *state* of a node of a code segment lists a set of pairs of the name and EA of a node in the Schedule Graph to which the node of the code segment is isomorphic to. This is needed because a single code segment may be used to execute different paths in the Schedule Graph, and the state is used to keep track of the flow of control. A piece of code is generated for each code segment following the algorithm outlined at the beginning of Section 3[2].

[2]Now applied to each segment only, rather than to the whole Schedule Graph, that would be very inefficient.

```
function traverse(s)
    nodelist = Create(); codelist = Create();
    appendElem(nodelist, root(s));
    while (!empty(nodelist))
        node_sch = getRemoveFirstElem(nodelist);
        newcodeseg = FALSE;
        if (!(node_code = findCodeSeg(node_sch, codelist)))
            newcodeseg = TRUE;
            node_code = initCodeSeg(node_sch, codelist);
        compare(node_sch, node_code, newcodeseg);
    return;
```

Figure 2. Pseudo-code for function `traverse`

The first property above guarantees that we minimize the memory requirements, since code is maximally shared. The second property tells us that within a code segment there can only be local jumps (such as `if-then-else`), while global jumps from one code segment to another occur only at the leaves. This means that once the execution of a segment starts, it continues until a leaf is reached. Moreover, looking at the minimum set of code segments means that they are maximal, since if a segment is not maximal, then it can be made isomorphic to a larger subtree of the Schedule Graph, by merging it with another code segment that corresponds to the newly covered subgraph, without violating the above properties. Therefore, we minimize the performance loss due to jumping from one code segment to another. The third property guarantees that the entire behavior can be represented in terms of code segments.

The traversal uses two main functions, `traverse` and `compare`. The first one prepares the data structure and starts the traversal, by calling `compare`, from the root node of the schedule. The second one recursively compares the schedule to existing code segments, or creates a new code segment if needed.

The pseudo-code for function `traverse` is shown in Figure 2. The argument s to the function is the acyclic schedule to be processed. Two lists are used in this function:

`nodelist` : a set of nodes of the schedule to be used as the starting point of a recursive comparison.

`codelist` : a list of the already generated code segments.

At the beginning the `codelist` is obviously empty and it should be filled by the traversal. The `nodelist` initially contains only the initial node of the schedule, which will be used as the first starting point; the function `compare` will then add new nodes to `nodelist` during its operation, as explained later. `traverse` gets and removes the first element of `nodelist` and checks if it is equivalent to the root of any already existing code segment: if it is not, a new entry in `codelist` is created with a root node equivalent to the node from the schedule, and a flag (`newcodeseg`) is set. The function `compare` is then called, either to fill the newly created code segment, or to compare an existing one to a subtree of the schedule rooted at the considered node.

```
function compare(node_sch, node_code, newcodeseg)
  if (newcodeseg)
    createKids(node_code, Kids(node_sch));
    ForeachKid(node_sch, &kid)
      if (nKids(kid) != 0)
        if (isAwait(kid) || findCodeSeg(kid, codelist))
          appendElem(nodelist, kid);
        else if (oldnode_code = isInCodeSeg(codelist))
          Detach(Kids(oldnode_code));
          appendElem(nodelist, kid);
        else compare(kid, Kid(node_code), newcodeseg);
      else   /* (nKids(kid) == 0) */
        if (oldnode_code = isInCodeSeg(codelist))
          Detach(Kids(oldnode_code));
  else   /* NOT newcodeseg */
    associate(M(node_sch), ECS(node_sch)) with node_code;
    if (all the associated ECS's coincide for node_code)
      if (nKids(node_code) == 0 && nKids(node_sch) != 0)
        appendElem(nodelist, node_sch);
      else
        ForeachKid(node_sch, &kid)
          compare(kid, Kid(node_code), newcodeseg);
    else   /* NOT ECSequal */
      if (nKids(node_code) != 0) Detach(Kids(node_code));
      appendElem(nodelist, node_sch);
  return;
```

Figure 3. Pseudo-code for function `compare`

Figure 3 is the pseudo-code for the function `compare`. Its goal is to compare a subtree of the schedule to existing code segments and create new segments if needed. The comparison ends when await nodes are encountered, and the await nodes are put in `nodelist` so that new invocations of the function can proceed with the traversal of the entire schedule. In some cases, however, the function returns before reaching an await node, if certain conditions occur, as detailed next.

This function works by calling itself recursively, and implements a depth first search of the subtree in the schedule graph. The argument `node_sch` is a node in the schedule, and `node_code` is the node in a code segment to be compared. Note that even if a code segment is new, at least one node always exists because it is created by the `traverse` function before calling `compare`.

If the code segment that has been identified is new (the flag `newcodeseg` is set), then new children are created in the code segment by copying the children from the node of the schedule and properly setting their state. This can be done because it is guaranteed that the two parent nodes in the code segment and in the schedule are equivalent and therefore have the same EA. Then, for each child in the schedule, termination conditions for the traversal are checked: it will stop if the child itself has no children (it means we have reached a loop in the schedule, that was cut in the initialization step), or if it has children but it is either an await node or a code segment already exists whose root has the same EA. In the last two cases, the child is also added to `nodelist` for further traversal. Sometimes, a node with an EA which is already associated with a node of a code segment is found, while the node in the code segment is not at the root. Then a new code segment is created, by cutting the existing one at that node, and making it the new root (function `Detach`); moreover, if the considered node had children, it is also added to `nodelist`. If none of the previous conditions occur, the traversal will continue recursively.

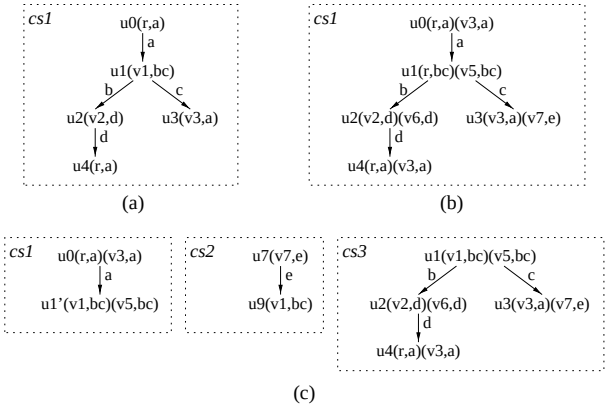

Figure 4. Example of schedule traversal

If the code segment is not new, i.e. `newcodeseg` is False, then the state and the EA of its node are updated by adding those of the node in the schedule to the state set. If the EA is identical to all the previous ones, it means that the node in the subtree and that in the code segment are equivalent, and therefore the comparison can continue. However, it should be stopped if the node in the code segment does not have children and the one in the schedule does, as we should select another code segment for comparison. On the contrary, if the EA is different from the previous ones stored in the node in the code segment, a leaf has been identified; if there are children in the code segment, then they should be detached to create a new code segment, and the node in the schedule is added to `nodelist`.

An example of traversal of the Schedule Graph presented in Figure 1 is presented in Figure 4 (the acyclic schedule obtained after cutting the loops is not shown). Three consecutive calls of the function `compare` are necessary to entirely traverse the graph, and the intermediate results that they produce are shown in (a), (b) and (c) respectively. The first call of function `compare` will generate code segment cs_1; the second call will compare a subtree starting at node v_3 with the already generated code segment, adding state and EA information. The third and last step creates a new code segment cs_2, and splits the first one into two different code segments, due to the back edge to node v_1.

3.3 Code synthesis

The goal of synthesis is to take the output of the traversal and generate code in a target language from it. Since the association between actions and code is already known, a structure should be added to reflect the schedule, and variables should be used to keep track of the state. The syntax of both the structure and the variables is language dependent, and we have chosen the C language to implement them (other languages can be added with little effort).

The generated code is divided into three main parts: decla-

rations, initialization and run.

Declarations This part includes several declarations needed in the C language, such as new data types, prototypes and global variables. These include declarations from the original specification, as well as declarations needed for the schedule structure, such as state variables.

Initializations State variables need to be properly initialized so that the first reaction is correct. The value is derived from the schedule. Being global variables, the value will not be lost between two successive executions, so it just need to be updated correctly by the task.

Run This part generates the code to be used to implement a task. It does so by generating code for each code segment, adding a structure to jump from one code segment to another to reflect the original schedule, based on the state information.

The structure of a code segment is always the same and can be separated into three sections: *execution*, *update* and *jump*. They will be described in details in the following:

execution It contains the real code for actions and data dependent choices, taken from the original specification. It always starts with a label, which is the concatenation of the name of the actions that form the EA of the first node of the code segment. The label is used to jump to the code segment.

Then the graph for the code segment is traversed in a depth-first search manner. For each node whose EA is a single action, the code for that action is copied into the output file. If the node is a choice, then an `if-then-else` construct is generated using the condition specified in the node.

When a leaf is reached, the *update* and *jump* sections are generated before going back in the traversal, so that if there are choices in the code segment, more than one `goto` appears in the implementation.

update At each leaf of a code segment the state must be properly updated so that:

1. the next code segment to call to complete the computation can be correctly selected,
2. the state at the end of a sequence of code segments corresponds to the node in the schedule reached by the execution.

If multiple code segments are traversed during a single reaction, each one of them is responsible to update the state variable to reflect the change between the root node of the code segment, and the leaf that is reached. The sequence of these updates constitutes the global state change for that particular reaction.

jump This section must find which code segment to call next, or should return if the reaction is finished. With the exception of leaves, for all the nodes in a code segment the EA associated with a set of states is always the same. For a leaf, this property is not true, and the EA represents what to do next. Therefore, a switch construct on the state is used to select a `goto` statement, which will cause the execution to jump to the label named after the destination EA. If the destination is an await node, then a `return` is generated instead of a `goto`.

```
1   static int sv;
2   sv = 0;
3   void ISR_a(void) {
4   a:  a(); goto bc;
5   e:  e(); sv = sv - 2; goto bc;
6   bc: if (condition(v1) == TRUE) {
7           b(); d(); return;
8       } else if (condition(v1) == FALSE) {
9           c(); sv = sv + 1;
10          if (sv == 1) return;
11          else if (sv == 2) goto e;
12      }
13  }
```

Figure 5. The final code generated from the code segments shown in Figure 4-(c).

Synthesis will therefore generate a function which has no local variables and starts with the first code segment (whose root is always the initial node of the schedule, by construction), followed by all the others in the order in which they were found during traversal. When the last code segment is generated, the function is closed. This function has just one entry point, but may have several exit points corresponding to all the leaves that perform a `return`.

Figure 5 shows the generated code for the example described in Figure 4. The declaration and initialization parts correspond to lines 1–2; the run function, called `ISR_a()`, starts at line 3. A variable *sv* is used to keep track of the state, and $t()$ denotes the code associated with the action t. Code segments start with a label, equal to the EA of their root node.

3.4 Properties and Analysis

We show that our algorithm generates a set of code segments that satisfy the properties in Section 3.2. The first thing to notice is that code segments are indeed isomorphic to subtrees of the Schedule Graph, as they are obtained by a depth-first traversal implemented in the function `compare`; in fact, they are spanning trees of subgraphs of the Schedule Graph, rooted at the node from which function `compare` is initially called. Actions appear only once in the set of code segments, because when creating a new code segment we first check if there is a node equivalent to the node being processed in either existing code segments or in the one currently being created, and the creation stops whenever this is the case. Further, since we traverse the full Schedule Graph and create code segments as needed, we guarantee to cover it completely. The set is disjunct, because as soon as we reach a node which is equivalent to one already present in another code segment, we stop build-

ing the new one, cutting it at the last reached node, and start a comparison with the old one. Finally, the set is minimal because we build the largest possible segments, as we never stop adding new nodes unless they are already present.

The algorithm for the traversal is polynomial with respect to the size of the input acyclic graph, which in turn is linear in the size of the original Schedule Graph. The traversal itself, being depth-first, is linear, but to guarantee properties we need to search already created code segments at most once for each node of the graph. As the total size of the code segments is never greater than that of the initial graph, even a simple linear searching technique would make the overall algorithm quadratic in the size of the Schedule Graph. Using a little more sophisticated search, the algorithm can be made $O(n \log n)$, where n is the number of nodes of the graph.

Our procedure is similar to other algorithms that operate on graphs, namely partitioning and matching; however, it differs in some details that make the computation easier. Partitioning is the problem of dividing a graph into subgraphs of certain properties, where it is known to be NP-complete for many properties, such as isomorphism ([4]). Our problem also requires to divide the graph, but each of the resulting graphs (segments) may be isomorphic to several subgraphs of the original one; further, each code segment can be incrementally created starting from a root node until some condition is violated, which can be checked locally. Matching assumes that a set of graphs is given, and the problem is to find the best covering of another graph by the set; this is also NP-hard. The difference in our case is that the subgraphs are not given, but can be created in such a way that the covering is minimal with respect to our cost function. Moreover, we consider only trees as both subject and candidate graphs, in order to keep a polynomial complexity.

4 Example

We have implemented the proposed algorithm within a framework for the design of embedded systems, described in [3]. It uses Petri Nets as the underlying model of computation and outputs a Schedule Graph, which can then be used to generate code. The input specification is based on C, so also the output code uses the same language.

We applied the methodology to an example coming from a multi-media application, showing the effectiveness of our technique. The generated code is extremely small in size, compared to the one obtained by directly translating the Schedule Graph into code. Table 1 shows the number of lines of C code for both the unprocessed and processed program, and the number of bytes required to implement them with different kind of compiler optimizations. **Direct** shows the case of direct translation of the Schedule Graph, while **Processed** is the result obtained by applying the proposed procedure. We used a Sun Ultra-1 running Solaris 2.5.1 and gcc 2.7.2.2. While the number of lines is reduced by almost a factor of four, the actual improvement in the memory size is consistently better. When the highest optimizations available in gcc are used, the code size for the direct translated program increases a lot with respect to the other experiments; however, the impact on the processed program is not as strong, so we achieve a higher ratio. The reason for this behavior is that inlining is introduced at this optmization step in the compiler, but the number of function calls where this can occur is reduced in the processed program, since they are shared.

	Direct	**Processed**	**Ratio**
# lines	1237	322	3.8
gcc-O0	34624	4844	7.1
gcc-O1	22540	3252	6.9
gcc-O2	22524	3212	7.0
gcc-O3	52288	4032	12.9

Table 1. Experimental results

5 Conclusions

We proposed a technique to synthesize efficient code from a schedule to be run on a single processor. The technique produces a minimal number of large code segments, so that an optimizing compiler can perform a good job. We presented an application of the methodology on an example from a multimedia application.

References

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

[2] S. Bhattacharyya, J. Buck, S. Ha, and E. A. Lee. Generating compact code from dataflow specifications of multirate signal processing algorithms. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, March 1995.

[3] J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, and A. Sangiovanni-Vin centel li. Task Generation and Compile-Time Scheduling for Mixed Data-Control Embedded Software. In *37th ACM/IEEE Design Automation Conference*, June 2000.

[4] M. R. Garey, , and D. S. Johnson. *Computers and Intractability*. W.H. Freeman, 1979.

[5] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow graphs for digital signal processing. *IEEE Transactions on Computers*, January 1987.

[6] H. Lekatsas and W. Wolf. SAMC: a code compression algorithm for embedded processors. *IEEE Transactions on CAD*, December 1999.

[7] S. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded dsp processors using data compression techniques. *IEEE Transactions on CAD*, July 1998.

[8] B. Lin. Software synthesis of process-based concurrent programs. In *35th ACM/IEEE Design Automation Conference*, June 1998.