An Integrated Approach to Optimal Code Generation for Irregular Architectures

Andrzej Bednarski and Christoph Kessler (Linköping University, Sweden)

Abstract

We employ a dynamic programming approach to solve the problem of optimal local code generation in an integrated manner, where production compilers generate code in a phase decoupled way.

Introduction

Code generation is a complex task. From a software engineering point of view it is more convenient to split this complex task into subtasks that are simpler to deal with. This phased decoupled code generation is illustrated in Figure 1 by following any path along the edges of the cube from the Intermediate Representation (“IR”) to “Target code”. Unfortunately, most important tasks in code generation, that are *instruction selection*, *instruction scheduling* and *register allocation*, are strongly interdependent. Even solving each phase efficiently can finally produce code of lower quality than expected.

Hardware irregularities such as exhibited by DSPs have stronger impact on the task interdependence. Therefore, an integrated solution that deals with the main tasks simultaneously is needed for producing code of high quality. This is illustrated in Figure 1 by the diagonal edge that directly goes from the “IR code” to the “Target code” solving concurrently the main tasks.

We are interested in building a retargetable optimizing code generation system that produces optimal code sequences for a given optimization goal (time, space, power consumption, *etc.*). The task is particularly challenging in the sector of irregular architectures, such as DSPs with non-homogeneous register sets, with specific constraints on memory banks and with irregular data paths.

Previous studies by Kessler (1998); Kessler and Bednarski (2001) show that our approach gives optimal solutions for medium size basic blocks (< 50 IR instructions) with reasonable time and space usage. We think that our framework is suitable for the following areas:

- Optimizing critical code sections: for real-time applications and embedded systems
- Evaluating heuristics employed in code generation: as research tool

Integrated Code Generation

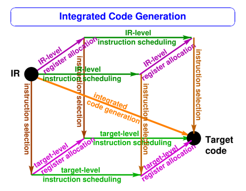


Figure 1. Task of code generation from an intermediate program as an phase decoupled approach (path along the edges of the cube) and integrated solution (follows the diagonal).

Kessler (1998) introduced a new representation of the space of partial solutions, the *selection DAG* that represents all scheduling possibilities in a compressed way. By constructing the selection DAG in a particular way that suits the optimization goal the algorithm becomes practical for medium size basic blocks as shown in Kessler and Bednarski (2001). The algorithm is based on dynamic programming and enumerates longer and longer schedules.

A *time profile* represents the occupation status of all functional units at a given time. It specifies for each functional unit the sequence of instructions that are being executed and still not completed. A time profile for a single issue architecture with three functional units is depicted in Figure 2.

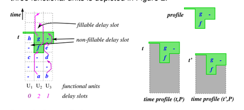


Figure 2. Time profile for a single issue architecture with three functional units.

A *space profile* contains the information in which register classes the values are located that are needed for subsequent schedules. We derive a general relation between registers and register classes from the instruction set where memory banks are a special kind of register classes.

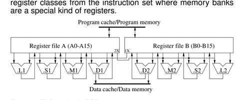


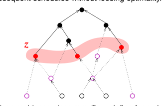
Figure 3. TI-C62x family DSP processor.

The architecture of TI-C62x DSP represented in Figure 3 exhibits 3 residence classes: register file A, register file B and main memory. At a given scheduling stage a space profile gives the location of a partial value. A value may be located simultaneously at different residence classes.
A *time space profile* is a data structure that combines time profile and space profile.

Solution

Currently we are performing concurrently instruction selection, instruction scheduling and register allocation on the basic block level where operations are represented in the form of a DAG that represents data dependences at the IR level.

We use time space profiles to represent all possible schedules produced by topological sorting of DAG nodes in a compressed way where “equivalent” nodes are grouped. Each solution node contains information about selected instructions, its register need and its data [illegible]. We proved that it is legal to group partial equivalent solutions, i.e. solution nodes with the same time space profiles, into one solution and use it as prefix for subsequent schedules without loosing optimality.



For a given residence class $\alpha \in \mathcal{P}$ we define for a given set ζ of IR nodes:

$$rpot(\alpha, \zeta) = \sum_{v \in \zeta} \begin{cases} 1 & \text{if } v \text{ resides in } \alpha \\ 0 & \text{otherwise} \end{cases}$$

We define the *residence potential* as a function *RPot* that computes, for a given zero-indegree set z and a level $l = |scheduled(z)|$ an index on the residence potential axis of the solution space:

$$RPot(z, l) = l \cdot |\mathcal{P}| \cdot |V| + \sum_{\alpha \in \mathcal{P}} rpot(\alpha, alive(z))$$

It gives a rough measure of the locality of values that are alive at a given selection step.

An *extended selection node*, is a node in the solution space that contains the time space profile data structure, and the residence potential as attribute. It contains all necessary information for future scheduling decisions.

We structure the solution space as a three-dimensional grid shown in Figure 4. The grid axes are the level, that is the length of the partial schedule, the execution time and the residence potential. A grid node is a set of extended selection nodes that have the same residence potential, level and execution time but different time profiles and/or space profiles. The structuring as a grid supports an efficient retrieval of all possible candidates for comparable partial solutions. Further it allows us to efficiently optimize the space requirement by removing solutions that are no longer used (there are no backward edges).

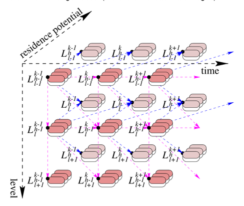


Figure 4. Structuring the space of partial solutions as a three-dimensional grid.

Figure 5 shows the complete solution space for the DAG in the right lower corner. For simplicity, we have chosen a simple two issue-architecture that consists of two functional units U_1 and U_2 respectively with latencies 1 and 2. The dashed arrows represent transfer instructions (inserted by the algorithm).

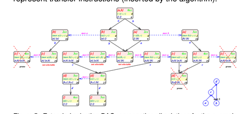


Figure 5. Extended selection DAG representing all solutions for the example DAG in the right lower corner.

Ongoing and Future Work

We plan to use the time space profiles to describe the boundaries of schedules for basic blocks and use them as connectors in the control flow graph. Thus, we plan to extend the currently local optimization on basic blocks along the program's control flow graph to perform global code generation.

At the present time we are working on a hardware description language called *ADML* that is expressed in XML format. For now ADML consists of four sections: registers, residence classes, functional units and instruction set. We are working on the extension of ADML to express constraints in the similar way as in ISDL by Hadjiyiannis et al. (1997).

Results

Previous results for time-optimal code for a homogeneous architecture are depicted in Table 1.

Source program, basic block	# IR nodes n	optimization time	
		$f=2, \omega=1, \ell=(1,2)$	$f=3, \omega=1, \ell=(1,2,3)$
FIR filter `fir.c`			
BB1 (first loop body)	16	3.5 s	4.0 s
BB2	16	8.0 s	9.5 s
BB3 (main loop body)	30	3:21:50.2 s	4:40:44.9 s
Matrix multiplication			
BB2 (main loop body)	30	1:05.0 s	1:41.8 s
same, unrolled once	40	6:08.5 s	9:47.2 s
Jacobi-style grid relaxation			
loop body, 5-point-stencil	40	1:15.8 s	1:31.8 s
loop body, 9-point-stencil	53	1:36:13.2 s	2:00:51.5 s

Table 1. Optimization times for basic blocks taken from C programs.

We have integrated the concept of space profiles into our framework. Table 2 shows our first simulation results for a TI-C62x like DSP. It gives the time and space usage for a given basic block size (#Nodes), as well as the size of the solution space (#ESDAG nodes). Column #Improved gives the number of schedules that turn out to be better only in the future. Finally, column #Merged measures the number of produced partial schedules that have an equivalent schedule already in the solution space, thus they do not need to be expanded further resulting in space optimization. The simulations have been running on a 400MHz Sun Microsystem UltraSparc-10 with 256 MB RAM and 200 MB swap.

BB	#Nodes	#ESDAG nodes	#Improved	#Merged	Mem. usage (Mb)	CPU Time (s)
Ex1.c	12	8898	9	36765	27	74
Ex2.c	14	30061	129	158120	36	279
Ex4.c	15	103015	28	499306	86	2,691
Ex3.c	17	180580	2656	1264740	95	6,135

Table 2. Code generation experiments for a TI-C62x like architecture.

Contact Information

Dr. Christoph Kessler
http://www.ida.liu.se/~chrke
chrke@ida.liu.se

Andrzej Bednarski Ph.D. student
http://www.ida.liu.se/~andbe
andbe@ida.liu.se

Home page: http://www.ida.liu.se/~chrke/optimist

REFERENCES

Hadjiyiannis, G., S. Hanono, and S. Devadas: 1997, ISDL: an instruction set description language for retargetability. *Proceedings of the 34th annual Design automation conference*, ACM Press, 299–302.

Kessler, C. and A. Bednarski: 2001, A Dynamic Programming Approach to Optimal Integrated Code Generation. *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems.*

Kessler, C. W.: 1998, Scheduling Expression DAGs for Minimal Register Need. *Computer Languages*, **24**, 33–53.

Programming Environments Laboratory