Generic Code Generator

Riad S. Jabri*

ABSTRACT

This paper introduces a generic code generator (GCG) based on a formal and a global optimizing approach. According to the proposed approach, GCG is specified in terms of a generic description of different machine architectures and at different levels. Each level specifies some of the machine aspects, represented by a respective generic machine language and an appropriate optimization. The GCG is implemented using a generic optimizing translation scheme that performs successive transformation of the machine languages of the predecessor levels into the languages of their respective successor levels, accompanied with an imposed optimization effect. Based on such approach, GCG has been instantiated (implemented) for three machines: a register virtual machine; the IA-32 and the UltraSPARC. Such instantiations have proved the distinguishing features of the proposed GCG from the similar ones; namely, ease of retargetability, effective optimization, correctness, systematic and formalized machine description. The experimental implementation of GCG revealed the fact that using generic templates to describe the machine instructions leads to a relatively less distinct description. Also, combining the local and global optimization leads to a relatively high instructions count covered by a limited number of registers.

Keywords: Code generator, Optimization, Machine description, Retagetability, Pattern matching.

1. INTRODUCTION

Many retargetable code generators have been developed in the past decade (Aho and Johnson, 1976; Glanville and Graham, 1978). However, retargetability remains important with emphases on important aspects, such as the degree of automation, the rapidness of retargetability (Oh and Paek, 2003), the effect of program transformations (Lattner and Adve, 2004), the correctness of the generated code (Leory, 2006) and the appropriateness of the machine dependent optimization. Consequently, the most recent efforts have been concentrated on the following:

a) The use of architecture description languages and their continuous improvements. Some of the existing languages are: TWIG (Aho et al., 1989) and LBURG (Fraser and Hanson, 1995). They have been successfully used to automatically generate both a code generator and a machine specific optimizer. However, efforts for improving their capabilities have been continued. Examples of such efforts are SPIM (Deshpande and Khedker, 2007), READ (Oh and Paek, 2003), ADL (Farfeledej et al., 2006) and OPTIMIST (Eriksson et al., 2008). SPIM has an objective to allow an incremental and a systematic machine description. The objective of READ is a rapid and an easy retargeting as well as a machine dependent optimization. ADL is based on using XML language for the architecture description and a respective optimizing code generator. OPTIMIST aims at supporting integrated code generation, using heuristics based on genetic algorithms.

b) The use of LR grammars, regular tree grammars and rewrite systems to represent machines instructions. Examples are the efforts given in (Shankar et al., 2000; Ferdinand et al., 1994; Cleophas et al., 2006), where the instructions are represented by patterns drawn from such grammars. The patterns are augmented by the costs of their respective machine cycles as minimization criteria. Pattern matching is then performed using parsing automata and /or code selectors to generate an optimal set of machine instructions, respective to the intermediate

* Computer Science Department, King Abdullah II School for Information Technology, University of Jordan, Amman, Jordan. Received on 2/2/2009 and Accepted for Publication on 31/5/2009.
representation (IR) of the front-end compilers (Borchardt, 2005).

c) The use of compilation frameworks, based on intermediate languages such as static single assignment (SSA) (Cytron, et al., 1991; Bilardi, 2003) and rewrite terms (Bravenboer and Visser, 2002) to support optimization, compilation correctness and safety. Representative recent efforts are LLVM (Lattner and Adve, 2004) and MetaPrl (Hickey, et al., 2003). LLVM is a low level code representation and a respective compiler that supports program transformation and analysis. MetaPrl is based on lambda typed terms to support correctness certification within a logical framework.

This paper is a continuation of such recent efforts and introduces a generic code generator (GCC) that can be instantiated for different virtual machines and actual ones. GCC is formally specified and implemented using an approach which emphasizes the factors which have impact on the degree of retargetability, effective optimization and correctness. Thus based on such approach, GCC is generically specified in terms of a target machine composed of three levels. Each level is specified by the generic triple Target-Machine (levli) = (machine language (MLi), metadata model (MDi), optimization scheme (OPSi)). The implementation of the generic code generator is then defined by successive applications of a generic translation, instantiation and optimization scheme (GTIOPSi) to perform the following:

1) Translate the generic machine language of level i (MLi) into the one of level i+1 (MLi+1) and define a generic optimization requirements for the level i+2.
2) Instantiate the machine language of level i+1 (MLi+1) and its respective optimization scheme for a specific machine to generate the specific MLi+1 with an imposed effect of the specific optimization.

GCC has been specified and implemented for a register virtual machine as well as for two real ones: the Intel IA-32 (Intel Corporation, 2004) and the UltraSparck (Sun Microsystems, 2007).

The remainder of this paper is organized as follows: Section 2 outlines the proposed approach. Section 3 specifies the GCC input as the target machine (level 0). The specification and implementation of the target machine (level 1) and the target machine (level 2) are given in sections 4 and 5, respectively. Experimental results including the instantiation of the proposed GCC by a virtual machine, IA-32 and UltraSparc are given in section 6. An evaluation of GCC and a conclusion are given in sections 7 and 8, respectively.

2. PROPOSED APPROACH

To emphasize the factors, which have impact on the degree of retargetability, effective optimization and correctness, we formulate the following requirements:

- The need to exploit the different machine architectures at their respective different levels in a systematic and a generic way to support the respective code generation and optimization.
- The need for low level intermediate representation to serve as a generic machine language, but preserves the structure of the high level language. Thus, permits efficient and global optimization as well as easy and correct adaptation to different levels of different machines.
- The need to formalize the code generator in a way that supports its correctness, genericity, integration and rapid retargetability.

To satisfy such requirements the proposed approach proceeds as follows:

1) The generic code generator is specified in terms of a hierarchy of target machine levels. Each level is characterized by the generic triple Target-Machine (levli) = (machine language (MLi), metadata model (MDi), optimization scheme (OPSi)). The implementation of the generic code generator is then defined by successive applications of a generic translation, instantiation and optimization scheme (GTIOPSi) to perform the following:

a) Translate the machine language (MLi-1) for level i-1 into a generic one for level i (MLi) with an imposed effect of a generic optimization (OPi) and to define the generic optimization profile for the optimization needed at level i+1 (OPRi+1).

b) Instantiate the machine language (MLi) and the respective level dependent optimization to generate the specific MLi with imposed effect of the specific optimization needed by level i.

Within the framework of this research, GCC is
specified in terms of a target machine composed of three levels and in terms of their respective translation, insanitation and optimization schemes. The considered levels are as follows:

1) **Target-Machine (Level 0) = (ML0, MD0, OPS0)** is defined as the front-end processor and as such constitutes the input for the GCG, where ML0 and MD0 are totally machine independent, preserve the structure of the high level language (HLL) and represent the architecture of the machine, as defined by the implementation of HLL. The optimization scheme OPS0=(OP0, OPR0) is defined as language dependent optimization(OP1), performed on the flow graph representation of ML0(OPR0).

2) **Target-Machine (Level 1) = (ML1, MD1, OPS1)**. Where: (ML1, MD1) represents the architecture of the machine as defined by the language implementation. However, ML1 is with an imposed effect of a global machine independent optimization (OP1 ∈ OPS1) and annotated with the respective global and generic machine optimization profile (OPR2 ∈ OPS1).

3) **Target-Machine (Level 2) = (ML2, MD2, OPS2)**. Where: (LIR2, MD2) represents minimum machine architecture in terms of its instruction set and addressing modes with a respective machine dependent optimization scheme (OPS2). This level is a machine dependent. However, specified in a generic way.

Further levels can be defined such as Target-Machine (Level 3) = (ML3, MD3, OPS3) that represents the machine resources in terms of execution units, pipeline, instruction execution cycle (ML3, MD3) with a respective machine dependent optimization, resource allocation and scheduling scheme (OPS3). However, within the framework of this paper the Target-Machine (Level 2) is defined as the GCG output. Thus, assuming a flow graph representation of three address code, produced by a font end processor (FEP), proposed by Jabri (1990), to instantiate ML0. An illustration example for ML0 is given in Figure 1.

### 3. SPECIFICATION OF THE GCG INPUT

The input of GCG is defined by Target-Machine (Level 0) = (ML0, MD0, OPS0). Hence, its specification is reduced to the specification of ML0, MD0 and OPS0 respectively and as follows:

ML0 is defined as a flow graph representation of a three address code, specified by a context free grammar (CFG) and having the forms:

a) \(X_{ij}: g X_{i+1j} X_{i+2j}\), where:
   i) \(ij\) is an index represents the occurrence \(i\) of the grammar symbol \(X\) in the basic block \(j\).
   ii) \(g \in \Sigma\) is an operator \((=, *, +, \ldots, -)\), \(X_{ij} \in \mathbb{N}\) and \((X_{i+1j}, X_{i+2j}) \in (\Sigma \cup \mathbb{N})\) are operands that are classified into the following types: value, address, expression and definition operands.

b) \(X_{ij}: g X_{ij}\), where \(X_{ij}\) is a label operand and \(g\) is a conditional and unconditional jump operator.

Such definition features SAA form. However, we adopt an intermediate representation generated by a front end processor (FEP), proposed by Jabri (1990), to instantiate ML0. An illustration example for ML0 is given in Figure 1.

The metadata model and the optimization respective to the target machine (Level0) are specified as follows:

1) The adopted representation of ML0 preserves the structure of source language. Hence, the types of its individual constructs represent the meta data model MD0 of target machine (Level 0).

2) The flow graph representation is considered as OPR0. The data flow analysis within the framework of the front end processor (Jabri, 1990) is considered as OP0. The data flow information computed by OP0 represents the respective optimization requirements for the target machine at level 1(OPR1). Such information is aggregated into two tables TABDES1 and TABDES2, respectively and having the following forms: TABDES1= \{Xi, Type, Occ-Set\} and TABDES2 = \{Key, ID-Occ-Set, Use-in-Set, Pi\}, respectively, where:
   a) Each entry in TABDES1 has a respective entry in TABDES2.
   b) \(Xi\) represents the individual terminal or non terminals in ML0.
   c) Type represents the type of \(Xi\), defined as value, address, expression or definition operand, with the following respective encoding: VAL-OPND, ADDR-OPND, EXP-OPND and DEF-OPND.
Figure 1: A Program in the ML0 Language.

4. SPECIFICATION AND IMPLEMENTATION OF THE TARGET MACHINE (LEVEL 1)

The machine independent code optimization within the framework of our code generator is defined as an extension to the one that is performed by the front-end processor (FEP), proposed by Jabri (1990). Such extension is motivated by two main objectives. The first objective is to perform global optimization for the three-address code, produced by the FEP. The second one is to perform global machine independent memory allocation and to determine the global storage requirements, needed by the subsequent machine dependent optimization and its respective machine dependent memory allocation. Hence, the Target-Machine (Level 1) is specified as (ML1, MD1, OPS1), where:

1) OPS1 = (OP1, OPR1)
   a) OP1 is defined as a global optimization scheme (GOPT) to perform machine independent optimization and memory allocation, as defined by the language implementation.
   b) OPR1 is instantiated by TABDES1 and TABDES2.

2) ML1 is defined as ML0, but with imposed memory allocation and optimization effect of the GOPT. MD1 is defined the same as MD0.

The generic translation, instantiation and optimization scheme (GTIOPS1) is then defined as:

GTIOPS1: ML0 → ML1 x TABDES1 x TABDES2 x GOPT → ML1 x OPR2, where:

- ML0 is a flow graph representation of a three-address code, produced by FEP and annotated by the data flow information, as previously described.
- OPR2 constitutes generic global storage requirements specification (GSRQ2) for machine dependent optimization.

Since ML1 is defined the same as ML0, GTIOPS1 is reduced to the application of GOPT on (ML0 x TABDES1 x TABDES2) to produce ML1 x GSRQ2. Subsequently, the implementation of Target-Machine (Level 1) is reduced to the application of GTIOPS1 as given in the following sub-section.

Global Machine Independent Optimization Scheme (GOPT)

Let the ML0 is the intermediate language produced by FEP and over which a data flow analysis has been performed, as previously described, where the operands of the three address code are classified into the following types: VAL-OPND, ADDR-OPND, EXP-OPND and DEF-OPND.

Let LSAS is a language dependent memory allocation scheme, defined by its implementation, according to which the storage (stack location, heap location) for the different high level language constructs is allocated and subsequently the operands of intermediate representation are designated. Such language dependent storage is defined by the triple LDS= (construct name, storage type, storage designator). The LSAS scheme and LDS are taken into consideration during code generation and machine dependent optimization. However, within the framework of machine independent optimization, it is sufficient to consider the LDS storage as a set of constructs names, decomposed into three segments. The first segment contains the global storage locations at the program level, across the boundaries of the flow graph basic blocks. The second segment contains the global storage locations within the basic blocks. The third segment contains the local storage within the basic blocks. Thus, each storage location, and subsequently each three address operand, is generically specified by the triple (SDR, SC, SS). Where: SDR is a machine independent storage designator, represented by the operand name and a global designator, if it has been allocated. SC is a storage class to specify the segment to which the storage location belongs. SS is a storage status to indicate whether a respective global storage has been allocated.

The implementation of GOPT and subsequently the implementation of GTIOPS1 are achieved by Algorithm-GTIOPS1, given below. This algorithm determines the storage class for each one of the ML0 operands; imposes the effect of global optimization on ML0; and computes the respective GSRQ2, defined by the set GORQ2 = {GORQ2- Category1, GORQ2- Category2, GORQ2- Category3} that represents the global allocation and the optimization profile, classified into three categories. Such categories specify the storage allocated to ML1 operands, the global storage requirements within each basic block and the global storage requirements across the basic blocks respectively. They are defined as follows:

- GORQ2- Category1 = \( \bigcup_{ij} \) GORQ2- Category1 (Xij) = \{SC [Xij], SDR [Xij], SS [Xij]\}.

- GORQ2-Category2 = \( \bigcup_i \) GORQ2-Category2
  (BBNoi) = \{GS-BB, \{ONAME, OTYPE, OCC\}. This category consists of the total number of the global storage per basic block and the set of the global operands, specified in terms of their names, types and occurrence frequency.

- GORQ2-Category3 = \{TNGS- PROG-LEVEL, \{ONAME, OTYPE, OCC, BBNoi\}\}. This category consists of the total number of the global storage at program level and a respective set of the global operand specifications.

Algorithm- GTIOPS1

Input: LM0, TABDES1 and TABDES2

Output: LM1 and global storage allocation and optimization requirement (GORQ2)

Method:

\{For each entry Xi in TABDES1
\{For each entry TABDES2 (k) respective to TABDES1(Xi)
\{Perform the steps 1, 2 and 3\}
These steps and their respective code segments are as follows:

Step1: Select the next subsequent operand Xij of type TABDES1(Xi). Type:
Xij= Select (TABDES2(K). Occ-Set )

Step2. Partition the Xij - data flow information according to the basic blocks in which they occur:
Xij-flowinf = \( \bigcup_{i=1}^p \) Xij-Bflowinf i | Xij-Bflowinf i =\( \bigcup_{j=1}^{p-1} \) Xij, \( \forall \) Xij \in TABDES2(k).Xij-flowinf

Step3. Determine the storage class, compute GORQ , respective to Xij and to the operands in Xij- flowinf. Then, perform the respective optimization as follows:
3.1- Determine whether the $X_{ij}$ storage class is a
global across basic blocks boundaries.

if $|X_{ij}-\text{Bflowinf}| > 1$

a- define the $X_{ij}$ storage class as global one: $SC[X_{ij}] = 3$, $SDES[X_{ij}] = \{X_{ij},G X_{ij}\}$ $SSx_{ij} = 0$

For each $Y_{ji} \in X_{ij}-\text{flowinf}$ \{ GORQ2- Category1 =

GORQ2- Category1 \cup GORQ2-Category1(Y_{ij}) = \{SC[Y_{ij}] = 1, SDR[Y_{ij}] = \{Y_{ij},GX_{ij}\},SSx_{ij} = 0\} \}
b- Compute the GORQ2, respective to $X_{ij}$ and to $X_{ij}-\text{flowinf}$:

GORQ2- Category3 = GORQ2- Category3 \cup \{TNGS-BB +1 \}, \{ ONAME = X_{ij}, TYPE = type, OCC = \min X_{ij}-\text{Bflowinf} \} \}

c- Impose the effect of the global optimization:

- Copy propagation
If TABDES1(Xi).Type = DEF- OPND \{ Replace $Y_{ij}$ by $X_{ij}$ in ML0 (Use-in-Set$y_{ij}$) for all $Y_{ij} \in X_{ij}-\text{flowinf}$ \}

- Equivalent expression elimination
If TABDES1(Xi).Type = EXP- OPND \{ Equivalent $\text{exp} = X_{ij}-\text{flowinf}\setminus X_{ij}$ Delete ML0 ($Y_{ij}$) for all $Y_{ij} \in \text{Equivalent } \text{exp} \} \}

3.2. Determine the storage class for the global operands within basic blocks, compute their respective

GORQ2 and impose the effect of the respective optimization as in the step 3.1(a), but using $X_{ij}-\text{Bflowinf}$ instead of $X_{ij}-\text{flowinf}$:

else if $|X_{ij}-\text{Bflowinf}| = 1$ and $\exists X_{ij}-\text{Bflowinf} \in X_{ij}-\text{flowinf}$

$\{SC[X_{ij}] = 2, SDR[X_{ij}] = \{X_{ij},GB\} SSx_{ij} = 0\}$

For each $Y_{ij} \in X_{ij}-\text{Bflowinf}(\text{GORQ2- Category1 }\cup \text{GORQ2- Category1}(Y_{ij}) = \{SC[Y_{ij}] = 2, SDR[Y_{ij}] = \{Y_{ij},GX_{ij}\},SSx_{ij} = 0\})$

GORQ2- Category2 = GORQ2- Category2 \cup \{TNGS-BB +1, ONAME = X_{ij}, OTYPE = type, OCC = \min X_{ij}-\text{Bflowinf} \}

End- Algorithm- GTIOPS1.

Example: Let a program $p$ in ML0 is given and as shown in Figure 1, the application of GTIOPS1 will
produce an optimized version of $p$ in ML1 as shown in Figure 2.

\begin{table}[h]
\begin{tabular}{|l|}
\hline
BB 1: Def$_{11}$ \rightarrow := Adr a 12 Con 1 13 & BB 2: Def$_{17}$ \rightarrow := Adr a 12 Const 30 23 \\
Def$_{14}$ \rightarrow := Adr b 15 Con 2 16 & \hline
Def$_{17}$ \rightarrow := Adr d 18 Con 5 19 & BB 4: Exp$_{41}$ \rightarrow * Def$_{14}$ Value d 43 \\
\hline
BB 3: Exp$_{31}$ \rightarrow + Def$_{11}$ Def$_{14}$ & Def$_{44}$ \rightarrow := Adr d 45 Exp$_{41}$ \\
Def$_{34}$ \rightarrow := Adr c 35 Exp$_{31}$ & BB 6: Def$_{64}$ \rightarrow := Adr d 65 Exp$_{31}$ \\
Exp$_{35}$ \rightarrow - Exp$_{31}$ Def$_{11}$ & Exp$_{66}$ \rightarrow + Value e 67 Con 15 68 \\
Def$_{38}$ \rightarrow := Adr d 39 Exp$_{35}$ & Def$_{69}$ \rightarrow := Adr e 610 Exp$_{66}$ \\
Exp$_{310}$ \rightarrow > Exp$_{35}$ Con 20 312 & BB 8: Def$_{44}$ \rightarrow := Adr b 85 Exp$_{31}$ \\
BB 5: Exp$_{35}$ \rightarrow > Def$_{17}$ Con 25 53 & Def$_{89}$ \rightarrow := Adr c 810 Exp$_{35}$ \\
BB 9: Exp$_{91}$ \rightarrow * Def$_{14}$ Def$_{17}$ & BB 6: Def$_{64}$ \rightarrow := Adr d 65 Exp$_{31}$ \\
Def$_{94}$ \rightarrow := Adr a 95 Exp$_{31}$ & Exp$_{66}$ \rightarrow + Value e 67 Con 15 68 \\
Exp$_{96}$ \rightarrow - Value a 97 Def$_{17}$ & Def$_{69}$ \rightarrow := Adr e 610 Exp$_{66}$ \\
Def$_{99}$ \rightarrow := Adr b 910 Exp$_{96}$ & BB 8: Def$_{44}$ \rightarrow := Adr b 85 Exp$_{31}$ \\
\hline
\end{tabular}
\end{table}

Figure 2: An Optimized ML1 Program.

5. SPECIFICATION OF THE TARGET MACHINE
(LEVEL 2) AND IMPLEMENTATION OF ML2

Level 2 of the target machine is specified using an approach that meets the following objectives:

- To represent the architecture of different machines, using their respective instructions set, addressing modes and machine dependent optimization.
- Inherit the context dependent conditions (data
types and scopes) and the data flow information of ML1, and subsequently of the source language.

- Permit global optimization.
- Permit efficient and generic mapping of the operations and operands of ML1 into a respective operations and operands in the target machine, constituting ML2.

To meet these objectives, Target Machine (level2) is specified in terms of the triple (ML2, MD2, OPS2), where:

1) MD2 = (STMD, STALLOCMD, TSAS).
   - STMD = (memory locations (stack frames, heap), set of general (and special) purpose registers) is a generic meta data model, representing the target machine storage.
   - STALLOCMD = (construct-type, addressing mode, storage designator) is a generic meta data model, used by the target machine storage allocation scheme (TSAS) to map the storage defined by the language implementation (LDS) and subsequently the different language constructs on the target machine storage. Such model represents the respective addressing mode and the storage designator assigned to each construct type, in a way that permits the most efficient access to such construct. Hence, the target memory allocation scheme is defined by the function: TSAS: STALLOCMD x LDS \rightarrow MDS = (construct name, storage designator) to produce the machine dependent storage (MDS), respective to the one defined by the language implementation LDS).

2) ML2 is specified by a generic rewriting scheme, as described in the following sub-section and assuming that the machine instruction set is specified using a generic format. Such format defines the instructions as quadruples(operations that involve zero or more operands) having the form: θ { } Topnd1 { } Topnd2 { } Topnd3 { }, where:
   - θ represents generic operation designator
   - Topnd1, Topnd2 and Topnd3 represent generic operands designators.
   - { } represent implicit semantic actions to replace θ ,Topnd1, Topnd2 and Top by their respective machine dependent ones based on: the operand types; the instructions in which they are used and the addressing modes, applicable to such instructions. The implicit semantic actions, associated with instructions operands, are assigned by OPS2 to impose the effect of the global optimization, as described below.

3) OPS2 is reduced to generic machine dependent global optimization, defined for this level by the function MDCOS: OPNDTYPE x SC x SS \rightarrow semantic action. MDCOS assigns a unique semantic action to each one of the ML2 instructions operands to determine its machine dependent designator, based on their respective types (OPNDTYPE), storage classes (SC), as propagated from ML1, and on the operands storage status (SS), as determined during code generation. The semantic action assigned by MDCOS to the ML2 operands determines their designators as one of the followings:
   - A machine dependent designator as determined by TSAS, if (SC=1 and SS=0).
   - A new global storage designator, if (SC=3, SC=2 and SS=0). This reflects the initial use (creation) of a global operand. Hence, the semantic action sets SS to one.
   - An existing global storage designator, if (SC=3 , SC=2 and SS=1). This reflects the subsequent use of a global operand and subsequently the global storage.

Specification of Intermediate Language ML2

The individual constructs of ML2 are specified by a generic rewrite scheme as follows:

1) ML1 is defined by context free grammar G=(Σ,N,P, S), considering all ML1 operands as non-terminals with the following productions:
   - For each ML1 construct, a production of the following form is defined
     p: AML1 \rightarrow ML1 α ∈ P such that: AML1 ∈ N ,( ML1 α = X1ML1…. Xn ML1 or ML1 α = g X1 ML1 X2 ML1 X3 ML1). Where: (X1 ML1…. XnML1) ∈ N and g ∈ Σ.
   - For each operand (X ML1 ) occurrence in ML1 α , a production p: XML1 \rightarrow T XML1 is defined. Where: T XML1 represents the terminal operand respective to XML1.

2) Matching criteria is formulated for each operand type, representing its different context dependent conditions (CDC), types (OPTYPE) , storage class (SC) and storage status (SS). Such matching criteria is defined for the individual nonterminals as the relation MC (XML1 ) = (CDC(XML1), OPTYPE ,SC, SS) and for each production p∈ P: AML1 \rightarrow αML1 as the relation: MC (AML1 ) = \bigcup (CDC(XML1), OPTYPE ,SC, SS) | Xi ML1 ∈ αML1. Where OPTYPE and CDC encode the operand type and its context such as: OPNDTYPE (def, exp, val ),scope, type,…, l-value, r-value.

3) A context free grammar GML2 is constructed to generate ML2 using the rewrite scheme: RRS (ML1
construct) → ML2 construct). Where: (ML1 construct) and (ML2 construct) are defined as: MC (AML1), AML1 → αML1) and (MC (AML2), AML2 → αML2), respectively.

4) For each construct (MC (AML1), AML1 → ML1α) ∈ ML1 a respective construct (MC (AML2), AML2 → ML2α) ∈ ML2 is defined, such that:
   i) AML2 and its respective MC (AML2) are defined the same as AML1 and its respective MC (AML1).

   Thus, the context dependent conditions and operand types of the individual ML1 constructs are propagated to ML2. Hence, the structure of ML1 is preserved by ML2. Further more, the propagated context dependent conditions and operand types are instantiated by the respective machine dependent ones during code generation.

   ii) αML2 = X1ML2...XiML2 if αML2 = X1ML2...XiML2
       X1ML2 X2ML2 g { } TX1 TX2 TA ML2
       if αML2 = g X1ML2 X2ML2
       TXML2 if αML2 = TXML2

   αML2 defines the meaning of αML1 in terms of the instruction set of the target machine. However, such definition constitutes partial and incremental translation of αML1 according to matching criteria. Hence, the interpretation of the different alternatives of αML2 is as follows:

   • The non terminal operands have a respective derivation to define their meaning. If such derivation is of the form TX then RSS defines the operand designator either as specified by the target machine or as specified by the global machine dependent optimization.

   • If αML1 is of the form g X1 X2 then the respective αML2 is of the form X1 X2 g TX1 TX2 TA ML2 that represents the translation of X1 and X2, according to their matching criteria, including their respective machine designators, according to the global machine dependent optimization (OPS2) and the target machine instruction in which they are used.

   • Within the instruction context, semantic actions are defined to replace the operands occurrences (TX1, TX2 and T AML2) by their respective designators, according to OPS2.

   • The generic format (g TX1 TX2 TA ML2) is considered as generic template, specified by a macro definition of target machine instructions or by a subject tree to be matched with the RTL patterns, defining the target machine instructions.

Implementation of the Language ML2

Based on the specification of Target Machine (level 2), given in the previous sub-section, the global translation instantiation and optimization scheme GTIOPS2 is instantiated as:

GTIOPS2: (RRS (ML1) → ML2) x (STALLOCMD x TSAS x STMD) x (GORQ2 x MDCOS) → ML2 x OPR2 to perform the following:

a) Parse the individual ML1 constructs and concurrently generate their corresponding ML2 constructs.

b) Parse the generated ML2 constructs to determine the machine dependent operand designators and to impose the effect of the machine dependent global optimization MDCOS.

Thus, the implementation of ML2 is reduced to the implementation of GTIOP2, according to the following algorithm

Algorithm- GTIOPS2

Input: FG(ML1), GORQ2 Tables, RRS Table, STALLOCMD Table, TSAS scheme and MDCOS Table
Output: FG (ML2) with imposed effect of MDCOS
Method:
For each basic block BBML1 in FG(ML1)
{Step 0: Create respective BBML2 in FG(ML2)
Step 1: (AML1 → αML1) = Production (Fetch next code (BBML1))
Step 2: MC(AML1) = Matching criteria (AML1)
Step 3: (MC (AML2), AML2 → αML2) = RRS (MC (AML1), AML1 → αML1)
Step 4: Add next code ((BBML2), IOS (MC (AML2), AML2 → αML2))}

where:
• FG(ML1) and FG(ML2) are the flow graph representation of ML1 and ML2, respectively.
• GORQ2 Tables (GORQ2T1, GORQ2T2, GORQ2T3) represent the three categories of GORQ2. They are accessed by the entries GORQ2T1[Xij], GORQ2T2[BBno] and GORQ2T3[i], respectively.
• RRS Table (RSS) represent the generic rewrite rules, accessed by the entry RRS (MC (AML1), AML1 → αML1), where the respective (MC (AML2), AML2 → αML2) ∈ ML2 is stored.
• MDCOS Table represent the global optimization scheme, accessed by the entry MDCOS (OPNDTYPE (Xij), SC(Xij), SS(Xij)), where the respective semantic action is stored.
STALLOCMD Table represent the machine dependent storage allocation scheme, respective to each operand type. TSAS is the machine storage allocation scheme, used by MDCOS to determine the machine dependent storage designator respective to a given operand.

Algorithm GTIOPS2 processes the subsequent ML1 in each basic block as follows:

a) It determines the production in ML1 and the matching criteria respective to the considered construct (Step 1 and Step 2) by the functions Production and Matching criteria, respectively, as previously described and using GORQ2 Tables.

b) It determines the production in ML2 respective to the considered one in ML1, as defined by the rewriting scheme RSS, using RSS Table (Step 3).

c) Applies the function IOS on the determined ML2 production (Step 4) to perform instantiation and optimization according as given in Figure 3, where:

```
IOS (MC (A_{LR2} , A_{LR2} \rightarrow \alpha_{LR3} )
{ MDCOS (opndype(AML2 ), GORQ2T1.SC (AML2 ), GORQ2T1.SS (AML2)
p \alpha_{LR3} = Parse ( \alpha_{LR3} )
Enter (A_{LR2}, translated)
ns = next symbol (p \alpha_{LR3} )
While (there is un processed symbol in p \alpha_{LR3} )
{ ns = next symbol (p \alpha_{LR3} )
if ns is semantic action { perform the indicated action}
if ns is terminal { output= output \cup MDCOS (opndype(ns ), GORQ2T1.SC (ns ), GORQ2T1.SS (ns))}
if ns is non terminal { if Not-in (ns, translated) { IOS (MC (ns), ns \rightarrow \alpha_{LR3} )} }
else{ output= output \cup MDCOS (opndype(ns ), GORQ2T1.SC (ns ), GORQ2T1.SS (ns))}
}<output>( \alpha_{LR3} )
```

Figure 3: The Function IOS.

The function Parse decomposes \(\alpha_{ML2}\) to its constituents symbols (terminal, nonterminal, ..., semantic action). The function Enter marks the operand as translated, upon its first occurrence. MDCOS performs the semantic action defined for the considered operand based on its respective type (OPNDTYPE), SC and SS to determine its storage designator, as described in the previous sub-section using the MDCOS function, as well as using the storage allocation schemes TSAS, AGDS and ALDS to determine (allocate) storage designators for local operands, global operands and for local computations. MDCOS function is implemented in terms of semantic actions as given in Table 1, where:

Semantic action 1: Determine the machine dependent storage designator, as specified by TSAS.

Replace the designator of code (considered) operand by the machine dependent one. Mark the operand designator as determined (SS=1). The following code segment represents these actions.

```
{GORQ2T1.SDR (operand) = TSAS (STALLOCMD (Type (operand), operand)) GORQ2T1.SS (operand) =1; code (operand) =code (GORQ2T1.SDR (operand))
```

Semantic action 2: Replace the code operand designator by the machine dependent one and return such designator {\{GORQ2T1.SDR (operand)\}}.

Semantic action 3: Allocate and add a global storage designator to the code operand designator, assuming that the RRS has issued instructions to move the second operand to the local and global storage respectively. Mark the operand designator as determined (SS=1), using the following code segment: {\{GORQ2T1.SDR (operand) = GORQ2T1.SDR (operand) \cup AGDS (STALLOCMD (Type(operand), operand ) GORQ2T1.SS (operand) =1; code (operand) =code (GORQ2T1.SDR (operand, GDS))
```

Semantic action 4: As semantic action 3, but assuming that the RRS has issued instructions to move the destination operand to the global storage.

Semantic action 5: Replace the code operand designator by the global storage designator and return such designator {\{GORQ2T1.SDR (operand, GDS)\}}.
Semantic action 6: Determine and add storage designator for local computation to the code operand designator, assuming that the RRS has issued instructions to move the destination operand to the local storage. The following code segment represents these actions
\[ \{ \text{GORQ2T1.SDR (operand)} = \text{GORQ2T1.SDR (operand)} \cup \text{ALDS (STALLCMD (Type (operand), operand))} \] 
Semantic action 7: Replace the code operand designator by the storage designator for local computation; and return such designator (GORQ2T1.SDR (operand, LDS).

Table 1: The MDCOS Function.

<table>
<thead>
<tr>
<th>Operand Type</th>
<th>Semantic Action(1,2,…,7)</th>
<th>Storage Class(SC) /Storage status (SS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value-Opnd</td>
<td></td>
<td>SC1/SS0 SC1/SS1 SC2/SS0 SC2/SS1 SC3/SS0 SC3/SS1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2 3 5 3 5</td>
</tr>
<tr>
<td>Exp-opnd</td>
<td></td>
<td>6 ------ 4 ----- 4 ----</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>------ 7 ------ 5 ------ 5</td>
</tr>
<tr>
<td>Exp-opnd</td>
<td></td>
<td>L-value Exp-opnd</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>------ 3 ..... 3 ...</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>------ 7 ------ 5 ------ 5</td>
</tr>
<tr>
<td>L-value Def-opnd</td>
<td></td>
<td>------ 7 ------ 5 ------ 5</td>
</tr>
<tr>
<td>R-value Def-opnd</td>
<td></td>
<td>------ 7 ------ 5 ------ 5</td>
</tr>
<tr>
<td>R-value</td>
<td></td>
<td>------ 7 ------ 5 ------ 5</td>
</tr>
<tr>
<td>Adr-opnd</td>
<td></td>
<td>------ 7 ------ 5 ------ 5</td>
</tr>
</tbody>
</table>

6. EXPERIMENTAL RESULTS

To demonstrate the proposed code generator, we have conducted the following experiments:

**Instantiation of GCG by a Virtual Machine**

We define a virtual machine (VM) with unlimited number of registers. The registers assume a variety of functions such as accumulator, index, base, frame pointer, stack pointer, and special purpose registers etc. The Virtual machine has an instruction set with one and two address format of the form: OP-code OPND1 OPND2. Where: Op-code represents the operation performed by the instruction, OPND1 and OPND2 are the source and the destination operands with the following combinations: register-register, register-memory, memory–register and memory-memory. The VM machine supports the same data types as the ones supported by the source language, with memory allocation organized as stack frames, register files and a heap. Its addressing modes are defined to express the most suitable ones for the different language constructs. They represent the most available ones on real machine.

According to our approach, the instantiation of GCG by the above defined virtual machine is reduced to the instantiation of the Target machine (level 2) = (ML2, MD2, OPS2) by the specification of such machine, as described in the following sub-section. The GTIOP2 is then applied to generate an optimized program in the VM machine language (MLvm), as described later.

**Instantiation of the Target Machine (Level 2) by the Virtual Machine.**

The instantiation of the Target machine (level 2) = (ML2, MD2, OPS2) by the virtual machine produces VM = (MLvm, MDvm, OPSvm) where:

- MLvm is the VM machine language, obtained as result of instantiating ML2 by the VM instructions, as described in the following sub-section.
- MDvm is the instantiated MD2= (STMD, STALLOCMD, TSAS).Where: STMD is defined as for Target machine (level 1) , STALLOCMD is instantiated by Table 2 and TSAS is reduced to a scheme that determines the storage designators of the operands allocated on stack frames.
- OPSvm is the implementation of OPS2 as VM dependent optimization scheme. It is reduced to the function MDCOS, as given in Table 1.

**The VM Machine Language MLvm**

The MLvm language is specified by instantiation of ML2 by VM instructions as follows:

- The VM machine instructions set is represented
using the generic format: 
\[ \emptyset \{ \} \text{Topnd}_1 \{ \} \text{Topnd}_2 \{ \}. \] Hence, the generic rewrite scheme is instantiated by the form:

\[
\theta_{\text{Topnd}_1 \text{Topnd}_2} \rightarrow \text{Topnd}_1 \text{Topnd}_2 \emptyset \] 

Table 2: The Storage Allocation Scheme (STALLOCMD).

<table>
<thead>
<tr>
<th>Construct Type</th>
<th>Addressing Mode</th>
<th>Storage Designator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operands frequently used</td>
<td>Global symbol</td>
<td>GR1, GR2,…,GRn</td>
</tr>
<tr>
<td>Global operands</td>
<td>Immediate</td>
<td>Val</td>
</tr>
<tr>
<td>Scalar variables, local/</td>
<td>Base-displacement</td>
<td>[BR +D]</td>
</tr>
<tr>
<td>nonlocal variables, starting address of</td>
<td>indirect</td>
<td></td>
</tr>
<tr>
<td>complex data structures and dynamic data structures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear array of scalars, element in vector of scalars</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{MC} (A_{\text{ML}1}, A_{\text{ML}1}) & \rightarrow \alpha_{\text{ML}1} = (\text{MC} (A_{\text{ML}vm}), A_{\text{ML}vm} \rightarrow \alpha_{\text{ML}vm}) \text{ where:} \\
\alpha_{\text{ML}vm} & = X \text{Topnd}_1 X \text{Topnd}_2 \emptyset \text{Topnd}_1 \text{Topnd}_2 \\
T_{\text{AML}vm} & - \alpha_{\text{ML}vm} \text{ is considered as header for templates of VM instructions, represented in generic format and defining the meaning of } \alpha_{\text{ML}1} \text{ based on the instantiated matching criteria } \text{MC} (A_{\text{ML}vm}). \\
\text{For example, the instantiated rewriting rules IRSS1, IRSS2 respective to the operands of the types: (DEF, ADR) and (EXP, VALUE) are as follows:} \\
\text{IRSS1: } & (\text{MC} (\text{Def}), \text{Def}) \rightarrow := \text{ADR \ } \text{Rvalue}_{\text{ML}1} \rightarrow \\
& (\text{MC}(\text{Def}), \text{Def}) \rightarrow X \text{Adr X Rvalue} := T_{\text{Adr}} T_{\text{Rvalue}} \\
& T_{\text{ML}vm} \text{ where:} \\
& - (\text{MC} (\text{Rvalue}), \text{Rvalue})_{\text{ML}1} \rightarrow (\text{Def} | \text{Exp} | \text{Const} | \text{Value})_{\text{ML}1} \\
& - ((\text{MC}(X \text{Adr}), X \text{Adr})_{\text{ML}vm} \rightarrow (T \text{Adr})_{\text{ML}vm} \\
& - (MC(X \text{Rvalue}, X \text{Rvalue})_{\text{ML}vm} \rightarrow (T \text{Rvalue})_{\text{ML}vm} \\
& - (:= T_{\text{Adr}} T_{\text{Const}} T_{\text{Def}})_{\text{ML}2} \text{ is as header for the following templates:} \\
& - (:= T_{\text{Adr}} T_{\text{Rvalue}})_{\text{ML}2}, \text{ if } \text{MC}(\text{Def})_{\text{ML}vm} = (1,0) , \text{where} \\
& T_{\text{Def}} = T_{\text{ADR}} \\
& - (:= T_{\text{Adr}} T_{\text{Rvalue}} ; := T_{\text{Def}} T_{\text{Rvalue}})_{\text{ML}vm}, \text{ if } \text{MC}(\text{Def})_{\text{ML}vm} = (2,0) \text{ or } (3,0) \\
\text{IRSS2: } & (\text{MC}(\text{Exp}), \text{Exp}) \rightarrow + \text{Rvalue Rvalue}_{\text{ML}1} = \\
& (\text{MC} (\text{Exp} LIR2), \text{Exp} \rightarrow X \text{Rvalue} X \text{Rvalue} + T_{\text{Rvalue}} T_{\text{value}} \\
& T_{\text{Exp}}_{\text{ML}2}) \text{ where:} \\
& - (\text{MC} (\text{Rvalue}), \text{Rvalue})_{\text{ML}1} \rightarrow (\text{Def} | \text{Exp} | \text{Const} | \text{Value})_{\text{ML}1} \\
& - (MC(X \text{Rvalue}, X \text{Rvalue})_{\text{ML}vm} \rightarrow T \text{Rvalue} \\
& - (+T_{\text{Value}} T_{\text{Exp}} T_{\text{Exp}})_{\text{MLvm}} = \text{is as header for the template:} \\
& - (:= T_{\text{Exp}1} T_{\text{Value}} ; + T_{\text{Exp}1} T_{\text{Rvalue}}), \text{ for all } \text{MC}(\text{Exp1})_{\text{MLvm}}. \\
\end{align*}

Implementation of MLvm

The implementation of MLvm proceeds as follows:
- GTIOPS2 is instantiated as:
  GTIOPS2: (RRSvm (ML1) \rightarrow MLvm) \times MDvm \times OPS2vm \rightarrow MLvm \times OPR2vm
- The input of Algorithm- GTIOPS2 is instantiated by FG (ML1), GORQ2vm Tables, RRSvm Table, STALLOCMDvm Table, TSASvm scheme and MDCOS Table.
- Algorithm- GTIOPS2 is applied on FG (ML1).

As a result, FG (MLvm) is obtained and as shown in figure 4. Where each basic block contains the specific VM instructions, constituting the MLvm constructs respective to the ML1 ones.

Instantiation of GCG by Intel IA-32

According to our approach, the instantiation of GCG by Intel IA-32 is reduced to the instantiation of the Target machine (level 2) = (LIR2, MD2, OPS2) by the Intel IA-32 specification, as described in the following sub-section. The GTIOP2 is then applied to generate an optimized program in the Intel IA-32 machine language (MLin), as described later.

Instantiation of the Target Machine (Level 2) by Intel IA-32

The instantiation of the Target machine (level 2) = (ML2, MD2, OPS2) by the Intel IA-32 produces Intel IA-32 = (MLin, MDin, OPSin), where:
- MLin is the Intel IA-32 machine language, obtained as result of instantiating ML2 by Intel IA-32 instructions, as described in the following sub-section.
BB 1: := BR+Da Const 1
 := GR1 BR+Da
 := BR+Db Const 1
 := GR2 BR+Db
 := BR+Dd Const 1
 := GR3 BR+Dd
 := BR+Da Const 1
 BB2: > GR3 Const
 := GR6 GR2
 BB4: := GR6 GR3
 BB3: := GR4 GR1
 + GR4 GR2
 := BR+Dd GR4
 BB5: > GR3 Const
 := BR+Dd GR3
 := BR+Dd GR5
 := BR+Dc GR4
 := BR+Db GR4
 := BR+Dc GR5
 := BR+Dd GR4
 BB6: := GR3 GR4
 BB7: := GR2 GR4
 BB8: := GR2 GR6
 BB9: := GR6 GR2
 * GR6 GR3
 := GR2 GR6
 - GR2 GR3

Figure 4: The VM Target Program.

- MDin is the instantiated MD2= STMD+ STALLOCMD, where:
  - STMDin = (memory locations (stack frames, heap), set of general (and special) purpose registers) = (linear space of 8-bit cells, eight 32-bit general purpose registers (eax, ecx, edx, ebx, esp, ebp, esi, edi), special purpose registers(16-bit segment registers (CS, DS), instruction pointer(EIP) ,control registers , flags , ) , floating point unit registers(…))
  - STALLOCMD is instantiated for Intel IA-32 as follows:
    - STALLOCMD for register operands is defined as: STALLOCMD1in = (Construct-type = all construct types , Addressing mode= register addressing mode, storage designator = GR), where: GR= (GR-32 ,GR-16, or GR-8) denotes the set of registers such as {eax, ecx, edx, ebx, esp, ebp, esi, edi}.
    - STALLOCMD for memory operands is defined as STALLOCMD2, as given in Table 2. But, with the following interpretation. GR is as defined above; D = (D-32, D-16 and D-8) denotes signed 32 bit, 16 bit, or 8 bit displacement respectively. Scale denotes scaling values 1, 2, 4 and 8. BR denotes base register represented by ebp for the stack frame variables or by any GR, otherwise. GR1, GR2…and GRn denote global memory location, based on the assumption that such location will be mapped into internal registers during execution. IR denotes index computation using the registers esi and edi.

Instantiation of ML2 by IA-32 (MLin)

IA-32 instruction set is represented by the generic format: θ { } Topnd, { } Topnd, { }.Where: Topnd, and T opnd, represent generic designators for the source and the destination operands respectively. These operands are defined either as register (r8, r16, r32), memory(m8,m16,m32) or immediate( imm8, mm16, imm32) designators, assuming different addressing modes. Semantic actions are defined to replace Topnd, and Topnd, by the respective machine dependent ones, based on the machine dependent global optimization scheme and on the relationship between the instructions and the addressing modes. The relationship lists the possible destination - source combinations. Based on the templates given in (Bastian and Onder, 2005), the destination - source combinations are reduced to 18 templates ( r8- r8, r8-m8, r8-imm8,…,) and for each instruction a subset of such templates are defined. The instructions appropriate to the individual ML1 constructs (AML1) are selected from these templates based on the matching constraints of the MLin constructs (MC (AMLin)), inherited from the ones (MC (AMLin)) of their respective ML1 constructs. Where, the operand types (integer, real, …,) and their context (simple variable, indexed variable,…,) are instantiated into machine dependent ones (signed integer-32, signed integer-8, float,…,) in the respective MC (AMLin). Thus, the instantiation of MLin by the IA -32 instruction set is
reduced to the instantiation of the generic rewrite scheme by appropriate instructions, as illustrated by the following example.

Let the considered ML1 construct is: BB1: (MC (Def11), Def11 → := Adr a 12 Const 1 13), where:

(MC (Def11) = SC x SS x Type = 3x0 x int) , (MC (Adr a 12) = SC x SS x Type = 3x0 x int) and (MC (Const 1 13) = SC x SS x Type = int). The respective rewrite rule is:

BB 1 ( MC(Def11), Def11 → X Adr a 12 X Const 1 13 := TAdr a 12 T Const 1 13 TDef11 ), where:

- MC(Def11) = MC (Def11) ∪ int-32, MC (Adr a 12) = MC (Adr a 12) ∪ int-32
- The selected template respective to ( TAdr a 12 T Const 1 13 TDef11) is:
  - The function IOS determines the operand storage designators (SDR) as follows: The SDR for Def11 is GR1; the SDR for (X Adr a 12 X Const 1 13) is ([BR-32 + Da], GR1); and the SDR for ( X Const 1 13 := T Const 1 13 ) is (D-32, val). Hence, the produced results are ( move [ebp + Da], 1; mov GR1, 1).

**Implementation of the Machine Language ML1**

The implementation of ML1 proceeds as previously described, where the input of GTIOPS2 is instantiated by the tables, schemes respective to IA-32 and as previously given. Furthermore, GTIOPS2 is augmented with instruction selection (INSTSEL) and register allocation (REGALLOC) strategies, appropriate to IA-32. INSTSEL is based on the machine dependent attributes of MC (DefMLin). Hence, it is reflected by appropriate rewriting rules. The REGALLOC strategy is based on the fact that for each basic block, the global registers and the scratch ones are determined in terms of their numbers and allocation. The global registers are allocated within the framework of the global storage allocation scheme (AGDS), if the operands are of type register. AGDS is instantiated to allocate global memory (GR) for the memory operands, assuming GR will be mapped to one of the internal registers, during execution. Hence, REGALLOC constitutes an instantiation to ALDS and is reduced to allocate either the special registers needed by some of the instructions or scratch registers for the local operands within the individual basic blocks. For each basic block the number of the scratch registers to be allocated is determined to be maximum 2. The scratch registers are selected from the free use registers. The operands of the individual instructions are allocated to these registers according to their respective order, unless the instruction requires special registers, the operands either are memory operands, global ones or represent immediate value. The factors on which the register allocation is based are formulated as context dependent conditions (cc) and considered as an argument to ALDS function in order to reflect the effect of REGALLOC within the semantic actions. The results of the application of GTIOPS2 are shown in Figure 5. Example to illustrate how GTIOPS2 proceeds is previously given. Figure 5 illustrates the output produced by GTIOPS2 for different ML1 constructs with different operand types.

**Instantiation of GCG by UltraSparc**

The instantiation of GCG by UltraSparc proceeds as follows:

The target machine (leve2) is instantiated by UltraSparc (level 2) = (MLul, MDul, OPSul), where: The meta data model MDul= (STMDul, STALLOCMDul); STMDul = (memory locations (stack frames, heap), set of general (and special) purpose registers) = (linear space of 8-bit cells, Thirty-two 32-bit general purpose registers (R[0]-R[33]); and STALLOCMDul is instantiated for UltraSparc as follows:

STALLOCMDul for register operands is defined as: TALLOCMDul = (Construct-type = all construct types , Addressing mode = register addressing mode, storage designator = GR), where: GR= denote the set of registers {R,FQ, FD, FS} and STALLOCMDul for memory operands is defined as STALLOCMDul and as given in Table 2, with the following interpretation:

GR is as defined above. D denotes a displacement as one of the following: signed 7 bit (simm7), signed 8 bit (simm8) bit, simm10, simm11, const4, const22 and value (64 bit). BR is a base register represented by: sp (R [30]), fp (R [14] for the stack frame variables or by any R, otherwise. (GR1, GR2, ..., GRn) denote global memory location , based on the assumption that such location will be mapped into the global registers (R [0] – R[7]) during instantiation. IR denotes index computation using any one of the registers R. The machine language at level 2 (MLul) is instantiated and implemented as follows:

1) UltraSparc instruction set is represented by the generic formats:

θ { } Topnd1 { } T opnd2 { } T opnd3 { } and θ { } Topnd1 { } T opnd2 { } . Where: Topnd1 and T opnd2 represent generic designators for the source operand while T opnd3 represents the destination operands. These operands are defined as follows:
### Figure 5: The IA-32 and UltraSparc Target Programs.

<table>
<thead>
<tr>
<th>ML1 Constructs</th>
<th>IA-32 Instructions</th>
<th>UltraSparc Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB 1: Def_{11} → := Adr a_{12} Const 1_{13}</td>
<td>mov [ebp+Da], 1, 1 mov GR1, 1</td>
<td>mov 1, [fp+Da] mov 1, GR1</td>
</tr>
<tr>
<td>Def_{14} → := Adr b_{15} Const 2_{16}</td>
<td>mov [ebp+Da], 2 mov GR2, 2</td>
<td>mov 2, [fp+Db] mov 2, GR2</td>
</tr>
<tr>
<td>Def_{17} → := Adr d_{18} Const 5_{19}</td>
<td>mov [ebp+Da], 1 mov GR3, 5</td>
<td>mov 1, [fp+Dd] mov 5, GR3</td>
</tr>
<tr>
<td>BB 2: Exp_{21} → &gt; Def_{17} Const 30_{23}</td>
<td>cmp GR3, 30</td>
<td>cmp , GR3</td>
</tr>
<tr>
<td>BB 3: Exp_{31} → + Def_{11} Def_{14}</td>
<td>mov eax, GR1 add eax, GR2 mov GR4, eax</td>
<td>mov GR1, GR2, GR4</td>
</tr>
<tr>
<td>Def_{34} → := Adr c_{35} Exp_{31}</td>
<td>mov [ebp+Dc], eax st GR4, [fp+Dc]</td>
<td></td>
</tr>
<tr>
<td>Exp_{35} ← - Def_{34} Def_{11}</td>
<td>sub ebx, GR1 mov GR5 , ebx</td>
<td>sub GR4, GR1,GR5</td>
</tr>
<tr>
<td>BB 4: Exp_{41} ← * Def_{14} Def_{17}</td>
<td>mov ebx, GR2 umul GR2 , GR3,R11</td>
<td></td>
</tr>
<tr>
<td>Def_{44} → := Adr d_{45} Exp_{41}</td>
<td>mov GR3, ebx mov R11, GR3 mov [ebp+Dd], ebx st R11, [fp+Dd]</td>
<td></td>
</tr>
<tr>
<td>BB 5: Exp_{51} → &gt; Def_{17} Const 25_{53}</td>
<td>cmp GR3, 25</td>
<td>cmp 25, GR3</td>
</tr>
<tr>
<td>BB 6: Def_{64} → := Adr d_{65} Exp_{31}</td>
<td>mov eax,GR4 mov R12, GR4,GR3 mov GR3 ,eax st GR4, [fp+Dd]</td>
<td></td>
</tr>
<tr>
<td>Exp_{66} ← + Value e_{67} Const 1_{68}</td>
<td>mov eax, [ebp+De] ld R12, [fp+De]</td>
<td></td>
</tr>
<tr>
<td>Def_{69} → := Adr e_{610} Exp_{66}</td>
<td>mov [ebp+De], eax st R13, [fp+De]</td>
<td></td>
</tr>
<tr>
<td>BB 8: Def_{84} → := Adr b_{85} Exp_{31}</td>
<td>mov eax GR4 mov GR4,GR2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov [BR+Db], eax mov GR2 ,eax</td>
<td></td>
</tr>
<tr>
<td>Def_{89} → := Adr c_{810} Exp_{35}</td>
<td>mov eax,GR5 mov GR5, fp+De</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov [BR+Dc], eax</td>
<td></td>
</tr>
<tr>
<td>BB 9: Exp_{91} ← + Def_{14} Def_{17}</td>
<td>mov ebx,GR2 add ebx GR3 mov R12, GR2,GR3</td>
<td></td>
</tr>
<tr>
<td>Def_{94} → := Adr a_{95} Exp_{91}</td>
<td>mov BR+Da, ebx st R12, GR1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov GR1 , ebx</td>
<td></td>
</tr>
</tbody>
</table>
- Topnds\(_1\) and T opnds\(_2\) are defined as one of the registers R, or as a memory address (in the instructions with the format \(\theta\{\}\ Topnds\(_1\)\{\}\ T opnds\(_2\)\{\}\)). Such address is specified as reg-plus-imm or as reg-plus-reg.

- T opnds\(_2\) is defined as one of the registers R, or as immediate (simm\(_7\),.., imm13) designators.

2) The instantiation of MLul by the UltraSparc instruction set is reduced to the instantiation of the generic rewrite scheme by appropriate instructions. For example, the rewriting rules respective to the operands of the types: DEF, ADR, VALUE and EXP are as follows:

- The rewriting rules respective to the operands of the type DEF

\[
\text{RRS} \ (\text{MC} (\text{Def}_{\text{ML1}}, \text{Def}_{\text{MLul}} \rightarrow \alpha_{\text{ML1}})) = (\text{MC} (\text{Def}_{\text{ML1}}, \text{Def}_{\text{MLul}} \rightarrow \alpha_{\text{MLul}})). \text{ Where:}
\]
\[
\alpha_{\text{MLul}} = \begin{cases} 
X_{\text{MLul}}..X_{\text{ML1}} & \text{if } \alpha_{\text{ML1}} = X_{\text{ML1}}..X_{\text{MLul}} \\
X_{\text{ML1}}X_{\text{MLul}} & \text{movTX2TX1, where } T_{\text{MLul}} = TX1 \\
\text{if } \alpha_{\text{ML1}} = (X_{\text{ML1}}X_{\text{MLul}}) & \text{MC(Def}_{\text{ML1}}(1,0)\text{andTX2 = const} \\
X_{\text{ML1}}X_{\text{MLul}} & \text{movTX2TX1; st } T_{\text{MLul}} = TX1 \\
\text{if } \text{MC}(\text{Def}_{\text{ML1}}) = (2,0) \text{or}(3,0) \\
TX_{\text{ML1}} & \text{if } \alpha_{\text{ML1}} = T_{\text{MLul}} \end{cases}
\]

3) The implementation of MLul proceeds by applying Algorithm- GTIOPS2 as previously described. However, GTIOPS2 is instantiated by the tables, schemes respective to UltraSparc. Figure 5 illustrates the output produced by GTIOPS2 for different ML1 constructs with different operand types. For examples, ( BB3 add GR1, GR2, GR4) is the respective MLul instruction to the global ML1 construct of type EXP (BB 3 Exp\(_1\) \rightarrow + Def\(_{12}\) Def\(_{14}\)) with global operands.

(umul GR2 , GR3 R11) is the respective MLul instruction to the Local ML1 construct of type EXP(BB 4 Exp\(_4\) \rightarrow * Def\(_{14}\) Value d\(_{43}\)) with global operands.

7. EVALUATION

The proposed GCG has been evaluated in terms of the following:

1) A comparison has been made between the approach on which GCG is based and the similar ones as given in the following sub-section.

2) The GCG has been instantiated for three machines as previously described. Table 3 shows the computational results of such instantiations in terms of the following criteria: Size of the instantiated rewriting rule (GRSS), size of the instantiated machine language, number of the generic and the specific templates, defining the machine instructions, and finally the specific routines within the translation and instantiation scheme GTIOPS2.

### Table 3: The Instantiations Results.

<table>
<thead>
<tr>
<th>Evaluation criteria</th>
<th>Target Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GCGvm</td>
</tr>
<tr>
<td>Size of instantiated GRSS Relative to Size (G(ML1))</td>
<td>2.6</td>
</tr>
<tr>
<td>Size of instantiated ML2 Relative to Size (G(ML1))</td>
<td>1.6</td>
</tr>
<tr>
<td>Generic templates per operand type</td>
<td>3</td>
</tr>
<tr>
<td>Specific templates per operand type</td>
<td>3</td>
</tr>
<tr>
<td>Specific (instantiated) optimization per operand type</td>
<td>-</td>
</tr>
</tbody>
</table>

The major entities (the intermediate languages, actions and data structures) used by the three instantiations are classified into the categories: common, parameterized and instantiated, with the following computational characteristics:

A. The common category:

ML1 Language: size (ML1) = O(G(ML0)), where

\(G(ML0) = \sum |pi| \text{ALD, } |pi| \text{ is the size of production (i) and ALD is constant reflecting the degree of the alternative production definitions}

\[\text{OPR1: size (OPR1)} = O\left(\text{OPNDTYPE}\right)\ \ast \]

Occurrence Frequency

\[\text{GTIOPS1:time (GTIOPS1)} = O\left(\text{OPR1}\right)\ \}

\[\text{GORQ2: } \sum \_{i} \ \text{size (GORQ2-Category i)} < (1/Cop)\ast\]

OPR1, where Cop is a constant reflecting the optimization effect.

GRSS: size (GRSS) = O (2\left|G(ML1)\right| + \
\text{Generic instruction format})

ML2: size(ML2) = O\left(G(ML1)\right) + \text{Generic instruction format \ast Matching Constraints} \}

B. The instantiated and parameterized entities:
GRSSi: Size (GRSSi) = size (GRSS) + O (|Matching Constraints| * Cr), where: Cr is a constant which reflects the degree of variations of the specific instructions templates

ML2i: Size(ML2) + O (Cr * |matching constraints|)

GTIOPS2: time (GTIOPS2) = O (No. Basic blocks* Max|basic block|* |GRSS|* |ML2i|* Csac

where: Csac is a constant reflecting the maximum time needed by the semantic actions.

Comparison with Similar Approaches

The quality of a code generator is reflected in a number of factors such as machine description, the embedded optimization, correctness, generality and ease of retargetability. Based on such factors, our approach is compared to the similar ones as follows:

Compared to the machine description used by GCC framework (GCC, 2009), GCG is systematic and formalized; it has less size and fewer amounts of details and repetitiveness. For example, GCC framework includes similarities in 63 macros and 769 distinct macros (Bastian and Onder, 2005). On contrast, the proposed GCG is based on 3-18 generic templates that can be instantiated to specific ones during code generation. Similar approach to the GCG machine description is the one proposed in (Bastian and Onder, 2005). Where, reducing the distinct description for every combination of x86 code with addressing modes is achieved by defining an encoded pattern for addressing modes and letting the instruction inherits the right pattern. However, in our approach the addressing modes are represented as part of the machine meta data model and imposed on the instruction during instantiation, as previously described.

The proposed GCG is based on incremental approach. Similar approaches can be classified in to categories. The first category is represented by the work proposed in (Deshpande and Khedker, 2007). It is based on decomposing the code generator into levels that covers aspects of the source language in levels that covers aspects of the incremental way. In contrast, our approach is based on decomposing the code generator into target machine in incremental way. However, such decomposition is influenced by the source language. Thus, it permits integration of the code generation phases (Jia et al., 2008; Errikson et al., 2008). The second category is represented by the machine description suggested in (Oh and Paek, 2003) and in (Dias and Ramsey, 2006). The description is hierarchical and consists of two levels. In (Oh and Paek, 2003), the two levels are defined as a high level to represent the over all target ISA and a low level for representing its micro architecture. In (Dias and Ramsey, 2006), the two levels are defined as a low level RTL and a high level represented as assembly language. Our approach permits both descriptions. However each level is represented in terms of a meta data model and a machine language, defined as rewriting rules and with imposed optimization effect. Since the work in (Dias and Ramsey, 2006) is similar to the proposed GCG, Table 4 shows the computational results (CG) as given in (Dias and Ramsey, 2006) and the ones for the GCG instantiated by Intel-32. Where CG uses RTL to describe instructions and Burg style to recognize and select respective assembly instructions.

Table 4: The Computational Results of CG and GCG.

<table>
<thead>
<tr>
<th>Number of f instruction</th>
<th>CG (Dias and Ramsey, 2006)</th>
<th>GCG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RTL code</td>
<td>BURG code</td>
</tr>
<tr>
<td>630</td>
<td>1160</td>
<td>754</td>
</tr>
<tr>
<td>64</td>
<td>96</td>
<td>384</td>
</tr>
</tbody>
</table>

The proposed GCG is embedded with global optimization. Similar approach is proposed in (Rogers, et al., 1999), where: 8 registers were needed to cover 30% of the instruction count. In contrast, 5 (2) registers were needed by GCG cover global (local) variables in 67.5% (42.5%) of the instructions count, respectively.

The correctness of the code generators is achieved by certifying formalism (Hickey, et al., 2003). In contrast, the correctness of GCG is demonstrated by the fact that the type information is propagated from the source language to the subsequent machine languages of the different target machine levels. It is used as matching criteria for the selection of the specific instructions and the appropriate addressing modes.

8. Conclusion

In this paper, we have proposed and implemented a generic code generator based on a formal and optimizing approach, characterized by its generality, efficiency and correctness. Such approach systematically decomposes the code generator, and subsequently the machine description, into levels. Each level constitutes a code generation phase and the same time a generic representation of respective aspects of the target machine.
The generality and the ease of GCG retargetability have been demonstrated by its instantiations for a register virtual machine, a CISC machine (Intel-32) and a RISC machine (UltraSparc). The experimental results have shown that the size of the rewriting rules, and respectively the size of the target program, for UltraSparc are relatively smaller than the ones for Intel-32 and the virtual machine. This is due to their instruction format and respectively the need for more moves instructions. However, within the framework of the proposed approach, their respective addressing modes are efficiently utilized. Compared to the publicly available computational results of other similar approaches, the instantiations results of the proposed GCG have proved to be competitive. For examples: a relatively less number of distinct instructions templates within a generic rewriting scheme and a relatively higher instructions count covered by the effect of a global optimization. Future work is as follows: study the possibility of eliminating the redundant moves instructions; automate the hand written rewriting rules; and introduce further target machine levels to cover instructions scheduling.

REFERENCES


