An Efficient Model for DSP Code Generation: Performance, Code Size, Estimated Energy

Catherine H. Gebotys
Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario,Canada N2L 3G1
cgebotys@optimal.vlsi.uwaterloo.edu

Abstract

This paper presents a model for simultaneous instruction selection, compaction, and register allocation. An arc mapping model along with logical propositions is used to create an optimization model. Code is generated in fast cpu times and is optimized for minimum code size, maximum performance or estimated energy dissipation. Code generated for realistic DSP applications provide performance and code size improvements from 1.09 up to 2.18 times for the TMS320C2x processor compared to previous research and a commercial compiler. In all examples up to 106 instructions are generated in under one cpu minute. This research is important for industry since DSP code can be efficiently generated with constraints on code size, performance, energy dissipation.

1.Introduction

High reliability, high performance and low cost are forcing higher levels of integration for VLSI embedded systems. As DSP applications are rapidly growing more complex, some designers are moving from full custom digital circuitry to programmable processors or in-house cores to obtain lower risk solutions. The DSP core is a DSP processor that can be reused and combined with program/data memory, dedicated logic, plus ASICs, and incorporated onto a large silicon chip, providing a cost efficient and flexible solution for many typical embedded applications requiring low power and high reliability. These systems demand small code size and high performance. Due to increasing complexities, high level compilation is a necessity. However the biggest drawback to both DSP processors or DSP core use is the code generation.

The use of conventional code generation techniques and even compilers specifically designed for commercial DSP processors produce very inefficient code [8,4]. There are many more limitations placed upon code generation for the DSP processor than for the general purpose processor. The difficulty arises from non-homogeneous register sets, small number of very specialized registers, very specialized functional units, restricted connectivity, limited addressing, and highly irregular datapaths[8]. For example specialized functional units such as multiplier-accumulators and address calculation units are typically found in these architectures. Instructions take operands and store results of computations in well defined registers with limited connectivity. Instructions are highly interdependent and involve the use of modes, for example the mode classes typically involve product-shift and sign-extension. Some instructions will function differently depending upon the current mode setting and extra instructions to reset the mode are typically required. Limited addressing modes and the use of address registers are also typical. Code generation for DSP processors must meet tight timing constraints dictated typically by DSP throughput, and meet tight resource constraints. Given that these DSP processors must meet these requirements using very small code space (all on chip ROM), the code generation problem is a very difficult one[10]. Typically DSP processors are difficult to use and require long product development times even though the program being developed may be stored on chip in less than 1K program ROM. The need for decreasing time to market, development costs, and maintenance costs, demands the use of high level language compilation. All of these factors imply several challenges in writing efficient code generators for such DSP processors. This is even more difficult for the design of in-house DSP processor cores since retargetable compilation is also required.

2. Problem Description and Related Work

The following problem, problem 1 given below, is an important part of the code optimization problem that will be studied in this paper. For simplicity let us assume that an algorithm to implement the application has already been assigned based upon accuracy required, low energy implementation, etc.. The algorithm is composed of basic blocks, which are given as a partially ordered list of code operations. For the problem definition below we assume that there is one target DSP processor or core defined with an instruction set architecture (instruction set architecture).

Problem 1.

Assume we are given a sequence of operations, represented by an ordered list of operations. The objective is to select instructions, compact code, and allocate registers for the target processor. The set of operations is mapped into a set of instructions, which is optimized for code size, performance, or energy dissipation.

Researchers have shown that code requiring a minimum number of cycles can be generated for expression trees[1] and an extension for general DAGs was researched in [2], however both techniques do not consider instruction-level parallelism. Researchers in [3] use graph based technique to perform code compaction, whereas in [4] a branch and bound scheduler and a heuristic search scheduler for the TMS320C2x processor are used to try to reduce the accumulator spilling and mode cost. A large integer linear programming (ILP) model was researched in [5] that performed simultaneous instruction selection, code compaction and register allocation. Unfortunately it was too complex for realistic processor representation.

Recent research in integer optimization has found that large subclasses of logical inference problems can be solved as a relaxed linear programming problem (LP) [6]. In particular if the system is composed of horn clauses[7] the optimal integer solution can be obtained from the relaxed linear programming problem solution (where binary variables are replaced by continuous positive variables with upper bounds equal to one). In the solution to the linear programming problem, any variable which does not have an integer value can be rounded up or down to the nearest integer value. It has been proven that this final integer solution is always optimal[7]. Researchers have also found that by rewriting constraints as horn clauses the computational time for solving the problem can be greatly reduced. Experience has shown that many logical inference problems can be solved as relaxed LP problems even if they are not horn clauses[7].

In this manuscript a new approach is presented to solve problem 1, DSP code generation. Unlike previous research, the target DSP processor is defined using arc mappings and propositional logic which is then automatically translated into mathematical programming constraints to form the optimization model. The basic block components of the application are transformed into a sequence of operations, which is input to the model. The model generates code for the target processor. This process supports retargetable compilation for processors whose instruction set architecture can be described using propositional logic. The next section will outline the assumptions and terminology to be used in the rest of the paper.

3.Assumptions and Notation

The following terminology will be used in this paper:

o is an operation in the basic block of an application.

$o1 \rightarrow o2$ is an arc where data produced by operation $o1$ is used as input for operation $o2$.

s is an integer valued parameter representing the set of instructions that an operation o may be mapped into.

$T(o)$ is the time that the operation o is scheduled at.

$P_{o,s}$ is a proposition representing the mapping of operation o into the set of instructions s.

$x_{o,s}$ is a binary variable representing the proposition $P_{o,s}$, when the proposition is true, the binary variable equals 1 otherwise it equals 0.

P_o is a proposition when true means that there is some parallel instruction that can be obtained by merging an instruction in the set of instructions for operation o with another instruction in the set of instructions for the next operation, $o1$, where $T(o)=T(o1)-1$.

p_o is the binary variable representing the parallelism proposition P_o.

$End(o,s)$ is the end resource being used to store the data output from this operation, o. For example the output data may be stored in a register or in memory (and accessed as a memory operand by another operation).

$E(o)$ is a parameter that identifies the type of arc for operation o. For example if $o \rightarrow o2$, where o is a multiplication and $o2$ is an addition then the type of arc for o, $E(o)$, is a *multiply-add* arc.

$cycles(o,s)$ is the number of cycles required to execute the set of instructions s for operation o.

$savings(o)$ is the number of cycles saved due to implementing a parallel instruction for o and $o2$, where $T(o)=T(o2)-1$.

$memaccesses(o,s)$ is the number of memory accesses performed for the set of instructions s for operation o (including memory operand accesses and the instruction accesses).

$msavings(o)$ is the number of memory accesses not required anymore due to implementing a parallel instruction for o and $o2$, where $T(o)=T(o2)-1$.

$codesize(o,s)$ is the number of instructions in the set of instructions s for operation o.

$csavings(o)$ is the number of instructions not required anymore due to implementing a parallel instruction for o and $o2$, where $T(o)=T(o2)-1$.

The rest of the paper will illustrate the model, constraints, and generated code results using the TMS320C2x processor[15]. The partial instruction set architecture for this processor is shown in table 1 (where || *Instr* refers to parallel instructions). This popular DSP processor has been described as having a very restrictive type of instruction level parallelism, making compaction a nontrivial task[8]. The optimization-based techniques presented in this paper provide a fast approach to generating optimized code. It is a retargetable model that can be used for modeling other DSP processors as well. The next section will provide an introduction to propositional logic and how it can be used to describe instruction set architecture constraints and mapped automatically into constraints of an ILP. The following sections, illustrate how arc mappings and ILP constraints are defined. Examples will be used to illustrate how optimization for code size is performed. Finally examples will be presented to show that new energy optimized solutions are produced unlike approaches using previous research or existing commercial compilers.

Table 1. Partial ISA TMS320C2x

Instr	Definition	|| Instr	Definition
ADD m	a ← a + m	LTA m	LT m, APAC
APAC	a ← a + p	LTP m	LT m, PAC
SUB m	a ← a - m	LTS m	LT m, SPAC
SPAC	a ← a - p	MPYA m	APAC, MPY m
MPY m	p ← t * m	MPYS m	SPAC, MPY m
LAC m	a ← m		
LT m	t ← m		
PAC	a ← p		
SPL m	m ← p		
SACL m	m ← a		

† m: memory operand, special registers: a,p,t

4. Introduction to Propositional Logic

In general logical expressions can be used to model qualitative information about a processor or system. Propositions and propositional logic expressions can be automatically translated into binary variables and linear constraints. Specifically logical propositions are replaced by binary variables, where true and false are equivalent to binary values of one and zero. Reasoning can be shown to be equivalent to solving a mixed integer linear programming problem. Although there are many advantages to using a ILP model instead of a production system, the biggest drawback is computational time, since solving a ILP problem is NP-complete[9]. However if the system is completely composed of horn clauses it can be solved by a LP solver and a rounding technique. Several examples will be given below to illustrate horn clauses and their translation into constraints of a ILP problem.

Horn clauses are logical expressions that have at most one conclusion. Specifically horn clauses are disjunctions with not more than one non-negated term. For example if A is true and B is true then C is true can be expressed as:

$$\text{IF } P_a \text{ AND } P_b \text{ THEN } P_c$$

which can be translated into

$$\text{NOT } (P_a \text{ AND } P_b) \text{ OR } P_c$$

$$(\text{NOT } P_a) \text{ OR } (\text{NOT } P_b) \text{ OR } P_c$$

In this last disjunction, there is only one non-negated term (the last term, P_c), so it is a horn clause. One can automatically translate any logical proposition into an ILP constraint using binary variables. For example this last disjunction can be translated by using the following binary variables: x_A, x_B, and x_C to represent the propositions P_A , P_B , P_C respectively and write the following constraint:

$$(1-x_A)+(1-x_B)+x_C \geq 1$$

5. Methodology and Modeling

This section will briefly describe the methodology for code generation. Given an application described in a high level language (such as 'C') and a DSP processor to run the application, the following methodology maps the application into a set of instructions to be executed by the DSP processor. First the application is scheduled based upon the parallelism of the targeted DSP architecture (for example scheduling operations to make use of multiple functional units would be very important for many parallel DSP architectures). Next detailed instruction mapping is performed by first generating a code generation model. Arc mappings are developed and the logical propositions detailing constraints of the targeted instruction set architecture are developed. A model is then automatically generated from these logical propositions representing the ILP model for code generation. The high level language describing the application is decomposed into a series of basic blocks which are then optimized through the code generation model for performance, code size and finally estimated power. After the code is generated (which assumes direct memory addressing), the data layout is performed using existing[13] or extended techniques[11] followed by an optimization approach to address generation (using auxiliary registers) such as [12]. In this paper for illustration purposes we will use the TMS320C2x processor as the target processor. The partial instruction set architecture is shown in table 1. The TMS320C2x architecture is a non-homogeneous register architecture that contains special registers a,p,t, and has multiplier and alu functional units. The next two sections will illustrate how the model is generated from a target processor.

5.1. Arc Mappings

To model the target DSP processor in our model we first describe the arc mapping process. Any basic block from an application can typically be represented by a data flow graph, where each operation is a node and arcs define the transfer of data output from one operation to the input of another operation. In this model each arc can be mapped into different sets of instructions, each representing the path taken by the data value in the architecture. The path for example may be composed of register to memory or memory to register types of data transfer.

The arc mapping process is to determine for each type of arc the total number of different paths that could occur and identify each one as a set, s. For example in figure 1a) there is a multiplication operation *, which sends data to an addition operation +. We identify this type of arc as a *multiply-add* arc. This means that the data produced by the multiplication operation will be input to an addition operation. Figure 1a) shows the four possible paths for transferring the result of the multiplication operation to an input of the addition operation in the TMS320C2x processor. For s=1 the output of the multiplication is stored in register p until it is used by the addition operation. No extra instructions are required for this path. In s=2 the data

is transferred from the *p* register into the *a* register right after the multiplication, using instruction *PAC*. In the remaining types it is transferred from the *p* register to memory (for example using instruction *SPLm*) and when it is required for input to the addition operation it is a memory operand (*s*=3) or loaded from memory into the *a* register (*s*=4) (for example using instruction *LACm*). Figure 1b) shows the three mappings for an *addition-subtraction* arc.

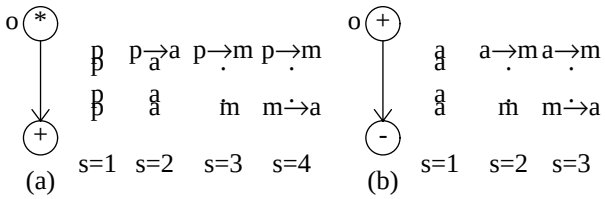

Figure 1. Example of arc mappings for *→+ in a) and +→− in b) for the TMS320C2x.

5.2 Instruction Set Architecture Constraints

Once the arc mappings are defined for the target processor, the constraints of the instruction set architecture of the DSP processor are defined using logical propositions and translated into a code generation model. The model takes the scheduled data flow graph (representing basic blocks of the application) as input and generates code as shown in figure 2. The example in figure 2 is the complex multiply ($cr = ar*br + ai*bi$, $ci = bi*ar + ai*br$). The corresponding scheduled data flow graph corresponding to a sequence of operations is shown at left hand side of figure 2. The final output from this model is shown at the right hand side. The logical propositions or the ILP constraints will also be given. Apart from basic model constraints the remaining processor specific constraints which look after conflicts in resources, register allocation, memory transfers, etc are horn clauses. Examples of some of the constraints for the code generation model will be provided below.

The basic formulation of the objective function to be minimized is a sum of costs. In particular the code size model is a direct sum of the number of instructions generated. The performance model is the actual number of cycles required to run the code and again a function of the instructions generated. The estimated energy model is a function of how many memory operands are used (where we account for the number of memory accesses from instruction memory and data memory).

$$\textit{Execution Time} = \sum_{o,s} cycles(o,s)x_{o,s} - \sum_{o} savings(o)p_o$$

$$\textit{Code Size} = \sum_{o,s} codesize(o,s)x_{o,s} - \sum_{o} csavings(o)p_o$$

$$\textit{Estimated Energy} = \sum_{o,s} memaccesses(o,s)x_{o,s} - \sum_{o} msavings(o)p_o$$

The simplest constraint is shown below which ensures that each operation must be assigned to only one set of instructions *s*.

$$\sum_{s} x_{o,s} = 1$$

The remaining constraints will be expressed using propositional logic to illustrate their representation as horn clauses.

The constraint that each input to an operation must be stored in a different resource, can be represented as : *for* $o1 \rightarrow o$ *and* $o2 \rightarrow o$, *where* $End(o1,s1)=End(o2,s1)$ *then*

IF $P_{o1,s1}$ THEN NOT $P_{o2,s2}$.

and translated into a mathematical ILP constraint using the transformations described in section 4. Similarly for the TMS320C2x, exactly one input to an adder must be in resource register *a* is formulated as follows: *for* $o1 \rightarrow o$ *and* $o2 \rightarrow o$, *where o is an addition operation,* $End(s1)=p$, $End(s2)=m$

IF $P_{o1,s1}$ THEN NOT $P_{o2,s2}$.

The general resource conflict constraints are illustrated in figure 3. In figure 3a) the operation * cannot be mapped into type s=1 (which uses register p) since operation *1 will always store it's output in register p. Similarly in figure 3b) operation + cannot be mapped into type t=1 due to conflict with resource a. This type of conflict constraint can be represented by, $o1 \rightarrow o2$, $T(o1) \leq T(o3) \leq T(o2)$, *where operation* $o2$ *uses a resource (such as the a* or *p* register) that is also being used to store the intermediate value generated by $o1$ *for* $o3$

IF $P_{o1,s1}$ AND $P_{o2,s2}$ THEN NOT $P_{o3,s3}$

Finally code compaction constraints are used to map to parallel instructions. These constraints are also horn clauses that restrict the parallel proposition P_o. For example if we have an addition operation that does not have any memory operands followed by a multiplication operation then we can use the *MPYA m* instruction to save on performance and code size (see table 1). Again this relates to two sequential operations , *where* $o1$ *is addition,* $o2$ *is multiplication,* $T(o1)=T(o2)-1 o11 \rightarrow o1, o12 \rightarrow o1$ *and* $End(o11,s1)=m$ *or* $End(o12,s2)=m$

IF ($P_{o11,s1}$ OR $P_{o12,s2}$) AND $P_{o1,s}$ AND $P_{o2,s}$

THEN NOT P_{o1}

This last constraint can easily be reformulated to support *mac* instructions in other processors, for example the M56000, where the multiplication can be chained with the addition operation.

This model also supports operations which transfer data to more than one operation. In this case a variable per operation is used to map to sets of instructions. For example input data to a multiplier can be a memory operand or loaded into the *t* register. If the same input data is used by more than one operation a second load into the *t* register (*LT m*) is not required. An example of this optimization is shown in figure 2 for input data *bi* which is stored only once in the *t* register, so the multiplication *bi*ar* does not require a *LT m* instruction. The model has also been extended to simultaneous scheduling. In this case the variable $x_{o,s}$ is replaced by $x_{o,t,s}$ where *t* refers to the time (or control step) that operation *o* is scheduled at. Precedence scheduling constraints are added and existing constraints are extended for scheduling.

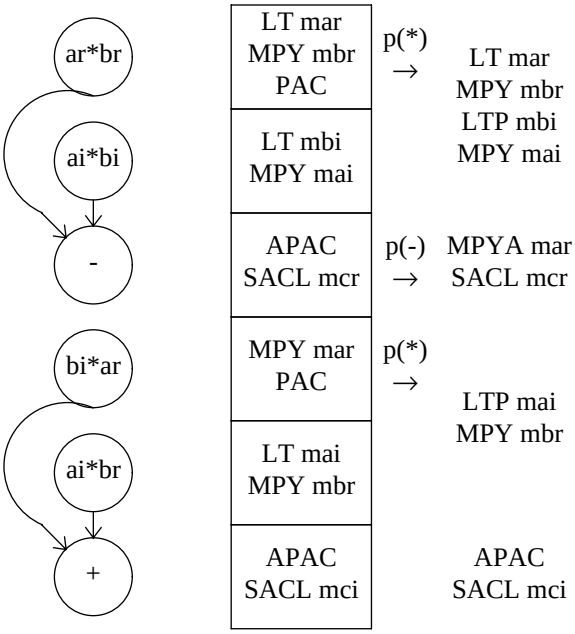

Figure 2. Example of mapping from a high level data flow graph into sets of instructions, and compacted code. These mappings are done simultaneously.

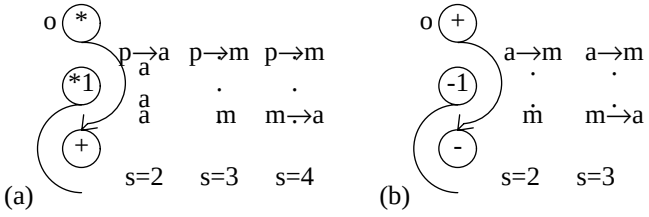

Figure 3. Example of conflict with resource *p* and *a* in a) and b) respectively, requiring the use of *s*=1 for both cases to be illegal.

6. Experimental Results

Several DSP examples are used to illustrate this methodology. Although any operation sequencer or scheduler can be used in comparison with previous research the same sequence of operations was used when we compared our results to previous research in order to make the comparison fair. All the code generated using the technique described in this paper has been verified through simulation with TI compiler generated code. The *cmul* example was taken from [11] and the *dfg* example was taken from [2]. The *lms* is a least means square algorithm and the *hp*1 and *hp*2 are high pass filter in direct form I and direct form II. The *fft* is a fast fourier transform taken from [16]. Although performance, code size or estimated energy can be optimized, the results in tables will provide code size or execution time results (or total number of cycles), since for TMS320C2x with on chip memory these are equivalent for basic blocks. Performance improvement is calculated as a [old # of cycles]/[new # of cycles] times improvement. All solutions are reported for the GAMS/Cplex LP/ILP solver[14] on a 133MHz Pentium PC.

Table 2 compares with the results of the TI C compiler generated code (set at the highest level of optimization) with the optimization model presented in this paper. To make a fair comparison all auxiliary register related instructions ie. computations/loads/etc were not included in the count. The code generated therefore assumes direct memory addressing and is only comparing actual computations related to the operations in the flow graph. For example the *cmul* example is that shown in figure 2 which uses 10 instructions (without counting indirect addressing-related instructions). The sequence of operations used for the *cmul* example is same as that used in [11] in order to compare with their code compaction technique which required 12 instructions.

Table 3 shows the size of the ILP models and the cpu times required to generate the solutions. When simultaneous scheduling was not performed, and a sequence of operations was input to the model, all solutions could be obtained from one LP solution. To illustrate the model performance the extension to simultaneous scheduling, instruction selection, register allocation and compaction was performed for the *cmul* example in only 2.3 cpu seconds, see second last row of table 3 (after searching through several nodes of the branch and bound tree). The code compactor in [11] (which does not perform instruction selection and register allocation) uses an ILP solution and runs on a RS/6000. A final run was also performed to see if the optimization model could improve upon the result of the sequencing of operations to optimize performance. In this case a cutoff value is used and the model for full scheduling, instruction selection, compaction and register allocation is solved. For the *dfg* example sequenced operations were obtained from the analysis in [2]. Using 21 instructions as a cutoff the ILP with simultaneous

scheduling was solved and after 1042 cpu seconds it determined that the problem was integer infeasible. This proved that the code generated from the ILP model in this paper was in fact optimal.

Since the TI 'C' compiler is an industrial compiler a complete comparison of final code generated was performed with our code generation model presented in this paper in combination with the address optimization technique in [12]. These results are shown in table 4 (where *O* refers to the number of instructions used to generate computations of basic block alone and *A* refers to instructions used for address computations alone). Minimization of the number of memory accesses, to estimate energy optimization, was also performed. In the *fft* example an optimal solution was obtained in 1 cpu second (which was in fact the same as the optimal performance solution, shown in table 2).

Table 2. Performance Comparison

Example	# of Cycles/Code Size Compiler/Prev Res	Optim	Times Impr
cmul	15/12	10	1.50/1.20
dfg	24	21	1.14
hp1	48	22	2.18
hp2	48	30	1.60
lms	43	26	1.65
fft	108	99	1.09

Table 3. CPU times and Model sizes

Example	ILP Size Var	Eqn	CPU sec
lms	112	149	0.3
hp2	124	171	0.1
hp1	123	190	0.1
dfg	65	163	0.3
cmul	43	59	0.1
fft	274	1504	3.4
cmul in [11]	141	44	0.7
cmul†	137	381	2.3
dfg†	332	1465	105

† Simultaneous scheduling

Table 4. Code Generation And Address Optimization

Ex	TI 'C' Compiler O	A	Cyc	Optimization O	A	Cyc	Times Impr
hp1	48	39	87	22	6	28	3.10
lms	43	36	79	26	6	32	2.46
fft	108	87	195	99	7	106	1.83

7. Discussions and Conclusions

Although the model is solved as an ILP, the majority of examples in this paper could be solved using only one LP solution. This is not surprising since related research has shown that the use of logical inferences and horn clauses in ILP's has this characteristic[7]. Although the strict use of only horn clauses may require large number of constraints we found that the number of constraints generated in this model were manageable. An ILP approach was used instead of a production system since it is independent of the order of rules (therefore does not restrict formulation or ordering of rules), and it is easier to model complex constraints, including non-horn clauses. The optimality of the code generated by this modeling approach is only as good as the model itself. In other words if the constraints and arc mappings defined in the model are not complete, then the code generated may not be optimal.

Any scheduler such as one to minimize maximum density of variable lifetimes or the scheduler in [4] can be used in conjunction with this instruction selection, compaction and register allocation model. Other DSP processors can be modeled, such as the Motorola M56K and TMS320C3x. In combination with our addressing technique the savings in performance and codes size is even more significant, see table 4. For example in the *cmul* case the acutal final code using indirect addressing provides improvement over what was reported as the theoretical lower bound in [11].

In summary code size and performance improvements of 1.09 to 2.18 times (see table 2) were attained over previous research and commercial compilers. The technique presented in this paper performs simultaneous instruction selection, compaction and register allocation in very fast cpu times unlike other researched models[5]. Simultaneous scheduling and instruction selection, compaction, and register allocation can also be done with this model, however cpu times do increase (see table 3). In contrast to previous research which examined code compaction without instruction selection and register allocation [8], or code generation without compaction [1] the model presented in this paper can perform these tasks simultaneously. This research is important for industry since code generation for DSP processors or cores that minimizes code size, energy dissipation and maximizes performance, is critical to ensuring that the final product will be reliable, cost effective, and competitive. This research is supported in part by grants from NSERC and ITRC.

References

[1] G.Araujo, S.Malik, "Optimal Code Generation for Embdded memory Non-homogeneous register architectures", 8th International Symp on System Synthesis, 1995, p36-41.

[2] G.Araujo, S.Malik, M.T-C.Lee "Using Register-Transfer Paths in Code Generation of Heterogeneous Memory-Register Architectures", Design Automation Conference, 1996.

[3] A. Timmer,M.Strik, J.vanMeerbergen, J.Jess "Conflict Modeling and Instruction Scheduling in Code Generaton for In-house DSP Cores", Design Automation Conference, 1995.

[4] S.Liao, S.Devedas, K.Keutzer, S.Tjiang, A.Wang "Code Optimization Techniques for Embedded DSP Microprocessors" Design Automation Conference 1995.

[5] T.Wilson, G.Grewal, B.Halley, D.Banerji "An Integrated Approach to Retargetable Code Generation " 7th Int'l Symp High Level Synthesis, 1994 p70-75.

[6] R.Raman, I.Grossman,"Relation Between MILP Modeling

and Logical Inference for Chemical Process Synthesis" Dept of Chemical Eng, Technical Report, CMU, Jan 1990.

[7] Hooker, J.N. "Resolution vs. Cutting Plane Solution of Inference Problems: Some Computational Experience", Operations Research Letters, vol7,No1(1) 1988.

[8] P.Marwedel, G.Goossens Eds. **Code Generation for Embedded Processors**, Kluwer Academic Pub, 1995.

[9] Garey, Johnson **Computers and Intractability**, Freeman, 1979.

[10] E.Lee, "Programmable DSP Architectures: Part I and II", IEEE ASSP Magazine, Oct and Nov 1988.

[11] R.Leupers, P.Marwedel, "Time-Constrained Code compaction for DSP's", Trans on VLSI Systems, Vol.5,no.1, March 1997.

[12] C.Gebotys, "DSP Address Optimization Using A Minimum Cost Circulation Technique", to appear Int'l Conf on Computer-Aided Design, Nov 1997.

[13] S.Liao,S.Devadas,K.Keutzer,S.Tjiang,A.Wang, "Storage Assignment to Decrease Code Size", ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1995.

[14] A.Brooke, D.Kendrick, A.Meeraus, **GAMS A User's Guide**, Scientific Press, 1992.

[15] **TMS320C2x User's Guide,** Texas Instrument Inc, 1996.

[16] P.E.Papamichalis, **Digital Signal Processing Applications with the TMS320 Family,** Vol.3, Prentice Hall, 1990.