Greedy and Heuristic-based Algorithms for Synthesis of Complex Instructions in Heterogeneous-Connectivity-based DSPs

Partha Biswas partha@cecs.uci.edu
Nikil Dutt dutt@cecs.uci.edu

Architectures and Compilers for Embedded Systems (ACES) Laboratory
Center for Embedded Computer Systems
School of Information and Computer Science
University of California, Irvine, CA 92697-3425
http://www.cecs.uci.edu/~aces

CECS Technical Report #03-16
Dept. of Information and Computer Science
University of California, Irvine, CA 92697, USA

May 1, 2003

Abstract

VLIW DSP architectures can have 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 Greedy algorithm and a Heuristic-based algorithm to synthesize ISEs for a generalized heterogeneous-connecitivity-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 algorithms on the TI TMS320C6xx, a representative VLIW DSP.

Contents

List of Figures

List of Tables

1 Introduction

The embedded application domain is the fastest growing market segment in the microprocessor industry. An application-specific instruction-set processor (ASIP) is particularly suited for applications having common characteristics like embedded control or digital-signal processing. These applications demand good performance, low power and reduced code size. The datapath of an ASIP is optimized for an application class by addition of special functional units for frequently-used operations and elimination of infrequently-used units. Unlike ASICs, ASIPs retain the benefit of flexibility by being a programmable processor.

A digital signal processor (DSP) is a common class of ASIP designed to perform common operations on digital encodings of analog signals. The operations carried out are signal processing tasks and are generally math-intensive. In order to boost performance, a new class of DSPs are 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 two different algorithms to augment the instruction set of a VLIW DSP with complex instructions that can significantly reduce the code size. Both the algorithms guarantee that the performance realizable by the compiler is preserved. In the rest of the report, we will refer to the complex instructions, as **Instruction Set Extensions** or **ISE**s.

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 [10] 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 ISEs for special purpose DSPs. 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 report, we present a Greedy algorithm and a Heuristic-based algorithm that synthesize new complex instructions for extending the instruction set of HCDSPs with the goal of code size reduction. The Greedy algorithm is easy to implement while the Heuristic-based algorithm is more efficient in generating effective ISEs. Using the architecture, TMS320C6xx [10], we demonstrate the ability of our algorithms to achieve significant code size reduction (upto 37 %) over the base instruction set architecture through synthesized ISEs with no loss in performance.

The rest of the report is organized as follows: In Section 2, we discuss related research work. Section 3 presents the motivation for our work. Section 4 demostrates TMS320C6xx as a HCDSP and discusses our model of an HCDSP architecture. Section 5 describes the complete flow of our approach as well as the algorithms (greedy and heuristic-based) for synthesizing complex instructions. Section 6 shows the efficacy of our approach on the MiBench suite. Finally, Section 7 concludes the report.

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 ([15],[16],[12],[14],[13]) 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 ([17]) 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 [9] 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 [5] [3].

The ISEs were also used as specialized instructions in coprocessors, which are extensions to the main processor instruction set. The main goal in [8], [5], [18], [7] and [6] 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. A complex instruction in the context of a VLIW processor is an instruction occupying a VLIW instruction slot. If two parallel instructions are combined to form a complex instruction, the number of operands in the generated instruction is the sum of the number of operands in the constituent instructions. In that case, the bitwidth allocated for each instruction slot may not be sufficient to accommodate the new instruction. For example, in TMS320C6xx, combining two parallel 3-operand instructions results in a 6-operand instruction which is impractical to be represented in a 32-bit instruction format. A VLIW instruction in our model of TMS320C6xx consists of 8 32-bit instructions, each of which can be a base instruction or a complex instruction.

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 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

We now discuss the different factors that motivated our work.

3.1 Poor Code Density in TMS320C6xx

The TMS320C6xx was tuned for DSP applications by matching them with right kind of functional unit. In 0.25um technology, it was very practical to have several multipliers, adders and other functional units on the chip. The resulting architecture was an HCDSP architecture.

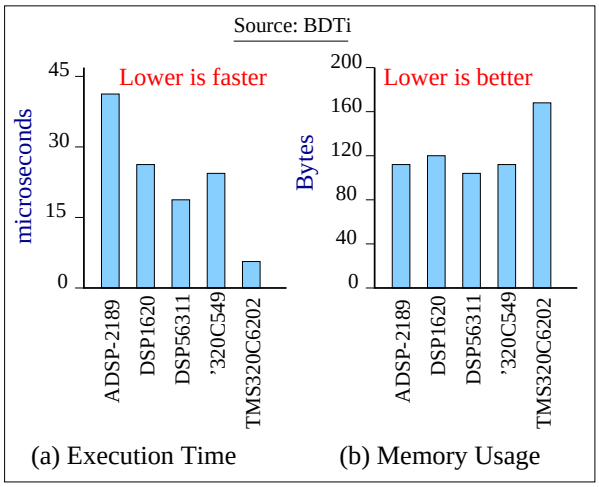

Figure 1. VLIW DSPs: (a) Execution Time on Complex Block FIR (b) Memory Usage on FSM Benchmark

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.

3.2 Legal Instruction Set Extensions

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 are: mul.M1;add.A1 and mul.M2;add.A2.

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

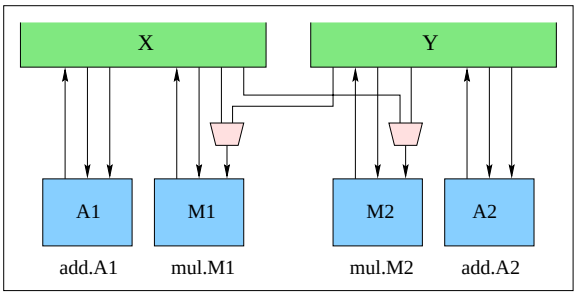

Figure 2. An example of HCDSP architecture

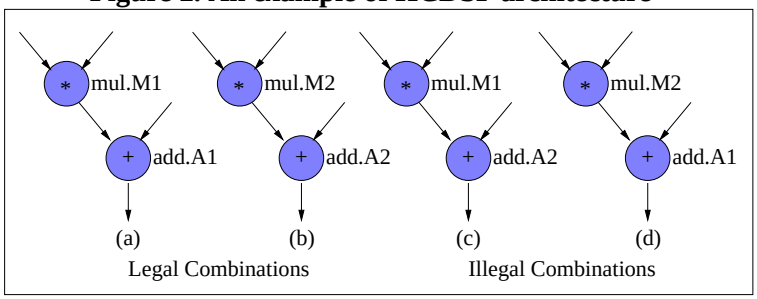

Figure 3. Possible combinations

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 sythesizing valid ISEs for HCDSPs.

3.3 Opportunities for Improvement in TMS320C6xx

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.

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 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.

When adding new compact instructions to the existing instruction set, it is also important to

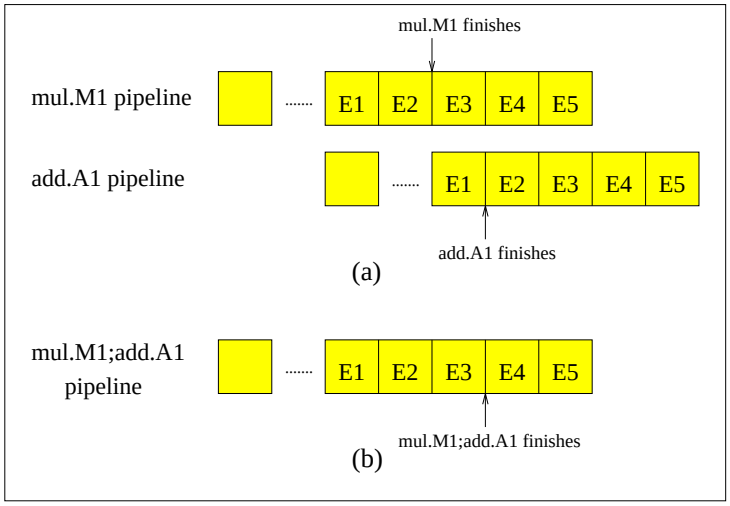

Figure 4. Instruction pipeline for (a) add.A1 Dependent on mul.M1 (b) Complex Instruction mul.M1;add.A1

consider the decoder complexity and the available bandwidth to support new instruction formats. Table 1 presents the existing instruction formats in TMS320C67x instruction set.

Table 1. Instruction formats in TMS320C67x

Units	Instruction Types	Bits				
		6	5	4	3	2
L	all	-	-	1	1	0
M	all	0	0	0	0	0
D	most	1	0	0	0	0
D	LD/ST w/ 15-bit off	-	-	-	1	1
D	LD/ST baseR + off	-	-	-	0	1
S	most	-	1	0	0	0
S	ADDK (16 bit cst)	1	0	1	0	0
S	Field ops(imm forms)	-	0	0	1	0
S	MVK, MVKH	-	1	0	1	0
S	Bcond, disp	0	0	1	0	0

Each instruction in TMS320C6xx is 32-bit wide, in which 5 format-select bits (bit positions 6, 5, 4, 3 and 2) are used for selecting one of 10 available formats. Entries marked '-' indicate some of the bit positions taken by opcode field. Using the format-select bits, it is possible to implement 32 different formats. Therefore, the bandwidth is sufficient to accomodate 22 additional formats. By accomodating the complex instructions in the unused format space, the decoder does not have to be entirely redesigned.

Enhancement of the instruction set of heterogeneous architectures will need careful considera-

tion of available resources and combinations of instructions allowed under non-trivial connectivity constraints. Because of code size improvement in a VLIW architecture without compromising performance, our work strives to have tremendous impact on embedded systems community.

4 HCDSP Architecture

With the background presented in Section 1 and Section 3, we present the TMS320C6xx architecture as an example of HCDSP in Section 4.1. Section 4.2 presents a generalized model of the heterogeneous connectivity in an HCDSP architecture. Our ISE synthesis approach requires some architectural assists from the HCDSP architecture, which are discussed in Section 4.3.

4.1 TMS320C6xx: An HCDSP

In this report, we use the TMS320C6xx architecture as an exemplar for illustrations and experiments. TMS320C6xx is an 8-issue VLIW machine with two register files A and B, each having 16 32-bit registers. The register file connectivity in the architecture is shown below:

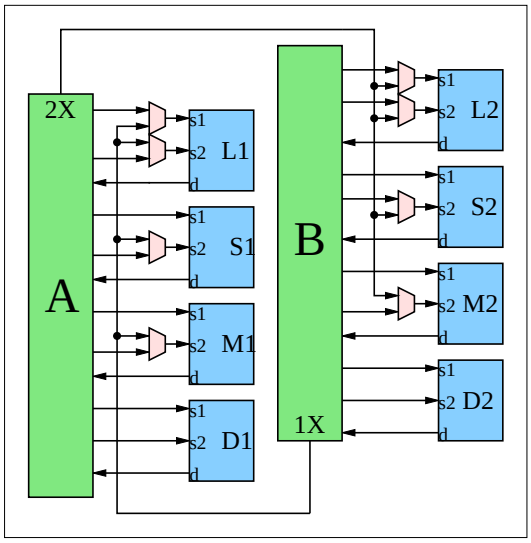

Figure 5. Heterogeneous Connectivity in TI TMS320C6xx Architecture

The eight functional units (L1, L2, S1, S2, M1, M2, D1 and D2) in the datapaths can be divided into two groups of four each. A functional unit, X1 in one group has an almost identical corresponding unit, X2 in the other, where X is L, S, M or D. Each unit is capable of running one instruction out of an 8-instruction VLIW packet. An instruction run in unit X1 is of the form, *mnemonic*.X1 and that in unit X2 has the form, *mnemonic*.X2.

There are 8 functional units available for executing 8 different instructions. Each VLIW execution packet to be fed to the functional units is decided statically by a compiler targeting TMS320C6xx. A packet consisting of instructions to be executed in parallel, should be free of resource conflicts.

4.2 HCDSP Architecture Model

We now propose a generalized model of the heterogeneous connectivity in an HCDSP architecture (shown in Figure 6).

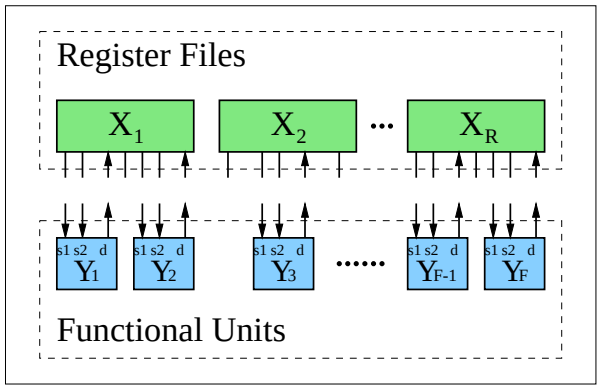

Figure 6. Heterogeneous Connectivity Model and Connectivity Constraint

The architecture has a set of R register files, $X = \{X_1, X_2, ..., X_R\}$ and a set of F functional units, $Y = \{Y_1, Y_2, ..., Y_F\}$, 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 $\{P_1, P_2, ..., P_m\} \subset X$, the second source operand is a register in $\{Q_1, Q_2, ..., Q_n\} \subset X$, 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 $\in Y$, 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 6.) We represent the resultant complex instruction as *i1.X;i2.Y*.

Using this formulation of the HCDSP architecture model, we show the derivation of connectivity constraints in Section 5.2.

4.3 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 another 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.

5 Our Approach

Our goal is to generate new legal ISEs which are frequently used by the application domain and contribute to the code size improvement. Our framework (Section 5.1) takes as inputs, an architecture description and an application, and generates ISEs that can improve the code density for the given application. We explain the derivation of connectivity constraints from the generalized HCDSP model (Section 4.2) in Section 5.2. Further important considerations for selection of complex instructions are discussed in Section 5.3.

5.1 Our Methodology

Figure 7 shows the flow of our framework.

The architecture description consists of datapath connectivity and instruction set architecture (ISA). The DSP architecture model, presented in Section 4.2 can be derived from the connectivity information. This involves evaluation of functions Binds(), Writes() and WrittenBy(). The derivation of connectivity constraints is later covered in the next section.

An input application is converted into an Intermediate Representation (IR) suitable for compiler optimizations. The IR is in Static Single Assignment (SSA) 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, 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 resources available.

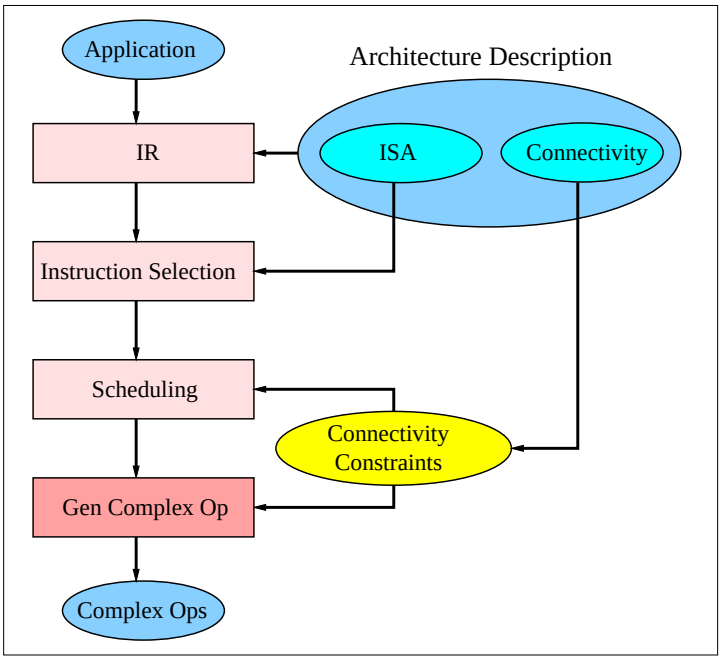

Figure 7. Our Framework

The *Gen Complex Op* phase (Figure 7) generates new complex instructions, which are legal and profitable combinations of the base instructions. All the characteristic applications are run and new instructions are added to a growing list. The selection of the final set of instructions to be added to the instruction set is based on maximum reuse of instructions and available instruction format bandwidth.

5.2 Derivation of Connectivity Constraint

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 $Y_2 \in Y$ (Refer to the HCDSP model presented in Section 4.2). For the model presented in Figure 6, the connectivity constraint for operation in I2 can be expressed as:
$Y_2 \vdash \text{Writes}(Y_2) \leftarrow \{P_1, P_2, ..., P_m\}$ op2 $\{Q_1, Q_2, ..., Q_n\}$
$\Rightarrow Y_2 \leftarrow YY_1, YY_2$
where

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

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

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 in-

struction 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. The TMS320C6xx architecture fits into the HCDSP model with
$Y = \{L1, L2, S1, S2, M1, M2, D1, D2\}$ and $X = \{A, B\}$.

5.3 Important Considerations for our Algorithms

One of the most important considerations for instruction set synthesis is the bit-width available for representation of complex instruction formats. In TMS320C6xx, each operand field consumes 5 bits for accessing 32 registers (16 registers in either of the two register files). Therefore, with a p-bit (discussed in Section 4.3, a 5-bit wide format-select field, and an opcode field varying in length depending on the instruction, we can allow only upto 4 operands in the instruction format. This essentially means that our algorithm should look for opportunities to combine two instructions only, each having at the most 3 operands.

While the connectivity constraints help prune illegal combinations, latency constraints are important for preserving the performance of the VLIW DSP architecture. Figure 8 shows a sequence of regular instructions which will be subject to the following dependency and latency considerations for valid combinations:

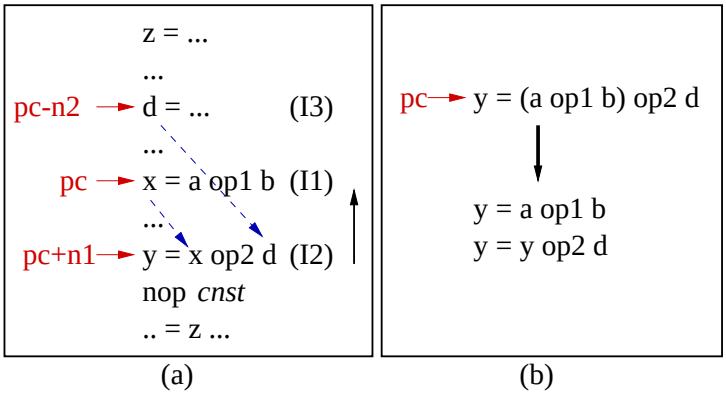

Figure 8. Instruction Sequence: (a) Candidate Pair {I1,I2} (b) Complex Instruction Generated from {I1,I2}

- *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. All the subsequent considerations are latency constraints (Figure 8(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 8(a)).
- Figure 8(b) shows the execution semantics of the complex instruction obtained by combining instructions I1 and I2 shown in Figure 8(a)). If the result x produced by I1 is 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, it can be used to satisfy the latency constraint without consuming any extra space. Otherwise, the combination is not profitable. The profitability of combination of instructions I1 and I2 in the presence of multi-cycle *nop* can be tested using the code shown in Algorithm 1. (Refer to Figure 8(a))

Algorithm 1 Combination in the presence of a multicycle *nop*

```
foreach i = (MAX_LATENCY − n1) downto 0 do
  L ⇐ Latency[Instruction[pc − i]]
  if L ≥ n1 then
    foreach j = (pc − i + 1)to(pc − i + L) do
      if Instruction[j] is a multi-cycle nop then
        {Combination is profitable}
      else
        {Combination is not profitable}
      end if
    end for
  end if
end for
```

In the above pseudo-code, MAX_LATENCY refers to the maximum of latencies of all the instructions in the instruction set. In Figure 8(a), 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.

The worst-case complexity of checking the dependency and the latency constraints for all the instructions is O($MAX\_LATENCY^2 * n$), where n is the total number of instructions.

5.4 ISE Synthesis Algorithms

We present a greedy algorithm and a heuristic-based algorithm with an objective of generating complex instructions that would result in maximum code size reduction. One of the most important constraints for the algorithm is the connectivity constraint that ensures generation of legal instructions. The other constraints are latency and dependency constraints that preserve the performance of the given instruction set architecture. The heuristic-based algorithm has an additional objective of minimizing the number of new complex instructions generated.

Algorithm 2 *Main*()

```
foreach Instruction I ∈ Instruction Set do
    DeriveConnectivityConstraints(I)
end for
foreach BasicBlock BB ∈ CFG do
    Synthesize(BB)
end for
```

The main procedure for the connectivity-aware generation of complex instructions is shown in Algorithm 2. We first derive the connectivity constraints for every instruction in the instruction set from the HCDSP model. This is done by the *DeriveConnectivityConstraints* procedure shown in Algorithm 3. For a given instruction I, this procedure determines a set of legal instructions that can generate results to be used as first and second source operands of I, designated as *LegalSourceInstructions[I,1]* and *LegalSourceInstructions[I,2]* respectively.

Algorithm 3 *DeriveConnectivityConstraints*()

$\{P_1, P_2, ..., P_m\} \Leftarrow GetSource1RegFiles(I)$
$\{Q_1, Q_2, ..., Q_m\} \Leftarrow GetSource2RegFiles(I)$
$YY_1 \Leftarrow \bigcup_{i=1}^{m} WrittenBy(P_i)$
$YY_2 \Leftarrow \bigcup_{j=1}^{n} WrittenBy(Q_j)$
$LegalSourceInstructions[I,1] \Leftarrow Binds(YY_1)$
$LegalSourceInstructions[I,2] \Leftarrow Binds(YY_2)$

The *SynthesizeISE* procedure then synthesizes ISEs at a basic-block-level granularity. When the greedy algorithm is used, the *SynthesizeISE* procedure is called *SynthesizeISE_Greedy* (Algorithm 5), while the heuristic-based *SynthesizeISE* procedure is called *SynthesizeISE_Heuristic* (Algorithm 6). Both algorithms use *SatisfyConnectivityConstraints* procedure (shown in Algorithm 4)to determine if two instructions i and j can be legally combined based on the connectivity constraints. The algorithms also use *SatisfyLatDepConstraints* procedure for evaluating the latency and connectivity constraints. This procedure encapsulates the considerations discussed in the previous section.

5.4.1 Greedy Algorithm

The rationale for the Greedy Algorithm is to quickly find ISEs that satisfy the connectivity and the latency constraints dictated by the architecture and the dependency constraints imposed by the

Algorithm 4 *SatisfyConnectivityConstraints*()

Input: Instruction I_1, Instruction I_2, Integer *srcNum*
Returns: *True*/*False*
  if $I_1 \in LegalSourceInstructions[I_2, srcNum]$ **then**
    return *True*
  else
    return *False*
  end if

application. This ensures that the performance remains unaltered when using complex instructions instead of the base instructions.

The pseudo-code for the Greedy Algorithm is presented in Algorithm 5. The algorithm implemented by *SynthesizeISE_Greedy* procedure progressively marches through each instruction I in each VLIW instruction (not shown in the figure) in the basic block, BB and explores greedily the possibilities of combining with all the instructions in the Definition-Use Chain of I. For each instruction pair {I,J} connected by a data dependency, the *WhichSource()* procedure finds the position of the source operand of J, which uses the result produced by I. Each possible complex instruction is immediately added to a growing list, CI if the constraints allow legal and profitable combination. The base instructions constituting the complex instruction are marked so that there is no overlap between the generated complex instructions. This ensures that the generated complex instructions can be potentially used in the given application without losing performance. At the end of the algorithm, we obtain a list of complex instructions accumulated in CI.

We define a **Restricted Data Dependence Graph (RDDG)**, G^{RDDG}(N,E) where each node $\in$ N is an instruction and an edge $\in$ E exists between two nodes in N only if the pair of nodes satisfies the latency and dependency constraints. The RDDG is actually a restricted subset of the Data Dependence Graph (DDG). Figure 9(b) shows the output of executing Greedy Algorithm on an RDDG shown in Figure 9(a). The bold edges in the final graph indicate the complex instructions chosen when all the nodes further satisfy the connectivity constraints.

In Figure 9(a), the instructions are labeled using alphabets a through j. In the solution presented in Figure 9(b), instruction a can be combined with either c or d. This choice can be made as a second pass by prioritizing the complex instructions based on their frequencies of occurrence. In this example, the number of complex instructions that can contribute to code size reduction is 3 - corresponding to the edges 1 or 2, 6 and 9. This is not an optimal solution because marking all the possible combinations originating from each instruction jeopardizes other likely combinations. For instance, edge 1 is preferrable to edge 2 for achieving maximum code size reduction because choosing edge 2 discards the possibilities of choosing edges 1, 3, 4 and 5, while choosing edge 1 discards only edge 2. The Heuristic-based algorithm circumvents this problem by employing a heuristic to maximize the code size reduction.

5.4.2 Heuristic-Based Algorithm

The Heuristic-based algorithm has two main objectives: To maximize the reduction in code-size without affecting the performance and to minimize the number of new complex instructions gen-

Algorithm 5 *SynthesizeISE_Greedy*()

Input: BasicBlock *BB*

```
foreach Instruction I ∈ Instructions[BB] do
    Mark[I] ⇐ 0
end for
CI ⇐ NULL
foreach Instruction I ∈ Instructions[BB] do
    if Mark[I] == 1 then
        continue
    end if
    foreach Instruction J ∈ DefUseChain[I] do
        if SatisfyLatDepConstraints(I, J, dep[I, J]) then
            {Whether result of I feeds source1 or source2 of J}
            srcNum ⇐ WhichSource(dep[I, J], J)
            if SatisfyConnectivityConstraints(I, J, srcNum) then
                {{I,J} is a complex instruction}
                CI ⇐ CI ⋃ {I, J}
                Mark[J] ⇐ 1
            end if
        end if
    end for
end for
```

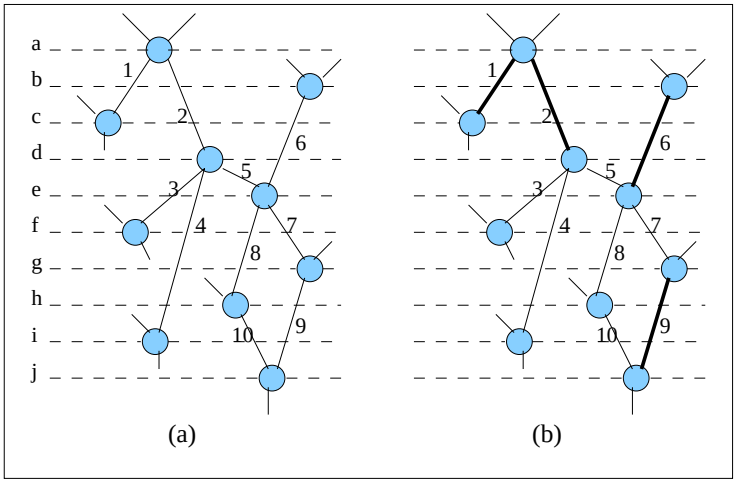

Figure 9. An Example (a) Restricted Data Dependence Graph (b) Output of the Greedy Algorithm

erated. Before getting into the details of the algorithm, we introduce a notion of **Dependence Conflict Graph (DCG)**. Each data dependence edge between two instructions I1 and I2 in RDDG represents a complex instruction combining I1 and I2 through one of the sources of I2. We say two data dependencies conflict when using one dependency as a complex instruction invalidates the possibility of the other becoming a complex instruction. 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.

Algorithm 6 *SynthesizeISE_Heuristic*()

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

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 [19]. So, any heuristic employed for getting a maximal solution should pay heed to the second objective i.e., minimizing the number of new instructions generated. Our Heuristic-based algorithm, shown as Algorithm 6 strives to meet both the objectives. The algorithm uses *SatisfyConnectivityConstraints* procedure (Algorithm 4) to determine if two instructions I and J can be legally combined based on the connectivity constraints. It also uses *SatisfyLatDepConstraints* procedure for evaluating the latency and dependency constraints which ensures a profitable composition (as discussed in Section 5.3).

The algorithm, embedded in *SynthesizeISE_Heuristic* procedure starts by creating the DCG from the DDG using *CreateDCG* procedure (Algorithm 7). By checking whether the latency and the dependency constraints are satisfied, this procedure effectively generates the DCG from the RDDG. The algorithm then calls *SelectCandidateNode* procedure (Algorithm 8), 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 and also checks for the 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

Algorithm 7 $CreateDCG()$

Input: BasicBlock BB
Returns: Graph $G(N,E)$

```
N ⇐ E ⇐ NULL
foreach I ∈ Instructions[BB] do
  foreach J ∈ DefUseChain[I] do
    if SatisfyLatDepConstraints(I, J, dep[I, J]) then
      {{I,J} represents a complex instruction}
      ciNode ⇐ CreateNode(G, {I, J})
      Instr1[ciNode] ⇐ I
      Instr2[ciNode] ⇐ J
      sn[ciNode] ⇐ WhichSource(dep[I, J], J)
      N ⇐ N ⋃ ciNode
    end if
  end for
end for
foreach Noden ∈ N do
  I1 ⇐ Instr1[n]
  I2 ⇐ Instr2[n]
  foreach I ∈ UseDefChain[I1] ⋃ UseDefChain[I2] do
    if Node[{I, I1}] ∈ N then
      e ⇐ CreateEdge(G, n, Node[{I, I1}])
    else if Node[{I, I2}]isinN then
      e ⇐ CreateEdge(G, n, Node[{I, I2}])
    end if
    E ⇐ E ⋃ e
  end for
  foreach I ∈ DefUseChain[I1] ⋃ DefUseChain[I2] do
    if Node[{I1, I}] ∈ N then
      e ⇐ CreateEdge(G, n, Node[{I1, I}])
    else if Node[{I2, I}]isinN then
      e ⇐ CreateEdge(G, n, Node[{I2, I}])
    end if
    E ⇐ E ⋃ e
  end for
end for
return G
```

Algorithm 8 *SelectCandidateNode*()

Input: Graph G, Set CI
Returns: Node n

```
N ⇐ Set of nodes with minimum degree
foreach n ∈ N do
  if ∃n1 ∈ CI such that CmplxInstr[n1] = CmplxInstr[n] then
    Frequency[CmplxInstr[n]]++
    return n
  else
    if SatisfyConnCons(Instr1[n], Instr2[n], sn[n]) then
      return n
    else
      return NULL
    end if
  end if
end for
```

stored in *Frequency*. The DCG generated from Figure 9(a) is shown in Figure 10(a). A possible output of the Heuristic-based algorithm is presented in Figure 10(b) as bold edges. The complex instructions are generated in the order: 1,3,6,9. This algorithm is better than the Greedy algorithm and is able to generate 4 complex instructions for the same application.

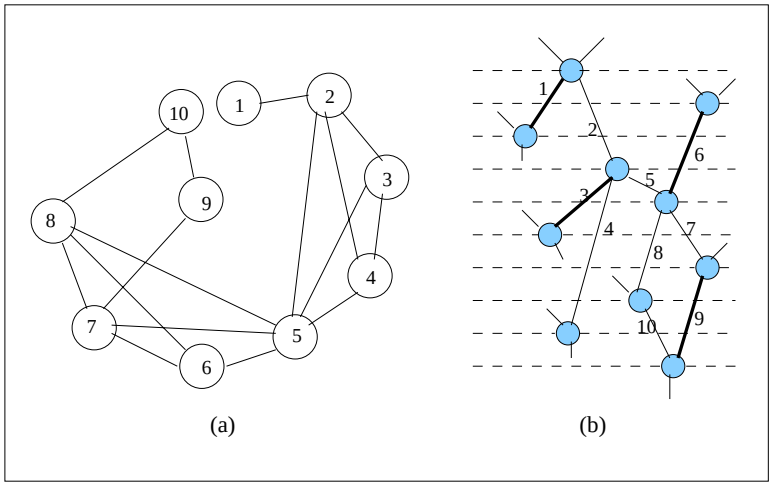

Figure 10. Running Heuristic-based Algorithm (a) DCG (b) Output of the Heuristic-based Algorithm

5.4.3 Complexity Analysis

Let us first analyze the algorithms when the connectivity constraints are not checked. The greedy algorithm is a fairly simple algorithm that does a quick pass of generating complex instructions in

O(n) time, where n is the total number of instructions in the application. The algorithm generates all possible complex instructions and leaves the user to select a desired set of complex instructions based on the maximum bit-width available for instruction representation.

The running time of the Heuristic-based algorithm is linear in terms of the number of dependency edges in the RDDG. Each node representing a base instruction in RDDG has two incident edges for the two source operands. So, the RDDG for a given application has 2*n number of edges. Consequently, the time complexity of the Heuristic-based algorithm is also O(n). In addition to being a practical algorithm for generating complex instructions, the heuristic used for minimizing the number of new instructions generated, presents an automated way to synthesize new ISEs. In case of tight bit-width constraints, the user can ponder over the *Frequency* of each complex instruction to consider addition to the current instruction set.

The *SatisfyConnectivityConstraints* procedure takes two instructions I1 and I2 and checks if I1 is present in the *LegalSourceInstructions* list of I2. The worst-case complexity of this procedure is O(n). So, both the algorithms when checking for connectivity constraints have the worst-case complexity, O(n^2). The greedy algorithm is easier to implement, while the heuristic-based algorithm is more effective in generating minimum number of new ISEs and at the same time leading to lesser code size.

Both algorithms are integrated with other complex phases of the compiler like Instruction Selection and Scheduling. So, the time taken to perform synthesis of new instructions is dominated by the time taken by the complex phases.

6 Experimental Results

We conducted our experiments on MiBench [11] benchmarks from University of Michigan. We implemented the flow presented in Section 5.1 on EXPRESSION [1] framework used for generating retargetable compiler and simulator. For our purpose, we modified the compiler targeted for TI TMS320C6xx architecture to explore opportunities for compacting instructions.

6.1 Code Size Reductions

The results of our experiments are presented in Table 2. The leftmost column shows the benchmarks 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. From the table, it is evident that the heuristic-based algorithm finds more opportunities than the greedy algorithm for combining instructions legally to form complex instructions. On an average, the heuristic-based algorithm achieves 25 % reduction in code size, 1 % more than that obtained by the greedy algorithm.

Both the algorithms are 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 efficacy of using an extended instruction set (generated by greedy or heuristic-based algorithm) over the base instruction set is depicted in Figure 11. The extended instruction set generated by the heuristic-based algorithm proves to be better than the greedy algorithm in terms of code size reduction. The heuristic-based algorithm also results in lesser number of new complex instructions than the greedy algorithm. (as shown in Figure 12)

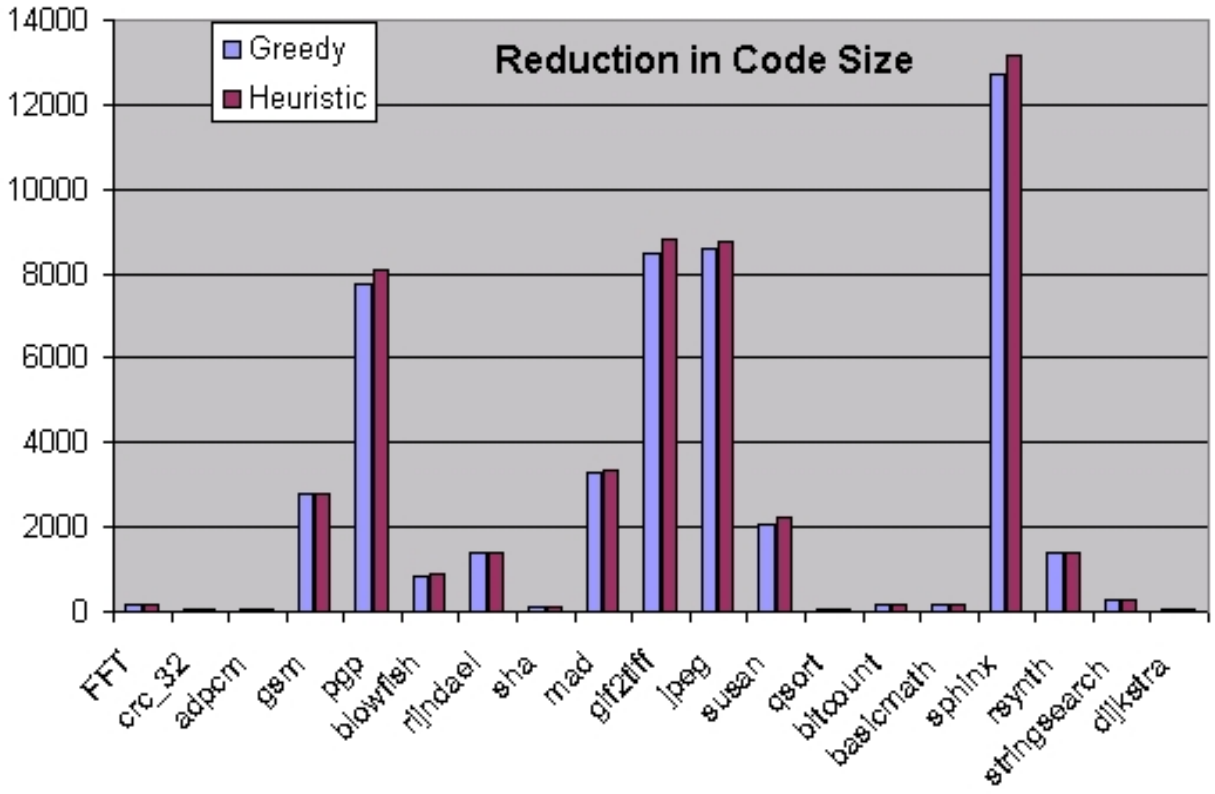

Figure 11. Comparison of Code Size Reduction on Mibench Benchmarks

Figure 12. # of New Complex Instructions

Table 2. Experimental Results on mibench Benchmarks

BM	# Base Ins	Greedy		Heuristic	
		# Oppor.	% Impr.	# Oppor.	% 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

Thus, using the heuristic-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 largest benchmark is *sphinx* having 56097 instructions from the base instruction set. For this application, the greedy algorithm synthesizes 332 new instructions, which when used as extensions to the instruction set yields 22 % reduction in code size. The heuristic algorithm for the same application brings 23 % code size reduction with only 274 new instructions. Given that there is bandwidth available for adding 22 more instruction formats, each accommodating upto 32 instructions, we can easily incorporate the newly generated complex instructions into the existing instruction set.

6.2 Impact of Our Work

With an average 25 % improvement in code size (as obtained by the heuristic-based algorithm), the memory usage of TMS320C6xx (as shown in Figure 13) becomes comparable with the other VLIW DSPs.

The performance of a VLIW DSP is essentially attributed to the Instruction Scheduler of the

compiler. The opportunity to find legal combinations of instructions is 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 scheduling instructions and the code-size reduction obtainable by exploiting the opportunities of using complex instructions. Our retargetable framework also allows us to vary the number of parallel resources and enforce any kind of heterogeneous connectivity constraints. The work presented in this report is extensible to any architecture that exhibits heterogeneous connectivity for VLIW-style DSPs (HCDSPs).

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 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.

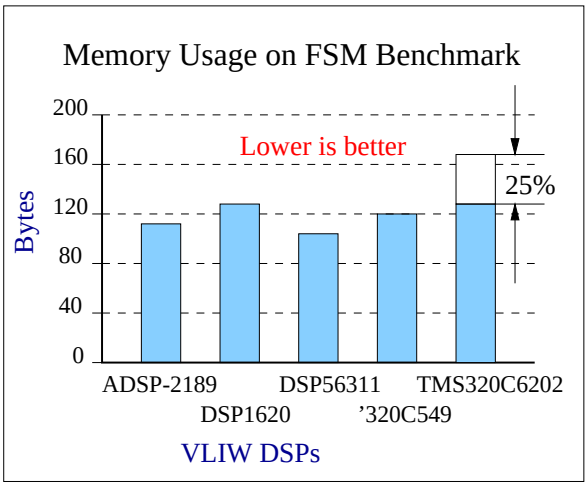

Figure 13. Comparison of Memory Usage with new TMS320C6xx Instruction Set Extensions

We gained a decrease in code size at the expense of slightly increasing the complexity of the dispatch unit. The dispatcher with the help of the format-specifier bits will identify a complex instruction and divide it into two regular instructions. The regular instructions will then be dispatched one after another. For example, MUL.M1.ADD.S1 will be broken into MUL.M1 and ADD.S1 and respectively dispatched into units M1 and S1 one after the other. Because of a dependency between MUL.M1 and ADD.S1, ADD.S1 should begin execution in S1 after MUL.M1 has finished writing the result.

One might argue that adding complex instructions to a regular instruction set of a VLIW machine can lead to increase in compiler complexity. It is important to note that the same (greedy or heuristic-based) 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 which normally targets an HCDSP architecture such as the TMS320C6xx with a regular instruction set.

7 Summary and Future Directions

We presented a Greedy and 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. On an average, the heuristic-based algorithm achieved 25 % reduction in code size, 1 % more than that obtained by the greedy algorithm. The heuristic-based algorithm also resulted in lesser number of new complex instructions than the greedy algorithm. Neither of the algorithms caused any degradation in 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.

8 Acknowledgments

This work was supported in part by grants from NSF (CCR 0203813 and CCR 0205712). We thank Jong-eun Lee and other members of the ACES laboratory (http://www.ics.uci.edu/ aces) for their input.

References

[1] A. Halambi et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. *DATE*, 1999.

[2] R. Leupers et al. Instruction Set Extraction from Programmable Structures. *ACM*, 1994.

[3] R. Leupers et al. Instruction Selection for Embedded DSPs with Complex Instructions. *EURO-DAC with EURO-VHDL*, 1996.

[4] A.K. Verma et al. Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints. *In proceedings of the 1st Workshop on Application Specific Processors, Istanbul*, November 2002.

[5] H. Choi et al. Synthesis of Application Specific Instructions for Embedded DSP Software. *IEEE Transactions on Computers*, 1999.

[6] A.K. Verma et al. Optimal Algorithm for Determining Speedup-Driven Instruction-Set Extensions under Register-File Port Constraints. *CASES*, 2002.

[7] L. Pozzi et al. Automatic Topology-Based Identification of Instruction-Set Extensions for Embedded Processors. *Technical Report CS 01/377, Swiss Federal Institute of Technology Lausanne, Switzerland*, 2001.

[8] M. Arnold et al. Instruction Set Synthesis Using Operation Pattern Detection. *5th Annual Conference of ASCI*, 1999.

[9] J. Lee et al. Efficient Instruction Encoding for Automatic Instruction Set Design of Configurable ASIPs. *ICCAD*, 2002.

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

[11] M. Guthaus et al. MiBench: A Free Commercially Representative Embedded Benchmark Suite. *http://www.eecs.umich.edu/jringenb/mibench/*.

[12] S. Aditya et al. Code Size Minimization and Retargetable Assembly for EPIC and VLIW Instruction Formats. *Technical Report, HPL PL-2000-141*.

[13] S.Y. Liao et al. Code Density Optimization for Embedded DSP Processors using Data Compression Techniques. *IEEE TCAD, 17(7):601-608*, 1998.

[14] M. Kozuch et al. Compression of Embedded System Programs. *IEEE ICCD, pages 270-277*, 1994.

[15] S. Hanono et al. Instruction Selection, Resource Allocation and Scheduling in the AVIV Retargetable Code Generator. *35th DAC, pages 510-515*, 1998.

[16] T.M. Conte et al. Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings. *29th MICRO, pages 201-211*, 1996.

[17] H. Zhou et al. Code Size Efficiency in Global Scheduling for ILP Processors. *6th Annual Workshop on ICCA*, 2002.

[18] F. Sun et al. Synthesis of Custom Processors Based on Extensible Platforms. *ICCAD*, 2002.

[19] M.R. Garey et al. Computers and Intractability: A Guide to the Theory of NP-Completeness. *New York: W.H. Freeman*, 1983.