An Integrated Approach to Retargetable Code Generation

Tom Wilson, Gary Grewal, Ben Halley, Dilip Banerji
VLSI-CAD Group
University of Guelph
Guelph, Ontario, Canada N1G 2W1

Abstract

Special purpose instruction set processors (ISPs) challenge compilers because of instruction level parallelism, small numbers of registers, and highly specialized register capabilities. Many traditionally separate subproblems in code generation have been unified and jointly optimized within a single integer linear programming (ILP) model. ILP modeling provides a powerful methodology for for generating high-quality code for a variety of ISPs.

1 Introduction

Instruction Set Processors (ISPs) are finding increasingly wide use in commercial products, because of their inherent flexibility and versatility. Compilers capable of generating code for these chips would dramatically hasten their product's time to market, but such compilers are not generally available.

One problem is the high quality of code that is often demanded by the applications . ISP chips are commonly employed for digital signal processing (DSP) with attendant real-time requirements. Furthermore, if the chip is specially designed to optimize a particular application, it often has "just enough" hardware to perform the function. The consequence is often an unconventional or heavily constrained architecture, for which available optimizing compilers are poorly suited. Awkward architectural features make generation of high quality code a difficult task for any one of the special ISP chips, not to mention a family of different chip designs.

We have developed both a methodology and a working prototype that can generate very high quality code for a variety of special purpose ISP architectures. We model the entire code generation problem as an integer linear program (ILP), which provides an integrated approach to several traditionally separate subproblems in code generation. We not only have a unified model of the problem, but one which is sufficiently abstract that it can be easily adapted to important variations in the target architecture. We believe this is the first such integrated approach to retargetable code generation.

We have concentrated on ISP chips that are specially designed for DSP applications. The chips in question have a single ALU that can do a multiply and accumulate in a single instruction cycle. Although the instruction repertoire is limited, the instruction words are long, permitting several things to occur in parallel during each cycle. These include: one ALU operation on data; concurrent movement of data to or from the data memory; loading a register with a constant from a field of the instruction itself; preparing address registers with a value for the next access to data memory; and updating the program counter, possibly to implement a conditional branch. The ALU is surrounded by a small number of registers whose capabilities are quite varied and specialized. Only certain ones may be used for ALU operations, for writing to data memory, extracting constants from program memory, generating addresses, etc. And different registers have different combinations of capabilities!

Much of the relevant published research has been oriented toward machines with less instruction-level parallelism and a friendlier ensemble of registers. The instruction-level parallelism tends to engender a pipelined programming style, in which an address is prepared on one cycle, used to obtain an operand on another, with the operand being used on yet another. The parallelism among several such operand streams differs markedly from the operation parallelism that conventional compilers might be able to handle.

Most register assignment schemes have stressed general purpose registers, and have focused on live variable conflicts in the assignment process, [1-4] for example. Special purpose registers inspire heuristic assignment techniques and provide the major bottleneck when compiling code for ISP chips. Most published code generation methods, such as [5-8], approach the

0-8186-5785-5/94 $03.00

problem by solving a succession of subproblems, usually employing heuristics. One approach [9,10] uses extensive peephole optimization to make the most of unoptimized generated code. In contrast, we not only have a code generator that considers the entire problem at once to obtain an integrated solution, but one that thrives on special purpose registers.

2 Integrated Code Generation

2.1 The Subproblems that are Included

We begin with a data flow graph (DFG), which is generated by the front end of an appropriate compiler. The following subproblems must be handled by a code generator:

1. Map combinations of generic DFG operations onto more inclusive machine instructions, such as multiply and add;
2. Schedule operations on control steps and bind them to specific functional units;
3. Assign data (and address) values to registers when possible;
4. In case the number of live values exceeds the number of registers, introduce *spills* which temporarily store certain values in memory;
5. Introduce register-to-register copies at certain points to resolve problems with special pupose registers and with values that wrap around loops;
6. Assign registers consistently across control block boundaries and around loops;
7. Correctly compact the individual components of (highly parallel) machine instructions into a minimum number of final instructions.

2.2 Important Concepts of the Model

Our solution is based on an integer linear program. Thus we model all of the subproblems mentioned above in terms of linear inequalities, whose combined effect is to specify the correctness criteria of any feasible solution. Since all the constraints are considered together by the ILP, any solution can relate and trade off issues that arise within the various "subproblems", thus providing an integrated result.

Although the actual constraints in an application of the model reflect the realities of a particular architecture, the model is expressed in general terms, such as sets of registers that could be used for certain DFG edges and "patterns" that represent more inclusive instructions and which cover parts of the DFG. This gives the model its inherent retargetability. What one needs to do is change the numbers of registers, sets of allowable registers for various operations, and patterns depicting instructions that the machine can support.

An important idea is the way we approach register assignment. We have abandoned the conventional view that stresses "live variable conflict" and mutual incompatibility, in favor of a *scheduling* view. Whenever two DFG edges are mapped to the same register, they must use that register in some order; one live value must be consumed before the other is generated. This is essentially the same as sequentially scheduling other objects (like operations) onto other non-sharable resources (like functional units). This notion allows special purpose registers to be handled in a systematic and uniform way.

Another basic idea is that our DFG contains options from which the ILP may choose. The DFG includes "extra" edges and operations, which represent alternative possibilities for use in the final code. For example, optional spill or copy operations are inserted into the DFG at places where spills or register copies might be useful, depending on overall characteristics of the solution. They appear in the generated code only if they are required for a correct solution. Another example concerns alternative modes of array addressing, exactly one of which must appear in the final design.

Similarly, the patterns, which relate groups of generic DFG operations to more inclusive machine instructions, may or may not be chosen as final instructions. The same generic DFG operation may have several candidate patterns that potentially subsume it; patterns chosen in a final solution cannot overlap with each other, and they affect which edges require registers in the final design. Operations, edges and patterns that represent chosen elements in the generated code are called *active*.

Our model also supports implication among optional objects. For example, some array could be addressed either by (re)computing its base plus current offset at each point of access, or by reserving an address register and traversing the array using autoincrement. Within certain regions of the code, use of the address register at one reference point might imply its use at another. Similarly, selection of an optional spill at one point requires using related spill code at associated points that access the same value.

Another useful feature is assignment of the same register to a set of edges in the data flow graph (DFG).

One application involves linking up "cyclic" edges, where one edge represents a value leaving a loop, and another represents a value – perhaps the same – (re)entering the loop. In general, ability to assign the same register to different edges enables the correct interconnection of control blocks when several blocks are being considered at once.

To support such flexibility, the ILP contains several constraints that are dynamically enabled or disabled, depending on the current values of solution variables.

3 The ILP Model

3.1 Operational Assumptions

We assume that a compiler front-end has produced a DFG and that generic operations have been replaced by operation sequences that are appropriate to the given architecture, for example, to correctly handle data path widths. A set of potential *covering patterns* has also been identified but not yet specifically selected. These are groups of primitive DFG operations which can be combined into single instructions by the architecture. A classic example is multiply and add. When such a pattern is chosen, it subsumes the entire DFG within it – both operation nodes and edges.

The model presented here was developed for machines with only a single non-pipelined ALU capable of data manipulation. An ALU operation can occur in parallel with one or two transfers of data between internal memory and operand registers or accumulators. Registers used for memory addressing may be updated in certain standard ways, such as autoincrement, during regular instruction execution. The Motorola 56001 exemplifies such an architecture.

If certain registers are restricted for use with only certain memories, we assume in this model that memory-resident data has already been allocated to a particular memory. This assumption allows us to know the specific family of resources required by each data manipulation, data movement, and address calculation operation. This, in turn, permits recognition of any potential conflict in resources. Such possibilities must then be circumvented by appropriate scheduling options within the model.

The model makes no distinction between single-block and multiblock designs. A DFG spanning several control blocks must be structured in such a way that operations remain within a logically acceptable set of control blocks. This can be done by introducing "dummy" nodes to depict points where control branches and merges. Additional node ordering constraints keep certain operations between relevant branch and merge points. Values traversing block boundaries may either be represented on inter-block edges or be associated through use of the same register on logically connecting edge segments. Inter-block code movement as such is not addressed here.

3.2 Overview of the Model

There are two basic ways of executing the ILP model. The simplest and fastest seeks any feasible solution and requires no objective function. Presumably the number of control steps, t, is prespecified, and any correctly formed solution within the required number of control steps is considered equivalent in practice to any other. The alternative approach treats t as a variable and seeks a minimum time solution, using the objective function:

$$\min(t)$$

Other cost parameters can also be included, but the running times to find an "optimum" in one attempt can be somewhat longer.

The constraints are the correctness criteria for any solution. The following list summarizes what the constraints guarantee and conveys the general strategy of our model.

- a basic DFG operation can be included in at most one active pattern;
- a DFG operation is active if it is not covered by an active pattern and if it is met by at least one active edge;
- an edge is active if it is essential to the design and not totally within an active pattern;
- an edge is inactive if it is totally within an active pattern;
- certain edge sets are either all active or all inactive (because they belong to the same alternative implementation);
- certain edge sets may have at most one active member (because they represent alternative implementations);
- active edges must be assigned to a register that belongs to an allowable register set;
- edges that represent the same value, within a control block, between control blocks, or wrapping around a loop, must use the same register (except for a copied value);

- active edges impose a strict scheduling order between the operations that they connect;
- active operations that are not related by a path in the DFG, but that require the same functional unit, must use that unit in some sequential order;
- active edges that are not members of the same path in the DFG, but that are assigned to the same register, must use that register in some sequential order.

3.3 Definitions and Variables

The following symbols index scalar objects:

p	covering pattern
i, j, h, k	operation (DFG node)
r	register

An edge of the DFG is identified by the ordered pair of nodes it connects, e.g., edge (i, j). Ordered pairs of nodes are used for many other purposes besides identifying edges within the ILP model.

The model determines which combining patterns, operations and DFG edges are to be "active" or selected in the final design, the control step for each activated pattern and operation, and the register assigned to each active DFG edge. These results are conveyed through *solution variables*, some of which are nonegative integers and others of which must be either 0 or 1. The solution variables (y,t,z,x,u), plus two auxiliary variables (p,q) for internal use by the ILP, are defined below:

variable	*type*	*meaning*
y_i	intgr	step where op_i scheduled
t	intgr	final step = total time
z_p	0-1	if 1, pattern p selected
z_i	0-1	if 1, op_i activated
x_{ij}	0-1	if 1, edge (i, j) activated
u_{ijr}	0-1	if 1, edge (i, j) uses register r
p_{ij}	0-1	if 1, op_i must precede op_j
q_{ij}	0-1	if 1, op_i cannot follow op_j

The following are *index sets* of scalar objects:

G	operations (nodes of the DFG)
F	operations that could appear last
B_i	patterns that cover op_i
B_{ij}	patterns that cover edge (i, j)
R_{ij}	registers suitable for edge (i, j)

The following are sets of *operation pairs*:

E	the DFG edges
U	potentially conflicting over ALUs
V	potentially conflicting over registers

U contains unordered operation pairs. However, E and V both contain ordered pairs. U identifies operations that: (i) use the same functional unit, bus, memory, or other non-register resource; and (ii) are not otherwise ordered by the DFG or other considerations. If (i, j) is in U, variables p_{ij} and p_{ji} must be defined. Similarly, if (i, j) is in V, variable q_{ij} will be defined. Membership means that: (i) op_i *may* consume a value from the same register in which op_j stores its result; and (ii) the operations are not otherwise preordered and could possibly conflict.

Finally, we have *sets of edge sets*:

set	index	constituent sets
A	a	mutually exclusive alternatives
D	d	mutually dependent for activation
C	c	requiring same register

Exactly one edge from each set A_a must be activated. D_d may contain a set of edges that all require the same activation status. It may also contain an "implicant" edge whose activation implies activation of a number of other members from the set. The sets of C_c identify edges that all require the same register. Such edges are often interblock segments or segments that wrap around a loop to convey a value to successive iterations. They are logically "connected", though physically separate in the model.

3.4 The Constraints in Detail

At most one combining pattern may cover any operation:

$$\forall i \in G: \sum_{p \in B_i} z_p \leq 1 \qquad (1)$$

From any set of *alternative* edges, A_a, exactly one edge must be either activated or covered by a combining pattern. Ordinary, non-alternative edges are also handled by constraint (2). Such an edge is the sole member of some A_a and must either appear in the final design or be covered by a pattern. An edge that is not alternative and has no possible covering pattern has its $x_{ij} = 1$ from the outset.

$$\forall a: \sum_{(i,j) \in A_a} (x_{ij} + \sum_{p \in B_{ij}} z_p) = 1 \qquad (2)$$

Each active edge must be assigned exactly one register from a suitable register set:

$$\forall (i,j) \in E : \sum_{r \in R_{ij}} u_{ijr} = x_{ij} \qquad (3)$$

Membership in C_c means that edges (i,j) and (h,k) must be assigned the same register. R_c represents the intersection of register sets R_{ij} available to individual edges in C_c. Constraint (3) assigns only one register to an edge, which insures that each summation in constraint (4) selects only a single value of 'r' with a coefficient of 1. Therefore, each summation identifies the assigned register, and the constraint enforces the same assignment to both edges. (Here we assume that both involved edges are required in the design; a more general version allowing optional edges is available.)

$$\forall c, (i,j) \in C_c, (h,k) \in C_c :$$
$$\sum_{r \in R_c} (r \times u_{ijr}) = \sum_{r \in R_c} (r \times u_{hkr}) \qquad (4)$$

Constraint (5) governs the activation of sets of *optional* edges. Each edge set, D_d contains an "implicant" edge, (h,k), and 'm' total optional edges. If edge (h,k) is activated, the left-hand side becomes n, which forces the summation to be at least n. When n = m, the entire edge set, D_d is activated together or entirely left out of the final design.

$$\forall d : \text{ if } x_{hk} \Rightarrow \text{at least n edges of } D_d\text{:}$$
$$n \times x_{hk} \leq \sum_{(i,j) \in D_d} x_{ij} \qquad (5)$$

Constraints (6) and (7) govern scheduling of operations that are connected by edges of the DFG. Assuming edge (i,j) exists, constraint (6) forces op_i to strictly precede op_j in the schedule, when that edge is not covered by a pattern. (Note that even unactivated optional edges imply strict ordering – when not covered.) However, if the edge *is* covered by a combining pattern, constraint (6) allows op_i and op_j to occur on the same step and constraint (7) requires it. N is a very large constant, which makes the right-hand side of (7) large when the summation is 0. Therefore, failure to cover edge (i,j) effectively deactivates this constraint, and selection of a cover fuses the two operations into the same control step.

$$\forall (i,j) \in E : y_i < y_j + \sum_{p \in B_{ij}} z_p \qquad (6)$$

$$\forall (i,j) \in E : y_j \leq y_i + (1 - \sum_{p \in B_{ij}} z_p) \times N \qquad (7)$$

Constraints (8), (9), and (10) govern scheduling of operations that are *not connected* by a DFG edge, that use the *same resources* , and could *possibly conflict* by being scheduled on the same control step. Constraint (8) forces z_i to 1 if op_i is activated. This condition is characterized by op_i having an activated outgoing edge. (If op_i has only incoming edges, an appropriate index adjustment is required.) Constraint (9) forces either p_{ij} or p_{ji} to 1 when *both* op_i and op_j are active. Constraint (10) forces op_i to strictly precede op_j if p_{ij} is 1, and op_i to follow op_j if p_{ji} is 1. Constraint (10) prevents both p_{ij} and p_{ji} from being 1. When a 'p' value remains 0, its corresponding constraint (10) is deactivated by including (large) N. The chosen 'p' value, if any, enforces exactly one of the possible scheduling orders.

$$\forall (i,j) \in U \text{ or } (h,i) \in U : \sum_{(i,k) \in E} x_{ik} \leq z_i \times N \qquad (8)$$

$$\forall (i,j) \in U : z_i + z_j \leq 1 + p_{ij} + p_{ji} \qquad (9)$$

$$\forall (i,j) \in U : y_i < y_j + (1 - p_{ij}) \times N \qquad (10)$$

Constraints (11) and (12) perform a similar ordering function between edges that are assigned the same register and could be active concurrently if not "scheduled" in some sequential order. When edges (h,i) and (j,k) are both active *and* have been assigned the same register, constraint (11) forces either q_{ij} or q_{kh} to be 1. The former causes the destination of edge (h,i) to be scheduled no later than the source operation of edge (j,k). The other choice of q causes the destination of edge (j,k) to finish by the time edge (h,i) receives a value. The actual ordering is imposed by constraint (12). Again, when 'q' remains 0, it causes constraint (12) to be satisfied automatically.

$$\forall i,j,h,k \text{ where } (i,j) \in V, (h,i) \in E, (j,k) \in E :$$
$$\forall r \in R_{hi} \cap R_{jk} :$$
$$x_{hi} + x_{jk} + u_{ijr} + u_{khr} \leq 3 + q_{ij} + q_{kh} \qquad (11)$$

$$\forall (i,j) \in V : y_i \leq y_j + (1 - q_{ij}) \times N \qquad (12)$$

The total number of control steps in the design is bounded below by the step on which any possible "final" operation is scheduled. If 't' is predetermined, the entire schedule is limited to 't' steps. If 't' is variable, the largest 'y' value in F determines the total number of steps.

$$\forall i \in F : y_i \leq t \qquad (13)$$

3.5 Application of the Model

Running a monolithic ILP on a large constrained problem may take considerable time. But this mode of operation is not necessary. Since all of the physical resources on the chip are already committed, the number of control steps (or their weighted sum) remains the only quantity to minimize. If M represents the minimum number of steps in which the given algorithm can run on the target architecture, then any M-step solution that satisfies the constraints is equivalent to any other. Hence, an optimum solution can usually be found very quickly by attempting a series of feasible solutions, in which the maximum number of steps allowed in each attempt is reduced from before. When the ILP fails to find a feasible solution that requires M-1 steps, we know that the previous M-step solution is the best possible.

Although this technique usually provides solutions in tens of seconds, we must still be wary of the amount of lattitude within the ILP formulation. One approach is to break a program into smaller, more manageable pieces. We first identify critical regions, such as innermost loops, optimize these, and propagate the results to surrounding regions as boundary conditions.

In addition, we either perform or propose several preprocessing steps that can further constrain the search, without precluding optimality. As an example, when one set of currently live edges can definitely use general purpose registers, we arbitrarily assign these edges to registers in advance. All such assignments are equivalent in practice, and the ILP is spared from evaluating all the equivalent permutations.

4 Conclusion

So far the code generator has been tested on fairly small examples, typically four control blocks containing about 20 to 30 operations. One reason is that the code generator is not yet fully integrated with our compiler front end. Nevertheless, the tests show object code of considerable complexity that is as good as hand-crafted code for these examples. The technique is definitely promising, especially for its ability to handle highly parallel instruction formats and heterogeneous register sets.

Our current work involves introducing several preprocessing analysis algorithms to eliminate clearly undesirable scheduling options, insert spills that will surely be required, rationalize array referencing, optimize circular buffer usage, and provide more focused constraints for the ILP itself. A new version of the ILP will handle architectures with more than one possibly pipelined ALU. Of course, we must still investigate the range of architectures over which our model is effective.

References

[1] REAL: A Program for REgister ALlocation, Kurdahi, F.J. and Parker, A.C., 24th DAC, 1987.

[2] A Global, Dynamic Register Allocation and Binding Scheme for a Data Path Synthesis System, Woo, N-S., 27th DAC, 1990.

[3] Register Allocation with Instruction Scheduling: A New Approach, Pinter, S.S., SIGPLAN'93, June, 1993.

[4] Load/Store Range Analysis for Global Register Allocation, Kolte, P. and Harrold, M.J., SIGPLAN'93, June, 1993.

[5] Validation of Hardware Descriptions by Retargetable Code Generation, Nowak, L. and Marwedel, P., 26th DAC, 1989.

[6] Open-Ended Systems for High-Level Synthesis of Flexible Signal Processors, Lanneer, D., Catthoor, F., Goossens, G., Pauwels, M., Van Meerbergen, J., DeMan, H., EDAC90.

[7] Combined Scheduling and Routing for Programmable ASIC Systems, Hartman, R., EDAC-92.

[8] DSP Design Tool Requirements for the Nineties:An Industrial Perspective, Paulin, P., Liem, C., May,. T.C., Sutarwala, S., Journal of VLSI Signal Processing, to appear.

[9] Code Generation for Streaming: an Access/Execute Mechanism, Benitez, M.E. and Davidson, J.W., Proc. 4th International Conf. on Architectural Support for Programming Languages and Operating Systems.

[10] A Design Environment for Addressing Architecture and Compiler Interactions, Davidson, J. and Whalley, D.B., Microprocessors and Microsystems, Vol. 15 No. 9, Nov., 1991.