Reducing Code Size for Heterogeneous-Connectivity-Based VLIW DSPs through Synthesis of Instruction Set Extensions

Partha Biswas
partha@cecs.uci.edu

Nikil Dutt
dutt@cecs.uci.edu

Center for Embedded Computer Systems
School of Information and Computer Science
University of California, Irvine, CA 92697

ABSTRACT

VLIW DSP architectures exhibit heterogeneous connections between functional units and register files for speeding up special tasks. Such architectural characteristics can be effectively exploited through the use of complex instruction set extensions (ISEs). Although VLIWs are increasingly being used for DSP applications to achieve very high performance, such architectures are known to suffer from increased code size. This paper addresses how to generate ISEs that can result in significant code size reduction in VLIW DSPs without degrading performance. Unfortunately, contemporary techniques for instruction set synthesis fail to extract legal ISEs for heterogeneous-connectivity-based architectures. We propose a Heuristic-based algorithm to synthesize ISEs for a generalized heterogeneous-connectivity-based VLIW DSP architecture. We achieve an average code size reduction of 25% on the MiBench suite with no penalty in performance by applying our ISE generation algorithm on the TI TMS320C6xx, a representative VLIW DSP.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles

General Terms

Instruction Set Extensions, Instruction Set Architecture, Static Single Assignment

Keywords

Heterogeneous-Connectivity-based DSP, Restricted Data Dependence Graph, Dependence Conflict Graph

1. INTRODUCTION

Embedded processors for Systems-on-Chip present unique opportunities to increase performance under stringent requirements of reduced code size, low power and low cost. A digital signal processor (DSP) is a class of embedded processors designed to perform common operations on digital encodings of analog signals. The operations carried out are signal processing tasks and are generally math-intensive. To support such operations effectively, a DSP architecture typically has special-purpose datapath units and special-purpose connections between units and register files. Such special-purpose processors can also serve as coprocessors to the main processors.

In order to boost performance, modern DSPs are increasingly employing VLIW-style architectures that can execute more instructions in parallel. A VLIW DSP, by virtue of having a regular instruction set presents a compiler-friendly processor model at the cost of larger code size. When these processors are embedded on a chip together with instruction memory, the code size has to be limited. The problems that VLIW architectures have with code size often confine their application to time-critical code segments. We present a novel algorithm to augment the instruction set of a VLIW DSP with complex instructions that can significantly reduce the code size. Our algorithm guarantees that the performance realizable by the compiler is preserved.

We call the VLIW DSP architecture with functional units having restricted accesses to register files, a **Heterogeneous-Connectivity-based DSP** or simply an **HCDSP**. The TI TMS320C6xx [1] processor is an example of an HCDSP that issues 8 instructions per cycle to 8 functional units partitioned into 2 clusters. Conceptually, the TMS320C6xx architecture contains 2 register files with any functional unit in each cluster able to access one or both register files based on the connectivity.

There has been a large body of work on synthesizing Instruction Set Extensions (ISEs) for special purpose DSP s. However, in the presence of heterogeneous connectivity between the register files and the functional units, contemporary techniques used for synthesizing ISEs fail to exploit and generate legal extensions to the instruction set. In this paper, we present a Heuristic-based algorithm that synthesizes new complex instructions for extending the instruction set of HCDSPs with the goal of code size reduction. On a representative HCDSP architecture (the TI TMS320C6xx

[1]), we demonstrate the ability of our algorithm to achieve significant code size reduction (up to 37%) over the base instruction set architecture through synthesized ISEs with no loss in performance.

The rest of the paper is organized as follows. In Section 2, we discuss related research work. Section 3 presents the motivation for our work. Section 4 discusses our model of an HCDSP architecture. Section 5 presents the latency constraints for the architecture. In Section 6, we present a heuristic-based algorithm to minimize the code size. Section 7 outlines our entire framework. Section 8 shows the efficacy of our algorithm on the MiBench suite. Finally, Section 9 concludes the paper.

2. RELATED WORK

We discuss related research work in two domains: code size reduction in VLIW DSP processors and application-specific instruction set synthesis.

The hardware solutions ([2],[3],[4],[5],[6]) developed for minimizing code size in VLIW processors mainly focus on changing the instruction formats to incorporate new templates which can lead to compressed code. A typical approach stores instructions in a compressed form in both memory and instruction cache. The instructions are expanded each time they are fetched from cache. The code size reduction thus comes at a cost of increased complexity in the control path. The software solution ([7]) to the same problem is integrated in a compiler which trades-off code size for performance while scheduling code for the VLIW processor. Our approach is complementary to these previous approaches, since we reduce code size by synthesizing ISEs with no penalty in performance.

Several research efforts have studied instruction set synthesis. One of the important steps used in automatic synthesis of ISEs is **Clustering** of atomic instructions (i.e., instructions that cannot be further subdivided). This clustering step can take two flavors: clustering dependent instructions and clustering parallel instructions. In the context of pipelined RISC processors, a complex instruction obtained by clustering instructions connected through a dependency chain in the data flow graph results in increased performance by exercising forwarding paths between the execution units. This is exemplified by a recent work [8] that generated ISEs for a pipelined RISC processor under bitwidth constraints and demonstrated a significant increase in performance over the native instruction set. The other kind of clustering (that groups independent instructions) was used in architectures containing parallel execution units with the goal of minimizing code size in DSP processors [9] [10].

The ISEs were also used as specialized instructions in coprocessors, which are extensions to the main processor instruction set ([11], [9], [12] and [13]). The main goal in these efforts was to maximize performance of the system in the presence of coprocessor(s) supporting specialized ISEs. Both kinds of clustering were employed in synthesizing ISEs with a bound on the number of read/write ports in the register file.

To the best of our knowledge, no work has yet been done to synthesize complex instructions for HCDSP processors. Due to its heterogeneous connectivity between the register files and the functional units, the HCDSP presents a more generalized model of a VLIW DSP than a simple VLIW architecture. We define a complex instruction (that can occupy one of the VLIW slots) as a composition of base instructions connected with each other by a read-after-write dependency chain. Our goals are to minimize the generated code size as well as to minimize the number of new ISEs generated. The second goal ensures that there is maximum reuse of the ISEs.

3. MOTIVATION

Figure 1 shows the execution time and memory usage in a representative benchmark for different VLIW DSPs. Out of these DSPs, TMS320C6xx is the only one with an HCDSP architecture and is the fastest. However, it also has the highest program memory bandwidth requirements. If we can extend the instruction set of the TMS320C6xx with useful complex instructions, the memory requirement can be decreased substantially to make the architecture comparable with other VLIW DSPs in terms of memory usage, without sacrificing performance.

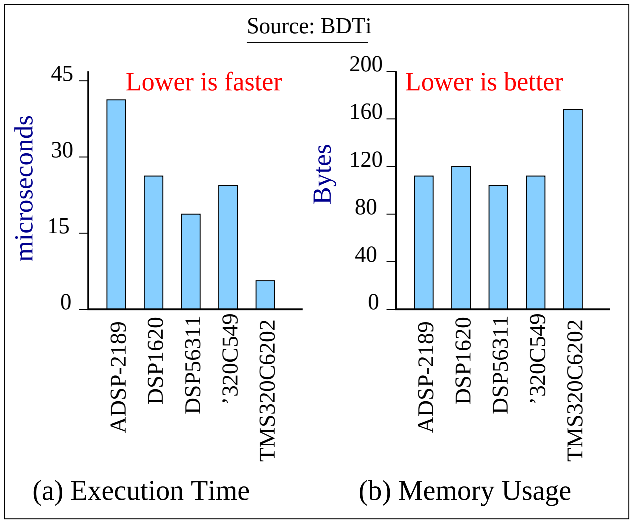

Figure 1: VLIW DSPs: (a) Execution Time on Complex Block FIR (b) Memory Usage on FSM Benchmark

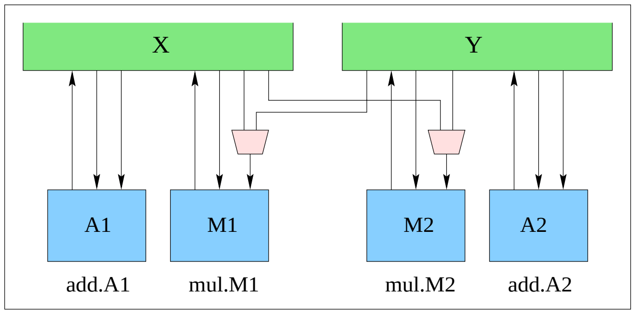

Figure 2: An Example of HCDSP Architecture

We present in Figure 2, a typical scenario of an HCDSP architecture. Instructions *add.A1*, *add.A2*, *mul.M1* and *mul.M2* can only be executed by functional units *A1*, *A2*, *M1* and

M2 respectively. The instruction *add.A2* cannot read registers written by *mul.M1* because *M1* can only write into the register file *X* and *A2* can read source operands only from the register file *Y*. Similarly, *mul.M2* writes into registers which cannot be read by *add.A1* based on the connections of functional units *A1* and *M2* to register files *X* and *Y*. The only legal combinations allowed for a MAC instruction are: *mul.M1;add.A1* and *mul.M2;add.A2*.

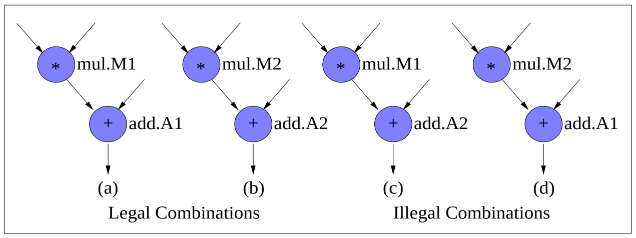

Figure 3: Possible Combinations for a MAC Instruction

Figure 3 shows the possible legal and illegal combinations of instructions that are allowed based on the connectivity of functional units to specific register files. Clustering instructions just on the basis of data-flow in an application results in spurious instruction combinations: *mul.M2;add.A1* and *mul.M1;add.A2*. Therefore, connectivity information is an important parameter in synthesizing valid ISEs for HCDSPs.

In a VLIW DSP architecture, the instructions, after dispatch may finish execution in varied time intervals. During the execution of an instruction, a functional unit reads the operands from register file(s), performs execution and writes the result into a register file. We call the difference between the finish time and the dispatch time of an instruction, the **execution latency** of the instruction. After waiting for a time equal to the execution latency of an instruction, the dependent instruction can start execution.

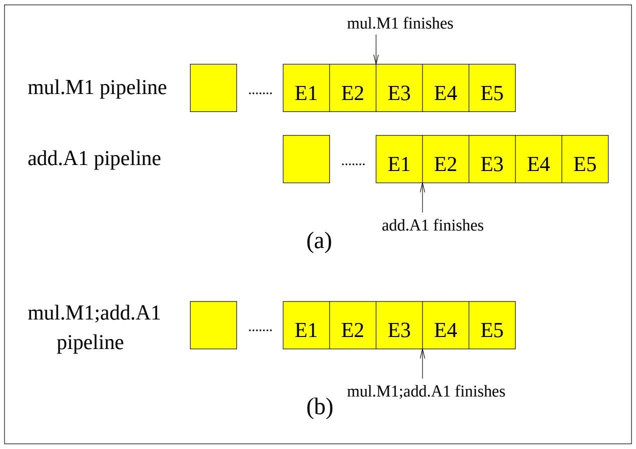

Figure 4: Instruction Pipeline for (a) *mul.M1* Instruction Followed by *add.A1* Instruction (b) Complex Instruction *mul.M1;add.A1*

In TMS320C6xx, the execution latencies of multiply and add instructions are 2 cycles and 1 cycle respectively. Applying these values to the sample architecture shown in Figure 2, we show the pipeline with a *mul.M1* instruction followed by a dependent *add.A1* instruction in Figure 4(a). The instruction *add.A1* begins execution after waiting 2 cycles for *mul.M1* to finish execution.

Figure 4 (b) shows the pipeline for *mul.M1;add.A1*, a complex instruction formed by combining *mul.M1* and *add.A1*. It is assumed here that the other instruction on which *add.A1* is dependent has been scheduled before *mul.M1* and there are no resource conflicts for dispatching *add.A1* along with *mul.M1*.

It is important to note that the execution of *mul.M1;add.A1* terminates exactly at the same point where *mul.M1* followed by *add.A1* finishes. This clearly shows that using *mul.M1;add.A1* compacts two instructions into one without affecting the performance. Therefore, we conclude here that clustering instructions into a complex instruction in a VLIW architecture does not affect performance.

4. ARCHITECTURAL MODEL

We propose a generalized model of the heterogeneous connectivity in an HCDSP architecture (shown in Figure 5).

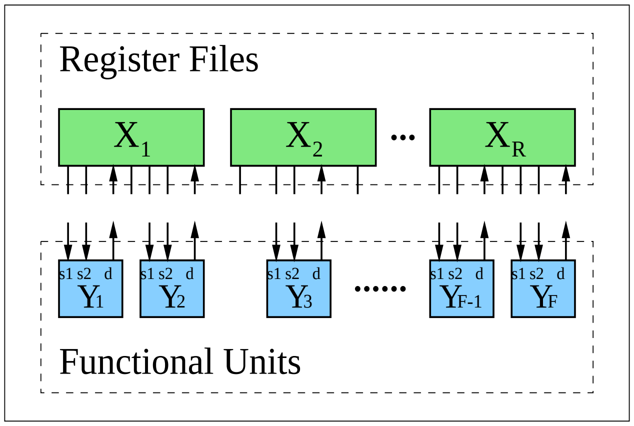

Figure 5: Heterogeneous Connectivity Model

The architecture has a set of R register files, $\mathcal{X} = \{X_1, X_2, ..., X_R\}$ and a set of F functional units, $\mathcal{Y} = \{Y_1, Y_2, ..., Y_F\}$, where F $\geq$ R. The heterogeneity is in the connection between the register files and the functional units.

The relation between a functional unit Y_i (1 $\leq$ i $\leq$ F) and a register file X_j (1 $\leq$ j $\leq$ R) can be represented as follows:

$Y_i \vdash X_j \leftarrow \{P_1, P_2, ..., P_m\}$ *operator* $\{Q_1, Q_2, ..., Q_n\}$
where,
$\vdash$ implies Y_i "writes into" X_j, *operator* defines the operation to be performed by the instruction, the first source operand is a register in $\{P_1, P_2, ..., P_m\} \subset \mathcal{X}$, the second source operand is a register in $\{Q_1, Q_2, ..., Q_n\} \subset \mathcal{X}$, and 1 $\leq$ m,n $\leq$ R. A base instruction run in unit Y_i is of the form, *mnemonic.*Y_i.

We define three functions that are used in deriving the connectivity constraints:

❒ **Writes(Y_i)**: returns the register file written by Y_i based on connectivity.

❒ **WrittenBy(X_j)**: returns a set of functional units that can write into the register file X_j.

❒ **Binds(Y_i)**: returns a set of operations bound to Y_i.

In order to legally combine two instructions, the following connectivity constraint must be obeyed.

An instruction i1.X can be combined with another instruction i2.Y dependent on the former through a source 's_i', where i = 1 or 2 and $X, Y \in \mathcal{Y}$, iff there exist paths from output 'd' of X to a register file and from the same register file to input 's_j' of Y, where j = 1 or 2. (The output port 'd' and the input ports 's_1' and 's_2' are shown in Figure 5.) We represent the resultant complex instruction as *i1.X;i2.Y*.

Our goal is to compact 3-operand instructions of the form I1: x = a op1 b and I2: y = x op2 c into a 4-operand instruction, y = (a op1 b) op2 c, where x and y are register operands; a, b and c can be register or immediate operands, and op1 and op2 are operators. The operations op1 and op2 to be executed by instructions I1 and I2 respectively, are bound to functional units Y_1 and $Y_2 \in \mathcal{Y}$. For the model presented in Figure 5, the connectivity constraint for operation in I2 is as follows:
$Y_2 \vdash \text{Writes}(Y_2) \leftarrow \{P_1, P_2, ..., P_m\} \text{ op2 } \{Q_1, Q_2, ..., Q_n\}$
It follows that the legal data flow across functional units for execution of I2 in Y_2 can be expressed as:
$Y_2 \leftarrow YY_1, YY_2$
where,

$$YY_1 = \bigcup_{i=1}^{m} WrittenBy(P_i)$$

$$YY_2 = \bigcup_{j=1}^{n} WrittenBy(Q_j)$$

In other words, the instruction I2 executing in functional unit Y_2 can legally get the first and the second source operands from the outputs of any of the functional units in YY_1 and YY_2 respectively. Thus instruction I2 can be coupled as a second instruction only with any instruction in Binds(YY_1) and Binds(YY_2) supplying first and second sources respectively for I2. This specifies the connectivity constraint for instruction I2. Similarly, we derive the connectivity constraints for all the other instructions.

In this paper, we use the TMS320C6xx architecture as an exemplar for illustrations and experiments. The TMS320C6xx is an 8-issue HCDSP with eight functional units (L1, L2, S1, S2, M1, M2, D1 and D2) and two register files A and B, each having 16 32-bit registers. This architecture fits into the HCDSP model with $\mathcal{Y}$ = {L1,L2,S1,S2,M1,M2,D1,D2} and $\mathcal{X}$ = {A,B}.

Architectural Assists

In order to apply our ISE synthesis approach, the generic HCDSP model defined above needs some architectural assists that are discussed below.

A complex instruction of the form B1;B2 (where B1 and B2 are base instructions) is processed through the pipeline as follows:

1. After decoding, the instruction B1;B2 residing in a VLIW instruction slot expands into two constituent instructions B1 and B2 (in decode or dispatch phase).
2. In dispatch phase, dispatcher issues B1 and B2 to respective functional units.
3. In execute phase, B2 starts execution after B1 has written its result.

This ensures that a synthesized complex instruction retains the performance achievable by the constituent base instructions (as shown in Figure 4).

The bit 0 in the TMS320C6xx instruction format (called the p-bit) determines whether the instruction executes in parallel with its next instruction. All instructions executing in parallel constitute an **execute packet**. A compiler targeting TMS320C6xx ensures that the execute packet is free of resource conflicts. When the instruction set is extended with complex instructions, the execute packet having one or more complex instruction(s) results in fewer than 8 instructions per packet because each complex instruction accounts for 2 instructions. With the aid of the p-bit, it is possible to have any number of instructions in the packet.

Each instruction in TMS320C6xx is 32-bit wide, in which 5 format-select bits are used for selecting one of 10 available formats. Using the format-select bits, it is possible to implement 32 different formats. Therefore, the bandwidth is sufficient to accommodate 22 additional formats. By accommodating the complex instructions in the unused format space, the decoder does not have to be entirely redesigned. Each operand field consumes 5 bits for accessing 32 registers (16 registers in either of the two register files). Therefore, with a p-bit, a 5-bit wide format-select field, and an opcode field varying in length depending on the instruction, we can allow only up to 4 operands in the 32-bit instruction format. This essentially means that our algorithm should look for opportunities to combine two instructions only, each having at the most 3 operands.

5. LATENCY CONSTRAINTS

While the connectivity constraints help prune illegal combinations, latency constraints are important for preserving the performance of the VLIW DSP architecture. Figure 6 shows a sequence of regular instructions which will be subject to the following dependency and latency considerations for valid combinations:

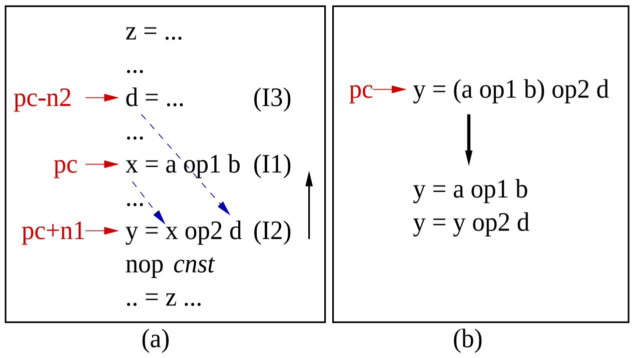

Figure 6: Instruction Sequence: (a) Candidate Pair {I1,I2} (b) Complex Instruction Generated from {I1,I2}

- ➤ *Primary Constraint*: Within a basic block, an instruction, I1 at *pc* and another dependent instruction, I2 at *(pc + n1)* can be combined to form a complex instruction if (1) I2 does not have a second source operand or the second source is dependent on an instruction, I3 at *(pc - n2)*, where n1,n2 > 0 and (2) I2 does not have any resource conflict with the execute packet at *pc*. The condition (1) is a dependency constraint and the condition (2) is a resource constraint. If conditions (1) and (2) are true, we call the instruction pair {I1,I2}, a *candidate pair*. All the subsequent considerations are latency constraints (Refer to Figure 6(a)).
- ➤ For two instructions, I1 and I2 to be combined, there must be an empty slot at *pc*. When combination takes place, the instruction I2 moves up to *pc*. If there is any other instruction residing at *(pc + n1)*, the latency constraints for all other instructions are maintained by default and this combination is allowed. Else, all the subsequent constraints must be imposed to meet the latency constraints and to ensure profitability:
- ➤ If (latency(I3) $\leq$ (n2 + latency(I1))), only then it is legal to combine I1 and I2 so that I2 gets the result of I3 well in time (Figure 6(a)).
- ➤ Figure 6(b) shows the execution semantics of the complex instruction obtained by combining instructions I1 and I2 shown in Figure 6(a). If the result x produced by I1 is also used in an instruction at *(pc + n3)* such that n3 > 0, then I1 cannot be combined with I2 because x does not exist in the complex instruction.
- ➤ In TMS320C6xx, a multi-cycle nop instruction, '*nop cnst*' can replace *cnst* number of *nop* instructions and save space for (*cnst*-1) instructions. When I2 moves up to *pc*, it leaves behind an empty slot. If all the other slots are empty, this might result in violation of latency constraints for other instructions. If a multi-cycle *nop* instruction is present in the vicinity (as shown in Figure 6(a)), it can be used to satisfy the latency constraint without consuming any extra space. Otherwise, the combination is not profitable. (For details, refer to [14].) If I1 can be profitably combined with I2, '*nop cnst*' can simply be converted into '*nop* (*cnst*+1)' without losing any code size for preserving the latency constraints.

6. ISE SYNTHESIS ALGORITHM

We first define a Restricted Data Dependence Graph (RDDG) as G^{RDDG}(N,E) where each node $\in$ N is an instruction and an edge $\in$ E exists between two nodes only if the pair of nodes satisfies the latency constraints. The RDDG is effectively a restricted subset of the Data Dependence Graph (DDG). Each data dependence edge between two instructions I1 and I2 in RDDG is potentially a complex instruction combining I1 and I2 through one of the sources of I2. Two data dependencies conflict when using one dependency as a complex instruction invalidates the possibility of the other becoming a complex instruction.

We now introduce the notion of Dependence Conflict Graph (DCG) which is the heart of our algorithm. A DCG is a graph $G^{DCG}(N^d,E^c)$ where N^d is a set of nodes representing data dependencies and E^c is a set of edges connecting two nodes with conflicting dependencies. Figure 7(b) illustrates a DCG generated from an RDDG shown in Figure 7(a).

Algorithm 1 *SynthesizeISE_DCGbased*()

```
Input: BasicBlock BB
  $CI \Leftarrow NULL$
  $G \Leftarrow CreateDCG(BB)$
  while $G \neq NULL$ do
    $n \Leftarrow SelectCandidateNode(G, CI)$
    $CI \Leftarrow CI \bigcup n$
    Delete $n$ from Graph $G$
    Delete nodes adjacent to $n$ from Graph $G$
  end while
```

A **greedy** solution to the problem would consider sequentially each unmarked instruction I in a basic block and test the possibility of legally and profitably combining I with each instruction $I1 \in DefUseChain(I)$ into a complex instruction. The test of legality and profitability is based on testing the connectivity and latency constraints (as discussed in Section 4 and Section 5). If the test evaluates to true, the greedy approach would immediately add the pair {I,I1} to a set representing generated complex instructions and mark the constituent base instructions. The consideration of only unmarked instructions for combination allows a valid replacement by the generated complex instruction. The order in which the greedy algorithm examines the instructions is determined by the sequence of instructions in the basic block, which may not be optimal. So, we propose a better Heuristic-based ISE Synthesis approach which employs the DCG representation to solve the problem. Hence, we call this approach a **DCG-based** approach.

Algorithm 1 (*SynthesizeISE_DCGbased* procedure) outlines the DCG-based approach, which has two main objectives: To maximize the reduction in code-size without affecting the performance and to minimize the number of new complex instructions generated. Finding the Maximum Independent Set (MIS) for G^{DCG} can yield the solution to the first objective - getting maximum code size reduction without hampering the performance. Unfortunately, MIS is known to be NP-complete [15]. So, any heuristic employed for getting a maximal solution should pay attention to the second objective i.e., minimizing the number of new instructions generated. Our DCG-based algorithm, shown as Algorithm 1 strives to meet both the objectives. The algorithm uses *SatisfyConnDepCons* procedure to determine if two instructions i and j can be legally combined based on the dependency and connectivity constraints (as explained in Section 4). It also uses *SatisfyLatConstraints* procedure for evaluating the latency constraints which ensures a profitable composition (as discussed in Section 5).

The algorithm, embedded in *SynthesizeISE_DCGbased* procedure starts by creating the DCG from the DDG using *CreateDCG* procedure (Algorithm 2). By checking whether the latency constraints are satisfied, this procedure effectively generates the DCG from the RDDG. The algorithm then calls *SelectCandidateNode* procedure (Algorithm 3), which selects a candidate node representing a profitable complex instruction to be added to a growing list called CI. The *SelectCandidateNode* procedure selects a node in DCG with minimum degree in order to maximize the chances of combination to form complex instruction. For instance, in Figure 7(b), selecting node 2 (with degree=4) as a complex

Algorithm 2 *CreateDCG*()

```
Input: BasicBlock $BB$
Returns: Graph $G(N, E)$
  $N \Leftarrow E \Leftarrow NULL$
  foreach $I \in$ Instructions[$BB$] do
    foreach $J \in$ DefUseChain[$I$] do
      if $SatisfyLatConstraints(I, J, dep[I, J])$ then
        {{I,J} represents a complex instruction}
        $ciNode \Leftarrow CreateNode(G, \{I, J\})$
        $Instr1[ciNode] \Leftarrow I$
        $Instr2[ciNode] \Leftarrow J$
        $sn[ciNode] \Leftarrow WhichSource(dep[I, J], J)$
        $N \Leftarrow N \bigcup ciNode$
      end if
    end for
  end for
  foreach $n \in N$ do
    $I_1 \Leftarrow Instr1[n]$
    $I_2 \Leftarrow Instr2[n]$
    foreach $I \in$ UseDefChain[$I_1$] $\bigcup$ UseDefChain[$I_2$] do
      if $Node[\{I, I_1\}] \in N$ then
        $e \Leftarrow CreateEdge(G, n, Node[\{I, I_1\}])$
      else if $Node[\{I, I_2\}] \in N$ then
        $e \Leftarrow CreateEdge(G, n, Node[\{I, I_2\}])$
      end if
      $E \Leftarrow E \bigcup e$
    end for
    foreach $I \in$ DefUseChain[$I_1$] $\bigcup$ DefUseChain[$I_2$] do
      if $Node[\{I_1, I\}] \in N$ then
        $e \Leftarrow CreateEdge(G, n, Node[\{I_1, I\}])$
      else if $Node[\{I_2, I\}] \in N$ then
        $e \Leftarrow CreateEdge(G, n, Node[\{I_2, I\}])$
      end if
      $E \Leftarrow E \bigcup e$
    end for
  end for
  return $G$
```

Algorithm 3 *SelectCandidateNode*()

```
Input: Graph $G$, Set $CI$
Returns: Node $n$
  $N \Leftarrow$ Set of nodes with minimum degree
  foreach $n \in N$ do
    if $\exists n_1 \in CI$ such that CmplxInstr[$n_1$] = CmplxInstr[$n$]
    then
      Frequency[CmplxInstr[$n$]] + +
      return $n$
    else
      if $SatisfyConnDepCons(Instr1[n], Instr2[n], sn[n])$
      then
        return $n$
      else
        return $NULL$
      end if
    end if
  end for
```

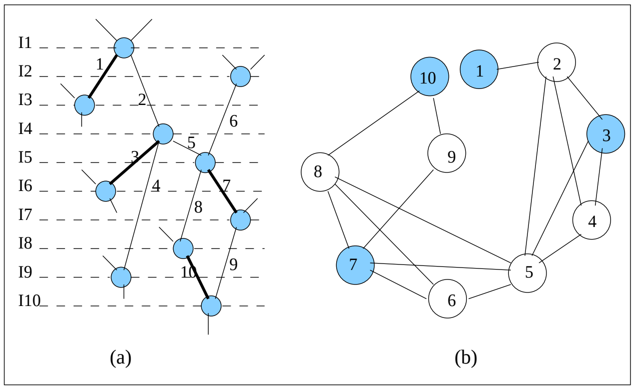

Figure 7: Running DCG-based Algorithm (a) Restricted Data Dependence Graph (b) Dependence Conflict Graph

instruction invalidates the possibilities of nodes 3, 4 and 5 to be considered as complex instructions, while choosing node 1 (having degree=2) jeopardizes the consideration of only node 2. The *SelectCandidateNode* also checks for the dependency and connectivity constraints to ensure a legal combination. When there are more than one nodes having the minimum degree, the heuristic breaks the tie by selecting the node having the complex instruction that has already been encountered before and increments its frequency. This guarantees maximum reuse of the selected complex instruction. The selected node and all its adjacent nodes are then deleted from the DCG. The process of selection and deletion is continued till the graph becomes empty. The list CI finally contains the nodes representing newly generated complex instructions with frequencies of their occurrence stored in *Frequency*. A possible output of the DCG-based algorithm is presented in Figure 7(a) as bold edges. The corresponding DCG is shown in Figure 7(b). The complex instructions are generated in the order: 1,3,10,7.

The running time of *SynthesizeISE_DCGbased()* is O(n^2), where n is the number of instructions. The algorithm is integrated with other complex phases of the compiler such as Instruction Selection and Scheduling. Therefore, the time taken to perform synthesis of new instructions is dominated by the time taken by the other, more complex phases.

7. METHODOLOGY

Figure 8 shows the flow of our framework. The target architecture is specified in terms of datapath connectivity and instruction set architecture (ISA) from which the instruction latencies and the connectivity model are derived. An input application is converted into an Intermediate Representation (IR) suitable for compiler optimizations. The IR is in Static Single Assignment (SSA) [16] form so that there are only Read-After-Write (RAW) dependencies. The instruction selection phase transforms each generic instruction into all possible target instructions. For example, an integer multiplication operation is mapped to MPY.M which encompasses two target instructions MPY.M1 and MPY.M2. The instruction scheduler schedules the target instructions to appropriate functional units based on the available resources. The scheduler ensures that the instructions in a candidate pair (as defined in Section 5) are separated by the latency

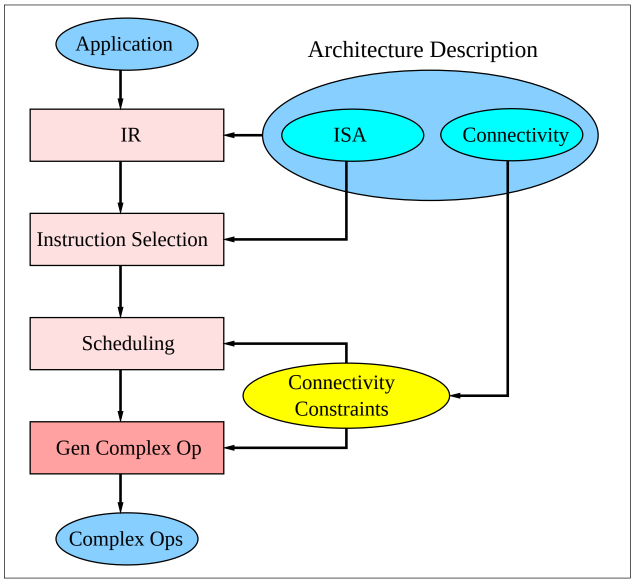

Figure 8: Experimental Results

Table 1: Experimental Results on MiBench Benchmarks

BMs	# BaseIns	Greedy		DCG-based	
		# Inst	% Impr	# Inst	% Impr
FFT	736	179	24	181	24
crc_32	150	35	23	36	24
adpcm	417	75	17	75	17
gsm	9855	2763	28	2806	28
pgp	40886	7750	18	8107	19
rijndael	3674	1378	37	1385	37
blowfish	3015	827	27	897	29
sha	410	109	26	110	26
mad	13230	3274	24	3369	25
gif2tiff	36983	8473	22	8814	23
jpeg	32827	8602	26	8756	25
susan	7232	2065	28	2217	30
qsort	247	49	19	49	19
bitcount	612	166	27	169	27
basicmath	994	171	17	172	17
sphinx	56097	12720	22	13170	23
rsynth	6318	1378	21	1408	22
stringsearch	997	274	27	282	28
dijkstra	292	75	25	76	26

distance dictated by the first instruction in the pair. This guarantees that the resource(s) used by complex instruction representing the candidate pair does not conflict with subsequent instructions. The *Gen Complex Op* phase (Figure 8) takes the scheduled code as input and generates new complex instructions, which are legal and profitable combinations of the base instructions.

8. EXPERIMENTAL RESULTS

We integrated our algorithm in the EXPRESSION [17] framework used for generating a retargetable compiler and simulator. We extracted from the EXPRESSION model of TI TMS320C6xx, the instruction latencies used for deriving latency constraints and the connectivity model used for deriving the connectivity constraints. For our purpose, we modified the retargetable compiler to explore opportunities for compacting instructions.

We conducted our experiments on MiBench [18] benchmarks from the University of Michigan. The results are collectively presented in Table 1 and Figure 9. The leftmost column in the table shows the benchmarks (*BMs*) in the order of different areas in Embedded applications: Telecommunications, Security, Consumer, Automotive and Industrial Control, Office Automation and Network applications. The next column shows the number of base instructions in memory (*# BaseIns*). The subsequent columns together present the efficacy of using ISEs (generated by greedy or DCG-based approach). The metrics used are the number of instances when a complex instruction replaces two base instructions (*# Inst*) and the percentage code size reduction (*% Impr*). The percentage code size reduction is calculated as follows:

$$\%Impr = (\#Inst/\#BaseIns)X100$$

On an average, the DCG-based algorithm achieves 25% reduction in code size, 1% more than that obtained by the greedy algorithm. Figure 9 shows that the DCG-based algorithm also results in fewer new complex instructions than the greedy algorithm. Thus, using the DCG-based algorithm, the base instruction set needs augmentation with fewer new instructions and at the same time the augmented instruction set achieves more code size reduction than that obtained using the greedy algorithm.

The ISE synthesis algorithm is restricted to finding complex instructions within a basic block. Therefore, the chances of combination are higher in applications having larger basic blocks, as is demonstrated in *rijndael* benchmark. The largest benchmark is *sphinx* having 56097 instructions from the base instruction set. For this application, the DCG-based algorithm synthesizes 274 new instructions, which when used as extensions to the instruction set yields 23% reduction in code size. The total number of new instructions synthesized for all the benchmarks was 523. Note that we can easily incorporate the newly generated complex instructions into the existing instruction set since there is sufficient unused encoding space for adding 22 more instruction formats, each accommodating up to 32 instructions totaling 704. Some of the examples of the generated complex instructions are *MPYDP.M1;ADDDP.L2*, *LDDW.D1;ADDSP.L1*, *AND.S2;ADD.S1* etc. With an average 25% improvement in code size as obtained by the heuristic-based algorithm, the memory usage of TMS320C6xx (illustrated by the tall bar for the TMS320C6xx in Figure 1(b)) becomes comparable with the other VLIW DSPs (i.e., same level as the other DSPs in Figure 1(b)).

The performance of a VLIW DSP is essentially attributed to the Instruction Scheduler of the compiler. The opportunities to find legal combinations of instructions are affected by the work done by the scheduler: If fewer VLIW slots are utilized by the Instruction Scheduler, then there are more opportunities for combination. This enables us to do a trade-off between the performance achievable by optimally scheduled instructions and the code-size reduction obtainable by exploiting the opportunities of using complex instructions. A complex instruction utilizes one register less than the number of registers used by the constituent base instructions. Consequently, there is an overall reduction in

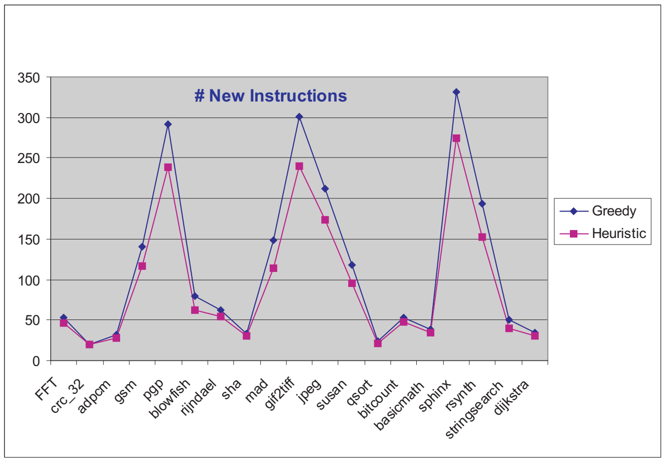

Figure 9: Comparison of the number of New Complex Instructions Generated

register pressure for every combination of base instructions. Therefore, it is likely that a spilled code can be freed of spilling just by efficiently combining instructions. As a result, the performance cannot degrade but at the best can increase.

One might argue that adding complex instructions to a regular instruction set of a VLIW machine can lead to increase in compiler complexity. In that context, it is important to note that the same algorithm for synthesizing complex instructions can be used to generate code for a VLIW machine having these complex instructions. This algorithm can be added as a back-end to a compiler targeting an HCDSP architecture like TMS320C6xx having a regular instruction set.

9. SUMMARY

We presented a Heuristic-based algorithm to synthesize Instruction Set Extensions (ISEs) that reduce code size for a Heterogeneous-Connectivity-based DSP (HCDSP) architecture like TMS320C6xx. By modeling the architecture in a retargetable framework, the connectivity and latency constraints were derived from the architecture description. Based on these constraints and the dependency constraints derived from the application, our framework was able to generate profitable complex instructions as extensions to the existing instruction set. The generated ISEs were able to achieve an average code size reduction of 25% without losing any performance. In order to apply our technique to HCDSPs, a generalized superset of a simple VLIW DSP, we augmented the generic connectivity model of HCDSP with a few architectural assists. We also assumed that it is possible to implement those architectural assists obeying timing and area constraints; future work will study this issue further and also quantify the performance improvements attainable using the generated ISEs.

10. ACKNOWLEDGMENTS

This research was partially funded by NSF grants CCR-0203813 and CCR-0205712.

11. REFERENCES

[1] http://www.ti.com. *TI TMS320C6xx user manual.*

[2] Silvina Hanono and Srinivas Devadas. Instruction Selection, Resource Allocation and Scheduling in the AVIV Retargetable Code Generator. *In Proc. of the Design Automation Conference (DAC), pages 510-515*, 1998.

[3] Thomas M. Conte, Sanjeev Banerjia, Sergei Y. Larin, Kishore N. Menezes and Sumedh W. Sathaye. Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings. *In Proc. 29th Int'l Symposium on Microarchitecture, pages 201-211*, 1996.

[4] Shail Aditya, Scott A. Mahlke and B. Ramakrishna Rau. Code Size Minimization and Retargetable Assembly for EPIC and VLIW Instruction Formats. *Technical Report, HP Labs PL-2000-141.*

[5] M. Kozuch and A. Wolfe. Compression of Embedded System Programs. *In Proc. of the Int'l Conference on Computer Design (ICCD), pages 270-277*, 1994.

[6] Stan Y. Liao, Srinivas Devadas and Kurt Keutzer. Code Density Optimization for Embedded DSP Processors using Data Compression Techniques. *IEEE Transactions on CAD, 17(7):601-608*, 1998.

[7] Huiyang Zhou and Thomas M. Conte. Code Size Efficiency in Global Scheduling for ILP Processors. *6th Annual Workshop on ICCA*, 2002.

[8] Jong-eun Lee, Kiyoung Choi and Nikil Dutt. Efficient Instruction Encoding for Automatic Instruction Set Design of Configurable ASIPs. *In Proc. of the Int'l Conference on Computer Aided Design (ICCAD)*, 2002.

[9] "Hoon Choi, Jong-Sun Kim, Chi-Won Yoon, In-Cheol Park, Seung Ho Hwang and Chong-Min Kyung. Synthesis of Application Specific Instructions for Embedded DSP Software. *IEEE Transactions on Computers*, 1999.

[10] Rainer Leupers and Peter Marwedel. Instruction Selection for Embedded DSPs with Complex Instructions. *In Proc. of the European Design Automation Conference (EURO-DAC)*, 1996.

[11] Marnix Arnold and Henk Corporaal. Instruction Set Synthesis Using Operation Pattern Detection. *5th Annual Conference of ASCI*, 1999.

[12] Fei Sun, Srivaths Ravi, Anand Raghunathan and Niraj K. Jha. Synthesis of Custom Processors Based on Extensible Platforms. *In Proc. of the Int'l Conference on Computer Aided Design (ICCAD)*, 2002.

[13] Kubilay Atasu, Laura Pozzi and Paolo Ienne. Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints. *In Proc. of the Design Automation Conference (DAC)*, 2003.

[14] Partha Biswas and Nikil Dutt. Greedy and Heuristic-based Algorithms for Synthesis of Complex Instructions in Heterogeneous-Connectivity-based DSPs. *UCI-ICS TR 03-16*, 2003.

[15] R.M. Karp. Reducibility Among Combinatorial Problems. *Complexity of Computer Computations*, Plenum Press, 1972.

[16] S.S. Muchnick. *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.

[17] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt and Alex Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. *In Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, 1999.

[18] Mathew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge and Richard B. Brown. MiBench: A Free Commercially Representative Embedded Benchmark Suite. *http://www.eecs.umich.edu/jringenb/mibench/.*