Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator

Silvina Hanono
Department of EECS, MIT
silvina@lcs.mit.edu

Srinivas Devadas
Department of EECS, MIT
devadas@mit.edu

Abstract—
The AVIV retargetable code generator produces optimized machine code for target processors with different instruction set architectures. AVIV optimizes for minimum code size.

Retargetable code generation requires the development of heuristic algorithms for instruction selection, resource allocation, and scheduling. AVIV addresses these code generation subproblems concurrently, whereas most current code generation systems address them sequentially. It accomplishes this by converting the input application to a graphical (Split-Node DAG) representation that specifies all possible ways of implementing the application on the target processor. The information embedded in this representation is then used to set up a heuristic branch-and-bound step that performs functional unit assignment, operation grouping, register bank allocation, and scheduling concurrently. While detailed register allocation is carried out as a second step, estimates of register requirements are generated during the first step to ensure high quality of the final assembly code.

We show that near-optimal code can be generated for basic blocks for different architectures within reasonable amounts of CPU time. Our framework thus allows us to accurately evaluate the performance of different architectures on application code.

I. INTRODUCTION

A. Hardware–Software Co-Design for Embedded Systems

For a variety of reasons, manufacturers profit from integrating an *entire system* on a single integrated circuit. However, designing an entire complex system as an Application Specific Integrated Circuit (ASIC) is neither economical nor practical. Furthermore, as time-to-market requirements place greater burdens on designers for fast design cycles, an increasing amount of system functionality is implemented in *software* relative to hardware.

In a *hardware–software co-design* [1] methodology, designers first partition the system functionality into hardware and software. A target processor, called an *embedded processor*, is then selected from existing processor designs, or an ASIP (Application Specific Instruction-Set Processor) is designed to execute the software. The hardware, software, and ASIP are designed and the resulting system is evaluated using a hardware–software co-simulator. The partitioning and processor design are repeated until an acceptable system is developed.

As the complexity of embedded systems grows, programming in assembly language and optimization by hand are no longer practical. Further, hand coding virtually eliminates the possibility of changing the processor architecture. We argue that in order to be able to explore the processor design space, an automatically retargetable compilation strategy is required.

B. Overview of the AVIV Retargetable Code Generator

We are developing a retargetable code generator, AVIV, whose inputs are the application program and a machine description of the target processor. The machine description, written in ISDL (Instruction Set Description Language) [2], includes an instruction set specification and some architectural information. AVIV produces code, optimized for minimal size, that can run on the target processor. Code size is our optimization cost function because we are focusing on embedded processors where the size of the on-chip ROM is a critical issue.

By varying the machine description and evaluating the resulting object code, the design space of both hardware and software components can be effectively explored. AVIV focuses on architectures exhibiting instruction-level parallelism (ILP), including Very Long Instruction Word (VLIW) architectures.

Retargetable code optimization requires the development of heuristic algorithms for instruction selection, resource allocation, and scheduling. AVIV addresses these code generation subproblems concurrently, whereas most current code generation systems address them sequentially. The main reason why current code generators address these problems sequentially is to simplify decision-making in code generation. However, decisions made in one phase have a profound effect on the other phases.

AVIV converts the application code to a graphical (Split-Node DAG) representation that specifies all possible ways of implementing the application's basic blocks on the target processor. The information embedded in this representation is then used to set up a heuristic branch-and-bound step that performs functional unit assignment, operation grouping, register bank allocation, and scheduling concurrently. While detailed register allocation is carried out as a second step, required loads and spills due to limits on available registers are generated and scheduled during the first step. This ensures that a valid detailed register allocation can always be found without undoing the chosen operation groupings or functional unit assignments.

C. Organization of this paper

Section II describes the compiler framework of the AVIV code generator. Section III presents the Split-Node DAG formulation for retargetable code generation. Section IV formulates the code generation problem as a covering problem on the Split-Node DAG. Section V presents previous work on retargetability and code generation, as well as comparisons to our own work. Results using our code generator and ongoing work on our compiler project are presented in Section VI.

II. COMPILER FRAMEWORK

Figure 1 illustrates the compiler framework used by the AVIV code generator. The compiler front end receives a source program written in C or C++. It performs machine independent optimizations and generates an intermediate format description in SUIF[1] [3]. The ISDL [2] machine description of the target processor is generated either by hand or by a high-level CAD tool. The compiler back end takes the SUIF code and the ISDL description as inputs and produces assembly code specific to, and optimized for, the target processor. The ISDL description is also used to create an assembler. The automatically generated assembler transforms the code produced by the compiler to a binary file that is used as input to an instruction-level simulator. This paper focuses on the back-end of the AVIV code generator.

AVIV uses the SUIF compiler [3] in conjunction with the SPAM[2] [4] compiler as its front-end. The front-end converts the source program to an intermediate form consisting of expression DAGs and control flow information. It also performs machine independent optimizations such as loop unrolling and other transformations that extract machine independent parallelism. Thus, the starting point of the AVIV compiler is a number of basic block DAGs connected through control flow information. A sample basic block DAG is shown in Figure 2. AVIV then focuses on extracting machine

[1] Stanford University Intermediate Format
[2] Synopsys, Princeton, Aachen, MIT

1-58113-049-x-98/0006/$3.50 DAC98 - 06/98 San Francisco, CA USA

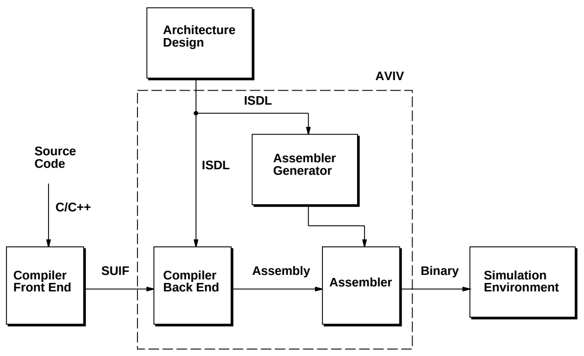

Fig. 1. **Retargetable Code Generation Framework using Aviv**

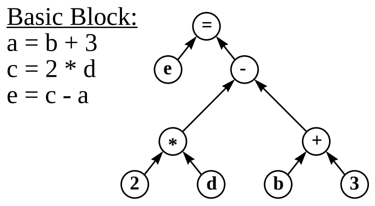

Fig. 2. **Basic Block DAG**

dependent parallelism using the ISDL description of the target processor. This is accomplished by converting the basic block DAGs, or expression trees, into Split-Node DAGs.

The instruction set information contained in the ISDL machine description is used to create several databases which are later used to create the Split-Node DAG, as described in Section III. One of these databases stores a correlation between the target processor operations and the SUIF basic operations such as ADD and SUB. This information comes from the RTL description of each instruction. A second database stores all possible data transfers explicitly stated in the target machine description, and is subsequently expanded to include multiple-step data transfers as well.

The machine description also describes constraints on the instructions that exist in the target processor. This information is used to remove the constrained operations from the databases ensuring that when creating the Split-Node DAG, it will only contain legal mappings from SUIF operations to target machine operations.

III. The Split-Node DAG

A. The Split-Node DAG Definition

The Split-Node DAG representation contains all the necessary information to generate code that will perform the operations of the original basic block DAG on the target processor. In addition, it allows for the exploration of parallelism, that is achievable on the target architecture, within the basic block. The Split-Node DAG provides information about the multiple ways that any operation can be performed on the target processor. For example, an ADD SUIF node will be split into several nodes representing the different functional units that can perform an ADD operation. The Split-Node DAG also includes data transfer information. It is important to include data transfer nodes because the cost of covering the Split-Node DAG should include the cost of transferring data between the units. Also, by including the transfer nodes, the transfers are automatically scheduled along with the operations.

Syntactically, a Split-Node DAG is similar to a DAG representing the operations performed in a block of code. A Split-Node DAG has two additional types of nodes: *split nodes* and *data transfer nodes*. A *split node* corresponds to an operation node in the original DAG. The immediate (non data transfer node) descendants of a split node correspond to all possible ways the operation may be performed on the target processor. The *data transfer nodes* are inserted on the path between a split node and its immediate operation descendants and correspond to data transfers required along that path.

The Split-Node DAG explicitly represents all possible *implementations* for a block of code on the target processor. Operation

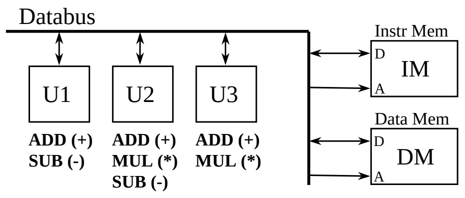

Fig. 3. **Example Target Architecture**

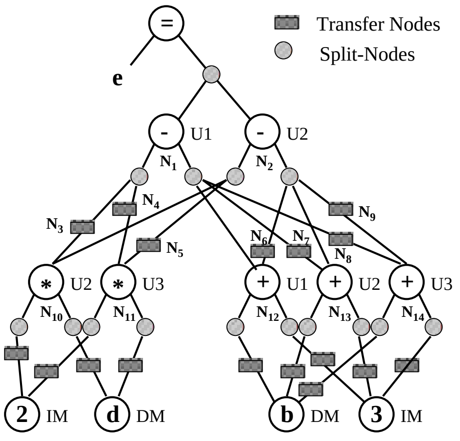

Fig. 4. **Split-Node DAG**

grouping, functional unit assignment[3], register bank allocation, and scheduling are performed simultaneously from the Split-Node DAG. An *implementation* of a block of code corresponds to a set of instructions in which all operations have been assigned an execution unit, and all data transfers[4] between these units are accounted for. At this stage, detailed register allocation has not been performed, however, we estimate the number of registers required from each register bank, and insert loads and spills if the available resources are exceeded. This guarantees that we do not have to modify the instruction selection during detailed register allocation.

B. Creating the Split-Node DAG

To clarify the structure of the Split-Node DAG, let us create one for the basic block DAG shown in Figure 2. We will use the example VLIW processor presented in Figure 3 as the target processor. In this processor, Unit U1 can perform addition (ADD), and subtraction (SUB). Unit U2 can perform addition (ADD), subtraction (SUB), and multiplication (MUL). Unit U3 can perform addition (ADD), and multiplication (MUL). Each unit contains its own register file and can perform only one operation at a time. The architecture also includes an instruction memory (IM), a data memory (DM), and a databus that connects all units and memories.

All of the information required to generate the Split-Node DAG is extracted from the ISDL target machine description, as described in Section II. Figure 4 illustrates the result of converting the basic block DAG into a Split-Node DAG. Each Split-Node has children corresponding to the multiple units that can perform the ADD, MUL, or SUB operations on the target architecture. Note that paths from several split nodes can reconverge to one set of operation nodes. Any edge connecting two operations, A and B, in the original basic block DAG, is now split into multiple edges representing all possible paths from operation A to operation B. If these edges result in a transfer from one unit to another, then a data transfer node is added along that edge. This includes multi-level paths if a direct transfer path is not available on the target architecture.

The Split-Node DAG structure can easily incorporate complex instructions, as well as basic operations by utilizing an initial pattern matching phase that detects which nodes in the original expression

[3]Operation grouping and functional unit assignment are the main tasks in instruction selection for VLIW processors.

[4]These data transfers are inserted between pairs of operations executed on different functional units.

```
Explore possible split-node functional unit assignments
   - Estimate cost of assignment
   - Select several lowest cost assignments to explore in further detail

Foreach selected assignment
   - Insert required data transfers
   - Generate all maximal groupings of nodes that could be executed in
     parallel
   - Select a minimal-cost set of maximal groupings that covers all nodes

Final solution is the lowest-cost solution found above
```

Fig. 5. **Overall Algorithm for Covering the Split-Node DAG**

DAG can be covered by a complex instruction supported by the target processor. When generating the Split-Node DAG, additional nodes and edges corresponding to the matched complex instructions are added in addition to the basic operation matches.

C. *Control Flow*

AVIV receives a collection of basic blocks connected by control flow information as an input. Code is generated for the basic blocks using the methods presented in Section IV. Code corresponding to control-flow instructions (e.g., jumps) needs to be generated. Conventional tree-covering algorithms are used for this step. Since AVIV currently targets minimum code size, control flow optimizations such as speculative execution which improve performance at the expense of code size are not incorporated.

IV. CODE GENERATION USING THE SPLIT-NODE DAG

The goal of the code generator is to cover the Split-Node DAG with a minimal-cost set of target processor instructions. Our methodology addresses the instruction selection, resource allocation, and scheduling phases of code generation concurrently because these phases are mutually dependent, thus performing them sequentially generally results in non-optimal code. Covering the Split-Node DAG, using the AVIV code generator, performs functional unit assignment, operation and data transfer grouping into instructions, register bank allocation, and scheduling. Detailed register allocation and peephole optimizations are performed as a second step.

Figure 5 describes our overall algorithm for covering the Split-Node DAG using a minimal-cost set of instructions. We introduce multiple heuristics in order to reduce the runtime of our algorithm without sacrificing the quality of the results. The algorithm begins by exploring the possible split-node functional unit assignments and selecting several of the lowest cost assignments to explore in further detail. The selection is made based on a heuristic cost function described in Section IV-A. Each of the selected assignments is then explored in detail. First, the data transfers required for the given functional unit assignment are added (Section IV-B). Next, the nodes in the current assignment, including the transfer nodes, are merged into *maximal groupings* of nodes that can be executed in parallel (Section IV-C). The maximal groupings correspond to VLIW instructions. A heuristic selection process (Section IV-D) then covers the nodes in the current assignment with a minimal-cost set of maximal groupings. An accurate cost reflecting the number of instructions required to cover all the nodes is maintained. Finally, the assignment resulting in the lowest cost is selected as the final solution.

A. *Exploring the Split-Node Functional Unit Assignments*

The first step in covering the Split-Node DAG is to select the target processor operation (i.e., functional unit) that should cover each split-node. For the Split-Node DAG of Figure 4, node N_1 or N_2 would be selected to cover the subtract split-node, node N_{10} or N_{11} would be selected to cover the multiply split-node, and so on. The number of possible split-node covering assignments can be quickly calculated by multiplying the number of possible target processor operations covering each split-node (i.e., for this example $2 \times 2 \times 3$). This is a very small basic block that results in few possible assignments. However, for typical basic blocks, the multiplicative growth in the number of possible split-node functional unit assignments quickly makes it prohibitively CPU-intensive to explore all possible cases.

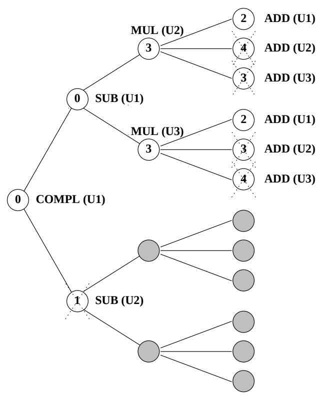

Fig. 6. **Pruning the Search Space of Split-Node Assignments**

Thus, the first step of our algorithm is to prune the search space by selecting only a few of the split-node functional unit assignments to explore in depth. This selection is made based on a cost function that takes into account the two main factors contributing to higher cost (measured in number of instructions) of covering the Split-Node DAG. These two factors are the amount of parallelism that is foregone due to the split-node covering assignments, and the number of data transfers required for the given assignment. It is important to include both factors in the cost function because this allows us to optimize the instruction selection and scheduling phases of code generation concurrently.

Calculating this cost function for all possible functional unit assignments can in itself take too long. Thus, we prune the search space of possible assignments by calculating an incremental cost for each split-node encountered and continue the search only for split-node assignments with minimum incremental cost. The split-nodes are tested in order of increasing level from the top of the Split-Node DAG. For the purpose of illustrating this cost function, let us assume that the `SUB` nodes in the Split-Node DAG of Figure 4 feed into a sink node that is a complement (`COMPL`) function that can only be executed on unit U1. Also let us assign a cost of 1 for each required transfer and for each node that cannot be executed in parallel due to the current functional unit assignment. Figure 6 shows the incremental cost at each node as well as the locations where the search space was pruned (marked by an X).

The incremental cost of the `SUB` node executed on unit U1 is 0 because it does not require a transfer to the `COMPL` node which is also executed on U1, and it does not preclude any possible parallel execution. The `SUB` on U2, however, has a cost of 1 because it requires a data transfer to unit U1 for the `COMPL` function. Thus, the search space is pruned at the `SUB` on U2 node. The incremental cost of implementing the `MUL` operation on unit U2 or U3 are the same, thus both paths are explored. Let us now examine the paths that include the `MUL` operation on unit U2. Performing the `ADD` on unit U1 introduces an incremental cost of 2 due to the two transfers required to load its operands. It does not require a transfer to the `SUB` operation because both are executed on unit U1. Also, it does not preclude the possible merging of the `ADD` with the `MUL` because the `MUL` operation is executed on unit U2. The `ADD` on unit U2, on the other hand, incurs an incremental cost of 4 (2 for its two operands, 1 because a transfer to the `SUB` operation is required, and an additional 1 because it precludes the merging of the `ADD` and `MUL` operations since both are executed on the same unit). Thus, the result, for this example, would be to select the two assignments where both the `SUB` and `ADD` operations are performed on unit U1.

B. *Adding Required Transfers*

For each selected split-node covering assignment, the required data transfer nodes are then added. In the case of a single data transfer path, this step is straightforward because for each pair of nodes that requires a data transfer between them, there is only one possible selection of data transfer node. However, in architectures containing multiple transfer paths, there may be more than one possibility

	N_2	N_9	N_{10}	N_{14}
N_2	0	1	1	1
N_9	1	0	0	1
N_{10}	1	0	0	0
N_{14}	1	1	0	0

Fig. 7. **Matrix for Finding Maximal Cliques**

```
gen_max_clique( clique, index ) {
    for (i = 0; i < number_of_nodes; i++) {
        if (i can be executed in parallel with all the nodes in the
            current clique)
            if (adding i will not preclude adding any other node)
                if (i < index)          // Pruning condition
                    return;
                else
                    add i to clique;
    }

    for (i = 0; i < number_of_nodes; i++) {
        if (i can be executed in parallel with all the nodes in the
            current clique)
            gen_max_clique(clique with i added, MAX(i, index));
    }
}
```

Fig. 8. **Pseudo-Code for Generating Maximal Cliques**

for any given data transfer. Thus, the problem resembles our initial problem of selecting the best split-node covering assignments where the number of options grows multiplicatively. Once again we apply our heuristic in order to select among the possible transfer paths, except that now the cost function is based solely on parallelism.

C. Maximal Groupings

Once we have selected a set of possible split-node covering assignments and their corresponding data transfer nodes, we want to explore these assignments in depth to find the one that will result in the minimum-cost set of target processor instructions to cover the nodes in the assignment.

The rest of this section will use the term *assignment* to refer to the collection of functional unit assignments made to cover all the split-nodes, along with their associated transfer nodes.

The goal is to examine the nodes in a given assignment and merge them into groups of nodes that can be executed in parallel on the target processor. Each grouping corresponds to a VLIW instruction. We then want to select the minimal-cost set of VLIW instructions that can cover all the nodes in the assignment.

In order to reduce the total cost of the VLIW instructions required to cover the Split-Node DAG, it is preferable to select instructions that make good use of the parallelism provided in the target processor. Thus, the groupings of nodes that we examine are all the maximal groupings, or **maximal cliques** of parallel nodes. In other words, subsets of a larger clique are not considered as a possible grouping. Note that every node in the assignment being explored is covered by at least one clique. It is possible for a clique to contain only one node. Also, the maximal cliques group operation nodes together with data transfer nodes.

C.1 Generating the Maximal Cliques

We create an $N \times N$ matrix representing pairwise parallelism, where N is the number of nodes in the current assignment of the Split-Node DAG. This matrix contains a 0 for element $[i,\ j]$ if N_i can be executed in parallel with N_j, and 1's elsewhere. Any two operations using different functional units and having no directed path between them can be performed in parallel. Figure 7 shows the matrix for a proposed assignment to the Split-Node DAG of Figure 4 consisting of nodes N_2, N_9, N_{10}, and N_{14}. Row N_{14}, for example, specifies that an `ADD` on unit U3 (N_{14}) can be performed in parallel with a `MUL` on unit U2 (N_{10}).

The maximal cliques generated for this example would be:
($C_1 : N_2$), ($C_2 : N_{10}, N_9$), ($C_3 : N_{10}, N_{14}$).

The pseudo-code shown in Figure 8 describes our algorithm for generating all the maximal cliques using the pairwise parallelism matrix. The routine **gen_max_clique** is called from a parent routine within a for-loop which iterates through all the nodes as the starting clique node. The value of the argument index is set to the starting clique node number.

Within the **gen_max_clique** routine, the first loop adds all the nodes that do not preclude the addition of any other nodes to the current clique. When no more nodes can be added that meet this criterion, the second loop spawns off several recursive calls to **gen_max_clique** where each call has one new node added to the clique. The new node can be executed in parallel with all the nodes in the current clique; however, it does preclude the addition of some other nodes. This process is repeated with the new clique until no new nodes can be added, at which point a maximal clique has been found.

The pruning condition follows from the fact that if i < index, then we will have already generated all the cliques that will be generated in this (recursive) call. Thus, we can terminate this branch. For example, suppose that we have a clique $c = \{j\}$ and we are adding a node i to the clique that does not preclude the selection of any other node. If $i < j$, then we would have already generated maximal cliques from clique $\{i, j\}$, when we started with clique $\{i\}$ as the seed.

C.2 Reducing the Number of Maximal Cliques Generated

Generating all of the maximal cliques is the most time consuming portion of our algorithm. In order to improve runtime, we have implemented a heuristic that only allows the merging of nodes whose level from the bottom and level from the top of the solution DAG are close to each other. It is generally not a good idea to merge nodes whose levels from top or bottom are very different because chances are that such a merging would preclude the merging of other nodes that would allow greater use of parallelism. We found that incorporating this heuristic improves runtime because fewer maximal cliques are generated, and maintains the quality of our results.

C.3 Eliminating Illegal Instructions

The maximal clique generation phase merges all nodes that have no data dependence between them and are executed on different functional units into single instructions (i.e., cliques). Merging based solely on this criterion is insufficient to guarantee legality of the instructions on the target processor. Illegal groupings are described in the Constraints section of the ISDL description. Thus, each proposed instruction must be compared with the constraints of the target processor. If the constraints are not met, then the illegal instruction, or maximal clique, is split into instructions with smaller cliques until all the constraints are met.

D. Selecting a Minimum-Cost Set of Maximal Cliques

Once the maximal cliques have been generated, the next step is to find the minimum-cost set of cliques that cover all the nodes in the assignment.

Our covering algorithm begins with an empty solution set. It then selects a maximal clique that covers the largest number of remaining uncovered nodes whose children have all been covered (i.e., nodes at the bottom of the Split-Node DAG will be scheduled before nodes that depend on them, thus creating a schedule as the cliques are selected) and whose register requirements do not exceed the available resources. The available resources are determined by performing a liveness analysis on the selected nodes and maintaining a running upper bound on the number of required registers for each register bank. After selecting the clique, the remaining cliques are shrunk so that they no longer include any of the covered nodes. This process is repeated until all the nodes have been covered.

In the case of a tie, where several cliques cover an equal number of nodes, the algorithm differentiates among them by using a lookahead cost function that estimates the number of cliques required to cover the rest of the nodes, if that particular clique was added to the solution set.

In the event that all the remaining cliques contain nodes that would require a spill to memory in order to satisfy the register resource constraints, the algorithm determines which covered node should be spilled. The decision is made based on the most needed resource, and the number of parent nodes that would later require the spilled value to be reloaded from memory. Once the node to be spilled is selected, the required load and spill nodes are added

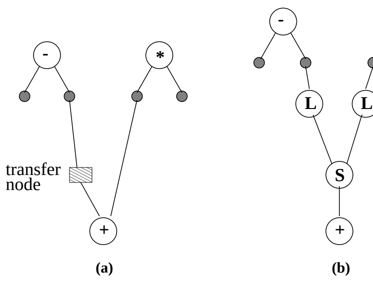

Fig. 9. **Inserting Loads and Spills into the Split-Node DAG (a)** the original Split-Node DAG, **(b)** the Split-Node DAG augmented with load (L) and spill (S) nodes

to the Split-Node DAG, and any transfer nodes that are no longer required are removed, as shown in Figure 9. Here the **+** node was selected as the node to spill, and the transfer node between the **+** and **-** nodes was removed. New maximal cliques are then generated for all the remaining uncovered nodes including the new load and spill nodes. The covering algorithm then continues with the new maximal cliques and the remaining uncovered nodes.

E. The Covering Solution

The assignment that required the minimum-cost set of cliques to cover all its nodes is selected as the final assignment. The order in which cliques were selected determines a schedule of the cliques (i.e., instructions). The final covering solution is thus a minimal-cost set of shrunk maximal cliques that cover the Split-Node DAG. It implies that unit assignment has been made, operation and data transfer nodes have been merged into VLIW instructions, register bank allocation has been performed including the addition of load and spill nodes when necessary, and a schedule has been determined. The remaining task is that of detailed register allocation.

F. Detailed Register Allocation

We perform detailed register allocation using conventional graph coloring algorithms [5]. We are guaranteed to be able to color each register bank graph using the given number of registers because we have analyzed the variable lifetimes in the instruction selection and scheduling step. However, it is possible that not all the inserted load and spill operations are required because the lifetime analysis used for inserting these operations is pessimistic.

G. Peephole Optimization

If, after performing detailed register allocation, it is determined that a particular load or spill is not needed, peephole optimization will be performed in an attempt to improve the schedule. It will remove the unnecessary loads and spills and try to compact the schedule by moving other operations into the empty slots if the dependency constraints allow it. This may, or may not, reduce the final number of required instructions.

V. Related Work

This section presents a brief survey of compiler literature that pertains to retargetable compilers for embedded systems, along with a comparison to our own work. We briefly review several representative research projects in this area: MIMOLA [6], FLEXWARE [7], CHESS [8], and the ILP-Based approach of Wilson et al. [9].

A. Mimola

MIMOLA's distinguishing feature is a microcode compiler that infers rules for code generation directly from a *structural* description (e.g., a net-list) of the target architecture instead of a *behavioral* description (e.g., the instruction set). The advantage of this approach is that the same machine description is used for both the synthesis of the target architecture and the generation of machine code. However, it is more difficult for the compiler to find code optimizations without having explicit behavioral information.

B. FlexWare

FLEXWARE includes a retargetable code generator, CODESYN, that takes an algorithm expressed in a high-level language and maps it onto a user defined instruction set to produce optimized machine code for a target ASIP or a commercial processor core. It uses a mixed structural and behavioral level model. This mixed model includes an enumeration of all possible partial instructions (i.e., microinstructions), an abstract netlist describing the data path topology, and a definition of the register classes.

CODESYN first converts the high-level source program into a hierarchy of Control-Data Flow Graphs (CDFGs). A distinguishing feature of CODESYN's pattern matching phase is its use of pre-sorted tree-like patterns that allow for pruning of the search tree of possible target processor instruction matches [10]. It determines the best implementation of the CDFG on the target processor, including the support of complex instruction recognition and utilization. Global scheduling and register allocation are performed. Finally, the compaction, assembly, and linkage phases produce the machine code.

The general framework of CodeSyn is similar to our own. However, AVIV is more focused on determining a globally optimum solution which considers instruction selection, resource allocation, and scheduling concurrently, rather than performing the various code generation steps sequentially.

C. Chess

CHESS is a retargetable code generation system targeting fixed-point digital signal processors and ASIPs. It generates machine code for the target processor, described in the nML language, and provides feedback as to how well suited the target processor is for the given application.

The nML target processor description is translated into an Instruction-Set Graph (ISG), a mixed structural and behavioral representation of the processor. The ISG models connectivity and encoding restrictions, as well as structural hazards.

The code generation process first translates the input algorithm into a CDFG. Code selection covers the CDFG with patterns that correspond to partial instructions supported by the instruction set, called bundles. Rather than making an exhaustive list of all possible bundles, they bundle instructions on the fly by searching for valid paths in the ISG. In CHESS each phase of code generation is performed separately; however, in order to ensure phase coupling, an intermediate scheduling view is constructed after each phase which takes into account constraints imposed by any previous phase.

CHESS is driven by nML which differs from ISDL, particularly in the way constraints are handled. Constraints in ISDL allow the code generator to treat all operations as completely orthogonal. Illegal operation combinations are then removed by comparing the maximal cliques to the constraints. nML, on the other hand, is an attributed grammar where production rules define the instruction set [11]. Therefore, all legal groupings of operations must be explicitly listed. Thus, ISDL descriptions can be more concise, and, as a result, easier for the code generator to handle. Further, as stated above, each phase of code generation is performed separately, though not independently. Again, AVIV is focused on solving the various code generation problems concurrently.

D. Wilson et al.

Wilson et al.'s Integer Linear Programming (ILP) based approach to code generation is based on a behavioral model of the target processor. Similar to FLEXWARE and CHESS, code generation begins by translating a high-level source language into a Data-Flow Graph (DFG). The designer can supply optimization hints. Pattern matching is used to recognize complex instructions. Next, a possible schedule that attempts to minimize overhead costs such as spills to memory and address calculations is identified. Register assignment, including necessary spills to memory, is then performed.

The cornerstone of all the optimizations is an ILP solver that can simultaneously do scheduling, instruction selection, register assignment, and compaction, as well as select from alternative spill and addressing candidates. Since all the constraints are considered simultaneously in the ILP formulation, trade offs can be made between the various optimizations leading to a globally optimal solution.

The problem with using ILP solvers for code generation is that finding the optimal solution is far too CPU intensive. The ILP solver does not have sufficient information about the structure of the problem in order to make intelligent decisions about how to prune

Basic Block	Original DAG #Nodes	Split-Node DAG #Nodes	#Registers per RegFile	#Spills Inserted	#Instr In Solution By Hand	#Instr In Solution Aviv	CPU Time (secs)
Ex1	8	30	4	0	7	7 (7)	0.1 (0.2)
Ex2	13	56	4	0	10	10 (10)	0.4 (37.2)
Ex3	11	55	4	0	13	13 (13)	0.9 (122.3)
Ex4	15	81	4	0	16	16 (16)	8.2 (41,466)
Ex5	16	106	4	0	14	16 (14)	10.7 (89,337)
Ex6	15	81	2	2	18	22 (18)	6.9 (29,072)
Ex7	16	106	2	1	15	18 (15)	9.9 (64,748)

TABLE I
Code Generation Experiments for the Example Target Architecture

Basic Block	Original DAG #Nodes	Split-Node DAG #Nodes	#Registers per RegFile	#Spills Inserted	#Instr In Solution By Hand	#Instr In Solution Aviv	CPU Time (secs)
Ex1	8	17	4	0	8	8	0.1
Ex2	13	28	4	0	11	12	0.2
Ex3	11	23	4	0	13	13	0.7
Ex4	15	29	4	0	16	17	3.0
Ex5	16	51	4	0	15	15	11.4

TABLE II
Code Generation Experiments for the Target Architecture II

the search space. Thus, in general, user supplied hints are required to produce good code within a reasonable amount of CPU time.

VI. Experiments and Ongoing Work

We have implemented a preliminary version of the retargetable code generator and run some experiments on several basic blocks. These examples are generic basic blocks that occur in DSP application code. Examples 1-2 are simple basic blocks that are found as part of a conditional statement or loop. Examples 3-5 are simple basic blocks of loops that have been unrolled twice.

Aviv generated code for these basic blocks targeting minimum code size. The results for the example target architecture of Figure 3 are summarized in Table I. The table summarizes the number of nodes in the original DAGs of the basic blocks examined, as well as, the number of nodes in the equivalent Split-Node DAGs for the example architecture. It compares the number of instructions found by hand-coding, to the solution found by Aviv. The hand-coded results are all optimal. The number of instructions equals the number of clock cycles required to execute the basic block on our target architecture. Examples 1-5 were run with four registers per register file. These examples did not require any spills to memory. Examples 4 and 5 were then rerun with two registers per register file to show what happens when the required number of registers exceeds the available resources. Both of these examples resulted in spills to memory, and their results are presented in examples 6 and 7 respectively. Note, however, that the optimal solutions for examples 6 and 7 did not require spills. These solutions were not found by Aviv because the initial functional unit assignment cost function did not detect that the functional unit assignments it made would result in spills to memory.

We see that Aviv's results are quite close to optimal. The CPU times shown for finding the solutions were measured on a Sun Microsystems Ultra-30/300. Aviv incorporates multiple heuristics that can be turned off if desired. For example, rather than selecting only a few of the possible split-node assignments, it can generate all possible assignments. Also, heuristics such as using the level from top and bottom to reduce the number of maximal cliques generated, can be turned off. In parentheses, we present the results of running these examples with the heuristics turned off. Note that turning off the heuristics does not result in an exact algorithm for code generation since we do not explore all possible schedules. It is clear that our pruning heuristics work very well, and generate the same quality results within a fraction of the CPU time required to find the optimum solution.

Our purpose in developing the Split-Node DAG structure, and heuristic algorithms for covering it, was to enable retargetable code generation. With this in mind, we changed the target architecture of Figure 3 by removing the `SUB` operation from functional unit `U1`, and completely removing functional unit `U3`. The results using this architecture are summarized in Table II. As can be seen, for several of these basic blocks, removing a functional unit does not degrade performance. The flexibility of the Aviv retargetable code generation system allows for the exploration of different architectures until the best one is found.

We are currently working on modifying the initial functional unit assignment cost function to incorporate register resource limits so that it can detect assignments that are likely to require spills to memory. In addition, we are working on peephole optimization methods that will be applied after detailed register allocation.

Acknowledgments

This research is supported by NSF Contract MIP-9612632.

References

[1] R. K. Gupta and G. De Micheli. Hardware–Software Cosynthesis for Digital Systems. *IEEE Design & Test of Computers*, pages 29–41, September 1993.

[2] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proceedings of the* 34^{th} *Design Automation Conference*, pages 299–302, June 1997.

[3] Stanford Compiler Group. *The SUIF Library*, 1.0 edition, 1994.

[4] SPAM Research Group. *SPAM Compiler User's Manual*, 1.0 edition, 1997.

[5] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register Allocation via Coloring. *Computer Languages*, 6:47–57, 1981.

[6] P. Marwedel. The MIMOLA Design System: Tools for the Design of Digital Processors. In *Proceedings of the 21th Design Automation Conference*, pages 587–593, 1984.

[7] P. Paulin et al. FlexWare: A Flexible Firmware Development Environment for Embedded Systems. In *Code Generation for Embedded Processors*, pages 67–84. Kluwer Academic Publishers, 1995.

[8] D. Lanneer et al. CHESS: Retargetable Code Generation for Embedded DSP Processors. In *Code Generation for Embedded Processors*, pages 85–102. Kluwer Academic Publishers, 1995.

[9] T. Wilson et al. An ILP-Based Approach to Code Generation. In *Code Generation for Embedded Processors*, pages 103–118. Kluwer Academic Publishers, 1995.

[10] C. Liem, T. May, and P. Paulin. Instruction-Set Matching and Selection for DSP and ASIP Code Generation. In *Proceedings of the European Design and Test Conference*, pages 31–37, Feb 1994.

[11] M. Freericks. The nML Machine Description Formalism. Technical report, TU Berlin, 1993.