Accepted for publication in IEEE Trans. Computer Aided Design (TCAD), 2000

Compact and Efficient Code Generation through Program Restructuring on Limited Memory Embedded System DSPs

Siddharth Rele* Vipin Jain* Santosh Pande*† J. Ramanujam‡

Abstract

Many embedded systems such as digital cameras, digital radios, high resolution printers, cellular phones, etc. involve heavy use of signal processing and are thus based on Digital Signal Processors (DSPs). DSPs such as the TMS320C2x and the DSP5600x have irregular data-paths that typically result due to application specific needs (such as chaining multiply-accumulate operations, etc). Efficient code generation for such embedded DSP processors is a challenging problem. The stringent requirements such as tight memory constraints and fast response time result in the need for compact and efficient code.

In this work, we address the problem of generating compact and efficient code for embedded DSP processors. Most of the DSP instruction set architectures (ISAs) feature *intra instruction parallelism (IIP)* enabling individual operations to be executed in parallel by generating a complex instruction. A reduction in generated code-size and improved performance can be achieved by exploiting this parallelism present in such ISAs. In this work, we present a code restructuring technique to fully exploit this parallelism through maximal utilization of the complex instructions present in the instruction set. We formulate this problem as a maximal benefit code restructuring problem, which is to derive the arrangement of statements to maximally exploit IIP without violating data dependencies. This problem is equivalent to Precedence Constrained Hamiltonian Path Problem for DAGs and the Traveling Salesman Problem (PTSP) in general, both of which are NP-hard. In this work, we present an optimal algorithm to solve the problem. We have implemented this optimal algorithm in the SPAM compiler targeted to generate code for the TMS320C25 DSP. We tested our framework on a number of benchmarks and found that the performance of the generated code (measured in dynamic instruction cycle counts) improves by as much as 9.9% with an average of 4%. The average code size reduction over code compiled without exploiting parallelism is 2.9%. We also studied the effect of loop unrolling on the available IIP. On-chip instruction cache can be effectively utilized by unrolling loops such that generated code fully occupies the memory. The benefit is reduction in dynamic instruction count due to higher number of complex instructions generated. We found that by unrolling loop by 4-5 times to fit available on-chip instruction cache, the dynamic instruction counts reduce by as much as 9.9%.

*Compiler Research Laboratory, PO Box 210030, Department of Electrical & Computer Engineering and Computer Science, University of Cincinnati, Cincinnati, OH 45221-0030. E-mail: {srele,vjain,santosh}@ececs.uc.edu. Phone: (513) 556-4785, Fax: (513) 556-7326.

†Corresponding author.

‡Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. E-mail: jxr@ee.lsu.edu. Phone: 225-388-5628; Fax: 225-388-5200.

1 Introduction

Embedded processors are widely used in a variety of applications such as cellular phones, pagers, printers, copiers, digital cameras, automobiles, flight navigation systems etc. Unlike general purpose processors, embedded processors are designed and optimized for specific (classes of) applications [13]. Embedded systems are constrained by limited on-chip program memory [2], real time performance requirements [18, 27] and low power consumption demands. The evaluation criteria for embedded processors are different from those of general purpose processors. The following criteria are typically used while comparing embedded processors [35].

Performance: The cost-performance ratio of embedded systems is measured in MIPS/dollar. It is one of the most important criteria for judging embedded processors due to the desired real time constraints and low costs of the system in which the processor is embedded.

Code size vs. density: The code size targeted towards *Complex Instruction Set Computing*(CISC) architectures can be smaller than the one that is targeted towards *Reduced Instruction Set Computing*(RISC) architectures due to the presence of complex instructions. However, in the absence of aggressive compiler optimizations, the code density of CISCs is poor compared to RISCs. Thus compilers have to play a big role in improving the code density by performing machine dependent optimizations specifically designed for reducing the size of the generated code through code restructuring transformations.

Traditionally, embedded processors and systems are programmed using assembly language, in order to meet the hard performance constraints and limited program memory. But, programming large complex applications in assembly language is tedious, error-prone, and time-consuming; in addition, such programs are difficult to maintain. High-level languages like C and C++ are replacing assembly language in embedded programming. Programming in high-level language simplifies the software development cycle. It can however incur a code size and performance penalty due to the semantic gap between the high level languages and the underlying embedded instruction set architecture. DSPs and embedded micro-processors provide specialized architectural features such as address generation units and specialized instructions that open opportunities for novel machine-dependent optimizations that enable both fast execution and compact code generation. Thus, the role of compiler to perform machine dependent optimizations is significant.

Traditional compiler optimizations have mostly focussed on improving code speed rather than code density or code size [10]. Although some optimizations can improve both code size and speed (e.g., common sub-expression elimination and dead-code elimination), several optimizations like loop unrolling, procedure in-lining etc., result in a speed-size tradeoff. Due to tight constraints on memory and real-time response, both code size and speed are very important for the application specific instruction processors (ASIPs) [8]. ASIPs offer new challenges to optimizing compilers and new approaches and paradigms are needed to generate efficient and compact code. DSPs form an important category of embedded ASIPs. Traditional compilers do not provide support for the specialized features of the DSPs by means of machine-dependent optimizations to reduce code size and improve code efficiency. Thus, new techniques must be developed for these systems.

```
a = b + c * d

x = y + z * k
```

Figure 1: Computational Dependency

1.1 VLIW vs. ASIP DSPs

Issues of instruction level parallelism (ILP) and code density are addressed for VLIW processors. In VLIW processors, pipelining allows to exploit EPIC style of parallelism but techniques developed for VLIW compilation may not be directly applicable to ASIP DSPs. In VLIW compilation, two completely independent operations are grouped together, while in the case of ASIP DSPs, the **computational dependencies** and data path constraints may not allow such arbitrary grouping. This is due to significant differences in ASIP DSPs and VLIW processors. The greater flexibility in VLIW architectures makes them easier targets for HLL compilers. Superscalar DSPs are capable of exploiting high levels of parallelism [26]. However as the parallelism increases, and the number of functional units and registers increase to support the complexity and cost of their control units also increases. A major impediment to using VLIW architectures in cost-sensitive embedded systems has been their high instruction bandwidth requirements. VLIW architecture has the ability to exploit more instances of parellelism as compared to the traditional architecture, but has low instruction density. Instruction density is one of the important issues for embedded DSPs. VLIW compilers allow to exploit ILP, however code density is secondary, whereas in DSPs, one must improve both **code density** and **execution efficiency**. Thus one must be able to generate as many complex instructions as possible. This is the topic of the paper.

Please refer to the code in Figure 1. Consider targeting the code shown in Figure 1 towards `TI TMS320C2X`[1]. Refer to Figure 3, which shows the TMS320C2X architecture. This DSP provides certain complex instructions such as `MPYA`, which are capable of doing a multiplication and addition at the same time. For the `MPYA` instruction, one of the source operands for the *addition operation* has to be in the `accumulator` while the other in the `P register` and for the *multiplication operation* one of the source operand is in the `T register` while the other is a part of the instruction(*i.e. loaded immediately*). In the above example, the multiplication `c * d` is carried out first and the result is available in the `P register`. The accumulator is loaded with `b` followed by loading of `z` in the `T register`. Since all the operands for the `MPYA` instruction are in the proper places, a compiler can generate a complex instruction replacing two instructions (one addition and other multiplication). In most of the cases, the operands are not in proper registers due to the ordering of the statements. Thus complex instruction exploiting parallelism can only be generated if code is restructured so that operands are in respective registers as explained above. Such issues are due to ASIP datapaths and are not faced in VLIW processors.

1.2 Loop Un-rolling

An important use of on-chip program memory is to to hold the instructions from short loops along with the associated data for ultra-fast computation. Such schemes are very attractive on

[1]Please refer to section 2 for a detailed discussion of this DSP architecture.

```
void main(){
for( i = 0; i < 10; i ++){
    a[i] = b[i] * c[i] + d[i];
  }
}
               (a)

void main(){
for( i = 0; i < 10; i = i + 2){
    a[i] = b[i] * c[i] + d[i];
    a[i+1] = b[i+1] * c[i+1] + d[i+1];
  }
}
               (b)
```

Figure 2: Loop unrolling

embedded processors where due to the presence of dedicated hard-ware on-chip (such as very fast multipliers-shifters etc.) and extremely fast accesses to on-chip data, the computation time of such loops is *extremely* small meeting almost all real-time demands. This is a very important goal in real time computing.

In this work, we carefully unroll short loops which allows cross-iteration code restructuring facilitating the complex instruction generation. Our code restructuring pass undertakes unrolling of small DSP loops to utilize the available on-chip program memory and to perform cross iteration code restructuring to generate more complex instructions. Complex instructions are formed by accumulation of a previous product and multiplication or a load in `T register` being carried out in parallel. For the code shown in Figure 2(a), the accumulation cannot be carried out with the multiplication and hence no complex instructions can be generated. If this loop is unrolled as shown in Figure 2(b), then there is an opportunity to generate a complex instruction by accumulation of `b[i] * c[i]` and `d[i]` in parallel with multiplication of `b[i+1]` and `c[i+1]`. The net result is much lesser dynamic instruction count and faster code execution improving real time response.

This paper is organized as follows: In Section 2 we discuss the intra-instruction level parallelism present in the TMS320C25 data paths. Section 4 describes our code restructuring framework for embedded DSP processors in general and Section 5 presents our cost benefit model. In Section 6 we briefly discuss our implementation. Section 7 presents the results of our experiments on some benchmarks. Section 8 describes related work and Section 9 concludes with a discussion of results.

2 Intra-Instruction Parallelism (IIP)

Embedded processors are ASIPs whose instruction set contains instructions for performing certain commonly executed operations in parallel for generating efficient and compact code. Thus machine dependent optimizations are needed at the lowest level of code generation. In this work, we focus on such optimizations for architectures such as TMS320C2X. The TMS320C25 instruction

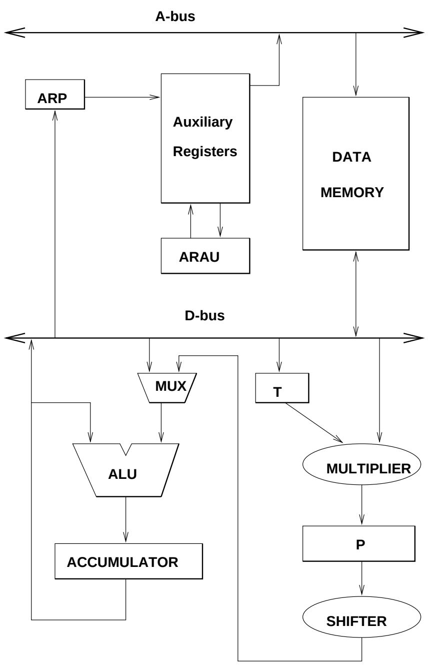

Figure 3: Model of the TMS320C25 data-paths

architecture [38, 21] features *intra-instruction parallelism* or IIP. In particular, this IIP exploits the ability of two functional units to operate in parallel. Figure 3 shows a model of the data path of the TMS320C25 architecture. In this architecture, the multiply/accumulate units can operate in parallel as follows. First, the multiplier takes as input two 16-bit operands, one from the T register and the other from the data memory, and places the the 32-bit product in the P register. Simultaneously, the ALU allows the contents of the P register to be added to or subtracted from the accumulator. The TMS320C25 ISA features several low-level instructions that control the basic operation of the multiplication unit [19] and associated registers such as those listed below.

LT: loads the contents of the specified memory location into T.

MPY: multiplies the contents of T by those of the specified memory location, and places the result in P.

PAC: copies the contents of P into the accumulator.

APAC: adds the contents of P to the accumulator.

SPAC: subtracts the contents of P from the accumulator.

The TMS320C25 ISA permits various combinations of these low-level instructions to be executed in parallel. Each combination is specified by encoding a new instruction whose opcode is formed by merging the opcodes of the individual instructions and occupies **one word** of memory space each. These complex instructions formed by merging opcodes are described below:

LTP: performs an LT and PAC operation in parallel.

LTA: performs an LT and APAC operation in parallel.

LTS: performs an LT and SPAC operation in parallel.

MPYA: performs an MPY and APAC operation in parallel. However, the previous value of the P-register rather than the result of the MPY operation, is added to the accumulator.

MPYS: performs an MPY and SPAC operation in parallel. However, the previous value of the P register, rather than the result of the MPY operation, is subtracted from the accumulator.

Thus, by carefully restructuring codes, the compiler can exploit concurrency amongst different operations allowing generation of these complex instructions. Our code restructuring transformation technique focuses on exploiting the IIP obeying the data path constraints as illustrated in the next section.

2.1 Motivating Example

(a)	(b)
`1: a = b * c + d;`	`2: e = f + g;`
`2: e = f + g;`	`3: h = k + m;`
`3: h = k + m;`	`1: a = b * c + d;`
`4: n = e * h;`	`4: n = e * h;`

Figure 4: Example C code

Consider the example code in Figure 4(a). The statements in the code have been numbered from 1 through 4. For the code shown in Figure 4(a), it is not possible to exploit IIP generating complex instructions. If we move statement 1 before statement 4, then it becomes possible for the code generator to generate a complex instruction `MPYA` which will carry out the addition in statement 1 and multiplication in statement 4 in parallel. The addition in statement 3, i.e., the expression (k + m) cannot be performed in parallel with the multiplication in statement 4, i.e., the expression (e * h), because the `MPYA` instruction accumulates a *previous* product (b * c) (stored in the `p` register in this case), rather than the current product (e * h). Thus, for generating `MPYA`, the product must be computed in P register before its accumulation [19]. Figure 4(b) shows the re-organized code after performing restructuring transformation thus exposing an opportunity to exploit IIP.

We model the problem of code restructuring using a DAG[2]. Figure 5 shows such a directed acyclic graph (DAG) for the motivating example. For the problem under consideration, Directed Acyclic Graph consists of the various statements as nodes and the edges indicating either the benefit of scheduling a particular statement after the other (indicated by the dotted edges) or the dependence between the two statements (indicated by solid edges). The four nodes represent the statements numbered from 1 to 4. A benefit edge[3] exists between node 1 and node 4, since a complex instruction `MPYA` can be generated in place of two simpler instructions `MPY` and `ADD` and thus reduces the size of the assembly code. We also note that statement 4 is dependent on

[2]Data Acyclic Graph

[3]A benefit edge denotes whether there is any benefit to be gained by scheduling the two nodes in question consecutively; the reader is referred to Section 5 for additional explanation regarding benefit edges.

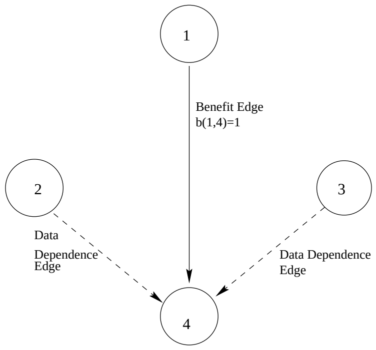

Figure 5: Basic block DAG with the benefit and data dependence edges

statements 2 and 3. This is represented by a broken directed dependence edge from node 2 to 3 and node 2 to node 4, indicating that nodes 2 and 3 must be scheduled before node 4 in any legal node sequence. Thus the code generation problem is to organize the operations to effectively utilize the IIP subject to dependency constraints given by dotted lines.

3 Terms and Definitions

Dependence Graph is constructed for the basic block, indicating the flow and anti dependences in the basic block. This is used later to restrict the number of legal permutations of statements.

Flow Dependence: A statement S1 containing use of a particular variable is said to be flow dependent on a statement S2, which contains the def of the same variable, if S2 is ordered before S1.

Anti Dependence: A statement S1 containing def of a particular variable is said to be anti dependent on a statement S2, which contains the use of the same variable, if S2 is ordered before S1.

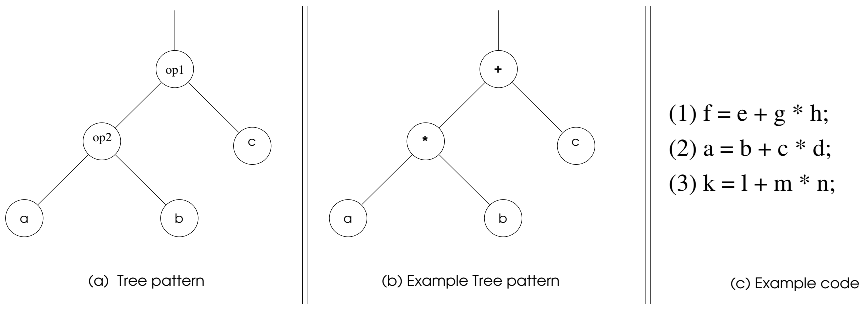

Figure 6: Candidate Statement

Statement Types: In our analysis, we classify the statements with respect to their ability to allow complex instruction generation according to the operands and operators contained in the statement. The statements are represented using an expression trees with the operands as the leaf nodes, and the operators being the internal nodes. Each expression tree is weighed according to the pattern it exhibits. They are classified as follows:

candidate: Please refer to Figure 6(a). It shows an expression tree which contains operators `op1` and `op2`, such that they use different functional units. If there exists complex instructions which can be generated by suitably combining `op1` and/or `op2` with corresponding operator of other sub-tree, we call statements containing the expression tree shown in Figure 6(a) as a **candidate**. For the TMS320C2X DSP, such possible instructions are MPYA, MPYS, etc. Such statments have the maximum potential for forming a complex instruction, because `op1` of this statement can combine with `op2` of previous statemtent to form a complex instruction. In addition to this, `op2` of the current statement can combine with `op1` of the next statement. *Candidate* statements can combine with other *candidate* or *sub-candidates* to form a complex instruction. Figure 6(b) illustrates the expression tree for a *candidate* statement `a = b * c + d`. In Figure 6(c), the `*` operator of statement 2 can be combined with `+` operator of statement 1. In addition to this, the `+` operator of statement 2 can be combined with `*` operation of statement 3.

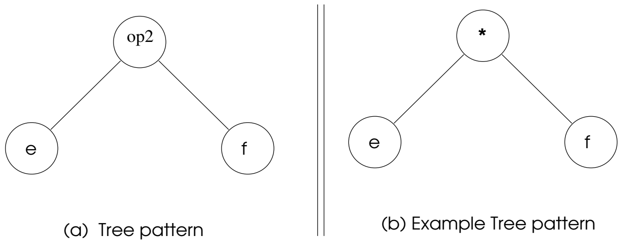

Figure 7: Sub-candidate Statement

sub-candidate: Statements containing only `op2` in their expression tree, which have potential to be combined with *candidate's* `op1` to form complex instruction are termed as **sub-candidates**. The tree pattern for *sub-candidates* is shown in Figure 7(a). The `op2` of *sub-candidates* combine with the `op1` of the *candidates* to form a complex instruction. These statements have a lesser weight because, the formation of a complex instruction strongly depends on a *candidate* statement being found in the basic block. In addition to this, *sub-candidates* can take part in formation of only one complex instruction. For example the pattern shown in Figure 7(b), the multiplication operation will be combined with the addition operation of the Figure 6(b).

dep-candidate: Statements containing only `op1` and having dependency on some other statement, containing `op2` are called as **dep-candidate**. For example, statements having at

```
(1) a = b * c;
(2) x = a + o;
(3) y = c * d;
```

Figure 8: Identifying the Dep-candidates

least one add operation, where at least one of the operands is the result of multiplication done in one of the earlier statements, are called **dep-candidates**. In the figure 8, statement (2) will classify as a *dep-candidate*. The expression trees of statements (1) and (2) in the Figure 8, if combined will result in a tree pattern similar to the one shown in Figure 6. Hence *dep-candidate* is a statement, which when combined with *sub-candidate* could form a *candidate*. Since an occurrence of a *dep-candidates* needs to be adjacent to two candidate statements, with the dependences and involves moving of 2 statements to obtain results, these nodes are given least priority.

Benefit Graph After marking the statements(*vertices in a graph*), **Benefit Graph** is created by inserting edges between the marked nodes. The weights of the edges are determined by the tag on the vertices *i.e the participating statements*

max weight The edges between two candidate have the maximum weight because in this case, **one complex instruction** will be **definitely** generated and also will lead to an **additional** opportunity for creation of one more complex instruction.

med weight The edge between a candidate and a sub-candidate is marked as medium weight. In this case, **one complex instruction** will be **definitely** generated.

min weight The edges sub-candidate $\rightarrow$ dep-candidate and dep-candidate $\rightarrow$ sub-candidate have the minimum weight. In this case, the possibility of generating a complex instruction depends on three statements and not only on two. Hence 2 statement must be moved, with the gain of only **one complex instruction**. Such cases might be rare due to the dependence constraints.

The weights are related to each other by the relation $maxweight > medweight > minweight$. The edges with max weight have the most benefit associated with them and an attempt has been made to select maximum possible edges with max weight, so as to generate maximum number of complex instructions.

4 Code Restructuring Framework

The problem of finding an appropriate code sequence which generates the best code (i.e., a sequence that leads to the most compact code) can be reduced to one of finding a maximum weighted path cover for a given directed acyclic graph (DAG) $G(V, E_b, E_d)$, where, V is the set of nodes and E_b is a set of directed benefit edges and E_d is a set of directed dependence edges. Each statement corresponds to a node n where $n \in V$. A directed benefit edge $e_b(n, m)$ exists between two nodes n and m if some benefit $b(e_b)$ is gained due to scheduling the node m (statement) after the node n (statement). The benefit is determined by an empirical cost-benefit function $f(n, m)$ that maps an ordered pair of two statements corresponding to nodes n and m to a benefit $b(e_b)$. This benefit intuitively represents the amount of parallelism possible by scheduling the two statements one after the other and the possibility of utilizing a suitable complex instruction to exploit the parallelism. The benefit edges $b(e_b)$ can take either of the weights(max/mod/min) as defined in section 3. Of course the data flow constraints of computation and data-path constraints of DSP instructions must permit such a complex instruction generation. A directed dependence edge $e_d(n, m)$ exists from node n to node m if node m is data-dependent on node n and has to be scheduled after node n in any legal sequence of nodes. Thus, the problem can be formulated in graph-theoretic form as follows.

Problem statement: Given a graph $G(V, E_b, E_d)$, find a maximum weighted path cover which also satisfies the partial order in the graph G represented by the set of data dependence edges E_d obeying data flow constraints of computation and data path concurrency of underlying DSP instruction, where the path cover is defined as follows.

Definition 1 *A path cover of a weighted graph $G(V, E_b, E_d)$ is a subgraph $C(V, E', E'')$ of $G(V, E_b, E_d)$ such that*

1. *For every node v in C, $deg(v) \leq 2$.*
2. *There are no cycles in C.* □

4.1 Complexity of the Code Restructuring Transformation Algorithms

The optimal solution to the code restructuring transformation algorithm calculates all possible legal statement sequences. The number of all possible statement sequences for the graph G described above is less than $n!$ because of the presence of the dependence edges. This problem can be reduced to a typical **precedence-constrained traveling salesman problem (PTSP)**. Note that we consider loop body DAGS; hence PTSP. In the precedence-constrained traveling salesman problem (PTSP), each node of a given directed acyclic graph (DAG) is labeled by a point in a metric space M. The problem is to serialize the DAG so that the induced tour through its vertices is of the shortest possible length, where the distance between successive nodes on the tour is the distance between the corresponding points. Charikar et al. [32] show that the PTSP is hard to approximate even on simple metric spaces such as the line and the hypercube showing that it is not possible to exploit geometry to obtain good approximation algorithms for PTSP. This problem has been well studied and is NP-hard and arises in practical transportation and sequencing problems [34, 17]. Charikar et al. try to solve the PTSP problem by developing heuristics that work well for special cases and prove upper bounds for some of the heuristics. Similarly, Lucio et al. have proposed a new bounding procedure and optimal algorithm based on dynamic programming [34] to solve PTSP.

In our problem, the number of benefit edges present in the DAG (basic block) are dependent upon the characteristics of instruction set architecture for which the code is being generated. If the DSP ISA (instruction set architecture) features many complex instructions and a lot of IIP (intra instruction parallelism), then the number of benefit edges increase due to increase in the number of opportunities for exploiting IIP and generating complex instructions. However, if the DSP ISA features a limited amount of parallelism like the TMS320C25, then the number of benefit edges is very small compared to the number of nodes. Due to this reason, we develop an optimal algorithm. The optimal algorithm in general case is exponential however the sparse benefit edges along with dependency constraints make number of possible permutations limited thereby effectively giving an algorithm of low complexity[4].

[4]Please refer to section 7 for performance evaluation of the algorithm.

5 Cost-Benefit Model

We now develop a cost-model to quantify the available IIP to determine the benefit associated with the benefit edges. A cost-benefit model is used to evaluate the benefit of ordering two statements one after the another. This model is built keeping in mind the underlying instruction set architecture but is general enough to extend to other known DSP architectures. In this model, we provide a sufficiently detailed description of the a DSP architecture. DSPs are heterogeneous register set architectures designed to meet performance and area constraints imposed by data-intensive applications found in signal processing and communication domains. In order to achieve this goal, designers choose to implement a set of highly specialized functional units that execute operations frequently required in these algorithms. Examples of these type of units are: Multiply and Accumulate Units (MAC) and Address Calculation units (ACU). It is also common for DSP designers to use dedicated interconnections between a few specialized registers and the available functional units to reduce the cost of the design. This specialization of functional units, registers, and interconnection is reflected in the processor Instruction Set Architecture (ISA). Frequently, the ISA of DSPs and Application Specific Instruction Set Processor (ASIPs), have dedicated instructions which take operands and store results into specific registers as opposed to using generic register files. We consider the Texas Instruments' DSP320C25 architecture as the representative target architecture for utilizing IIP.

```
(1) add m      a:  PLUS(a,m)
(2) apac       a:  PLUS(a,p)
(3) spac       a:  MINUS(a,p)
(4) mpy m      p:  MUL(m,t)
(5) mpyk k     p:  MUL(t,CONST)
(6) lack k     a:  CONST
(7) pac        a:  p
(8) sacl m     m:  a
(9) lac m      a:  m
(10) lt m      t:  m
```

Figure 9: Partial Description of TMS320C25 processor ISA

5.1 Partial Functional Description

Figure 9 describes the partial functional description of the TMS320C25 Instruction Set Architecture (ISA) [6, 3]. The partial functional description indicates the mapping of pseudo instructions to the actual data transfer paths. The left hand side of the figure 9 indicates a small subset of the pseudo instructions available in TMS320C25. The format shown on the left hand side, shows the source of the operands, the operator and the destination of the operation being done. Both the source and destinations are registers present in the DSP. The pseudo instructions are numbered to help in identifying the *Register Transfer Graph* as shown in Figure 10. Storage locations in the data path are represented by lowercase letters (`a, p, t, m`), where `a` is an accumulator, `p` the product register, `m` the memory and `t` is used to store one of the operands of the multiplier.

Uppercase letters are used to specify operations (`PLUS, MINUS, MUL`) and constants (`CONST`).

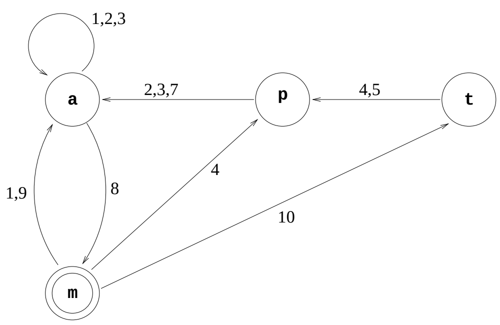

Figure 10: TMS320C25 Register Transfer Graph (RTG)

5.2 Register Transfer Graph (RTG)

Figure 10 gives a register transfer graph (RTG) representation of the TMS320C25 architecture [6, 5]. This representation can be easily derived from the partial description of the processor's ISA in Figure 9. The instruction identifiers in Figure 10 are used to label each edge of the RTG. The RTG is a structural representation of the data path topology which contains information about the instructions in the processor ISA. The nodes in the RTG describe locations in the data path such as registers, register files and memories. Memories and register files are represented by double circles in order to distinguish them from simple register nodes. An edge between nodes r_1 and r_2 in the RTG defines a path between these locations in the data path. Labels on this edge correspond to the identifiers of the instructions in the ISA which take one operand from the location r_1 and store the result into location r_2. For example, from Figure 10, one can see that accumulator a has two incoming edges numbered (1). One edge originating at node a (self -loop edge) and another at node m, together describe the transfer operations required by instruction `PLUS(a,m)` labeled by (1).

We use this **register transfer graph (RTG)** representation to estimate the **benefit** of ordering one statement after another. The total benefit involved is the amount of **intra instruction parallelism** possible.

5.3 Benefit Algorithm

We use the following algorithm to estimate the amount of intra instruction parallelism possible between two successive statements.

Step 1: Obtain a partial functional description of the target Instruction Set Architecture as shown in Figure 9 for TMS320C25.

Step 2: Obtain a Register Transfer Graph for the target architecture that shows all the data paths traversed when the instructions contained in the Instruction Set Architecture are executed.

Step 3: Analyze the representations obtained in Steps 1 and 2 and the Instruction Set Architecture to obtain all the parallel data paths in form of ordered pairs. The first data path in the pair can be overlapped with the second data path which means that a complex instruction

```
function MaxBenefitCodeRestructuring(G)
Input:
  G (TreeDAG)
  {TreeDAG is a directed graph representation that contains no common subexpressions.
  Each node in a TreeDAG, represented by TreeDAGNode corresponds to an expression tree.
  A TreeDAG edge directed from node n_1 to node n_2 specifies that the expression tree
  corresponding to n_1 must be scheduled before the expression tree corresponding to n_2.}
Output:
  Optimal beneficial tree node list (tnl), the most beneficial statement sequence.
begin
    Build_Benefit_Graph(G);
    tnl = ∅;
    depth_of_tree = 0;
    Optimal_Code_Restructuring_Algorithm(tnl,depth_of_tree);
end

function Optimal_Code_Restructuring_Algorithm(tree_node_list curr_tnl, curr_depth)
Input:
    curr_tnl (The current Schedule)
    curr_depth (The current depth of the tree where the new node is to be inserted)
Output:
    best_sched (Tree node list with optimal schedule)
begin
    if curr_depth = number_of_nodes
            Find the weight of the curr_tnl using the Benefit Graph
            if weight > curr_max_weight
                curr_max_weight = weight
                best_sched = curr_tnl
                end if
            end if
    for each node in the schedule do
            Check_if_valid(tnl, curr_node) // dependency check
            if valid
                insert the curr_node in the tnl
                depth++
                Optimal_Code_Restructuring_Algorithm(curr_tnl,depth)
            end if
    end for
end
```

Figure 11: Optimal code restructuring algorithm.

```
function Build_Benefit_Graph(G)
Input:
  G (TreeDAG)
Output:
  Benefit Graph which indicates the benefit of having two statements next to each other.
begin
    for each node n1 in the TreeDAG G do
            for every other node n2 in the TreeDAG G do
                  Identify parallel operation between n1 and n2 as per the RTG.
                  Mark the edge between n1 and n2 with appropriate weight
                  (MAX / MED /MIN)
            end for;
    end for;
end
```

Figure 12: Function for building the benefit graph.

can be generated leading to the exploitation of the IIP. As seen from the RTG in Figure 10 and partial description in Figure 9, the `MPY` instruction follows data paths `p to t` and `p to m`, while the `ADD` instruction follows the data paths `a to a` and `m to a`. These paths are independent of each other and hence these instructions can be overlapped and either a complex instruction can be made (as in TMS320C25) or executed in parallel (as in EPIC style of processors such as TMS320C6X). The `MPY` and `ADD` instruction can be combined together to form the `MPYA` instruction. Other such kind of pairs are found and stored in a set for further comparisons.

Steps 1 to 3 need to be done for a particular DSP architecture only once and will be used for subsequent compiling of codes. This information is required for identifying the instructions exhibiting parallelism.

Step 4: Analyze the expression trees of the two statements and obtain the data paths for them. The selection of the two statements is being done in an optimal way by finding all the possible combinations.

Step 5: If the data paths needed for execution belong to the set of the parallel data path pairs present in the RTG, then there is a benefit in executing the two statements successively. The amount of benefit is decided by the various patterns represented in the expression tree. The statements are categorized into **candidates**, **sub-candidates** and **dep-candidates** according to the tree structure of the statement. Thus after identifying the benefit, we also identify the weight of the edge. Finally the schedule with the best benefit is obtained by searching exhaustively using the algorithm shown in Figure 11

5.4 Illustration of our Approach

We illustrate our approach on the code shown in Figure 13. For a particular ISA, the RTG is created using the partial description of the ISA. Dependence Graph for the code is shown in

`S1:`	`a = b + c * d`	Candidate
`S2:`	`x = y + z;`	
`S3:`	`o = m * n;`	Sub-Candidate
`S4:`	`p = o + g;`	Dep-Candidate
`S5:`	`l = x * o;`	Sub-Candidate
`S6:`	`e = f + i * j;`	Candidate

Figure 13: Illustration of our approach

Figure 14(b). Then the statements are marked for benefit, as per the expression trees. `Statements 1` and `6` are candidates as the tree pattern shown in Figure 6 is a sub-tree of their expression tree. Similarly `Statements 3` and `5` are sub-candidates. We have to use the dependency information to mark the dep-candidates, which is `Statement 4`

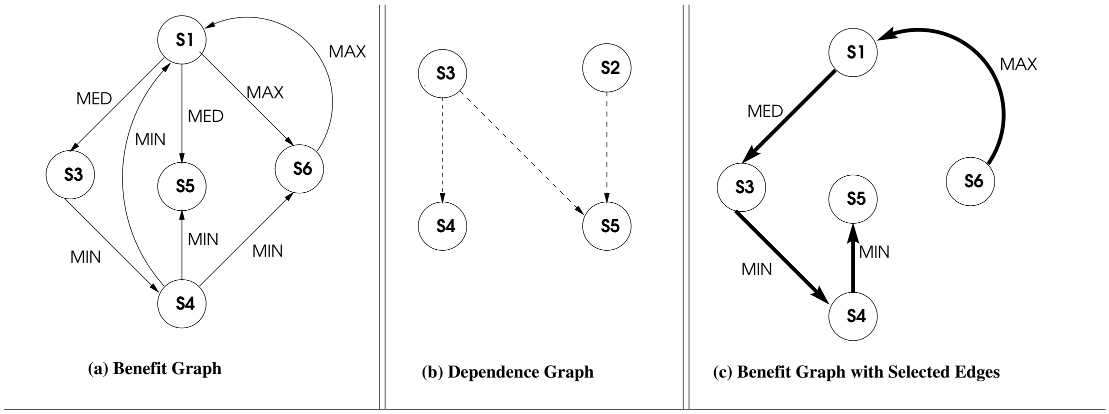

Figure 14: Benefit and Dependence Graphs for the Illustration Code

We start inserting the edges in the *tree node list* using the *benefit algorithm* shown in Figure 11. We start with Statement 1 as the first node in *tree node list* and then keep on adding nodes one by one till all the statements are inserted in the tree node list. This is indicated by the variable *depth*, which becomes equal to the number of statements in the block. Then we calculate the sum of weights of all the edges and compare with the current value of **max_weight**. For the example shown in Figure 13, the first set of legal sequence will be `S1` $\rightarrow$ `S2` $\rightarrow$ `S3` $\rightarrow$ `S4` $\rightarrow$ `S5` $\rightarrow$ `S6`. The only benefit edges selected in this process are `S3` $\rightarrow$ `S4` $\rightarrow$ `S5`, hence the total weight will be **2**. Node are selected from the set of remaining nodes and pushed on the top of the *tree node list*. Once the *tree node list* set is full, nodes are popped off, one by one and then pushed on the list again such that we get different combinations. This continues till we exhaust all the possibilities. For the figure shown, the best possibility will be `S2` $\rightarrow$ `S6` $\rightarrow$ `S1` $\rightarrow$ `S3` $\rightarrow$ `S4` $\rightarrow$ `S5`, which has a total weight of 14 as shown in the Figure 13(c).

Pruning of the Benefit Graph We now show that although our algorithm is potentially exponential, the dependency constraints considerably prune the search space. This is a typical **Traveling Salesman Problem(TSP)**. Due to dependencies of statements, not all statement sequences are legal and thus adding constraints to the benefit graph. This leads to the problem of **Precedence Constrained TSP**. Consider the situation where there are 3 elements currently in the *tree node list* i.e `S1` $\rightarrow$ `S6` $\rightarrow$ `S3`. If we try to insert `S4` in the *tree node list*, the whole sub-tree following after that point is pruned, since `S4` is dependent on `S2` and `S2` is not yet present in the *tree node list*.

6 Implementation of our approach

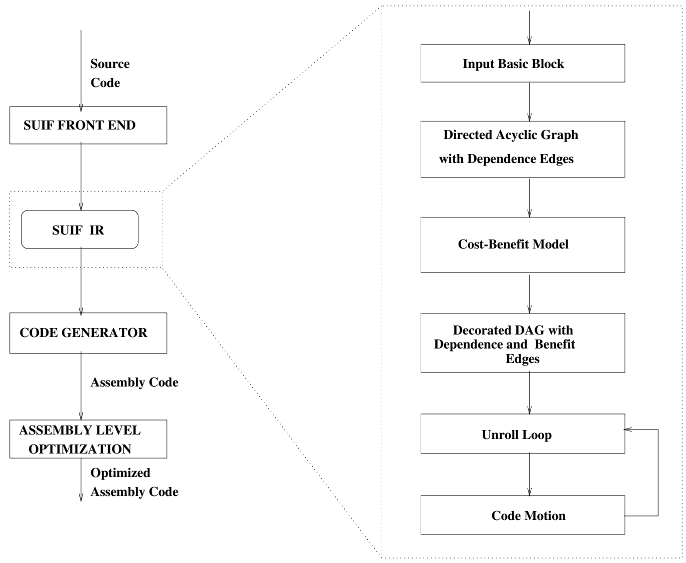

Figure 15: Outline of our approach

We have implemented our code restructuring transformation framework on the optimized IR generated by the SUIF [36] front end. Refer to Figure 15 for details. This work primarily involves the steps shown in the dotted boxes, which shows the different phases of our work. SUIF serves as a front end to the SPAM compiler. SPAM [19] is a retargetable optimizing compiler for embedded digital signal processors. The structure of the SPAM compiler is shown in Figure 16. Please refer to [19] for detailed description on SPAM.

The C code, which is to be compiled is input to the SUIF [36] front end, which results in the SUIF IR. The IR is then input to the SPAM compiler. The dependence graph(DAG) and the benefit graph is created (as explained in the section 5.3) and the code is restructured in the basic block to get maximum number of complex instructions. The loops are unrolled to get more benefit as we have more opportunities for generation of complex instructions. The new optimized IR is then fed to the code generator to obtain the final assembly code. Optimizations at assembly level, if any, are then applied to the code. We have implemented the optimal algorithm, due the fact that the benefit graphs are sparse. Data dependencies further prune the graph thus resulting in compile times as shown in the Table 1. The time taken for compilation is dependent on three factors:

- Number of statements in each basic block

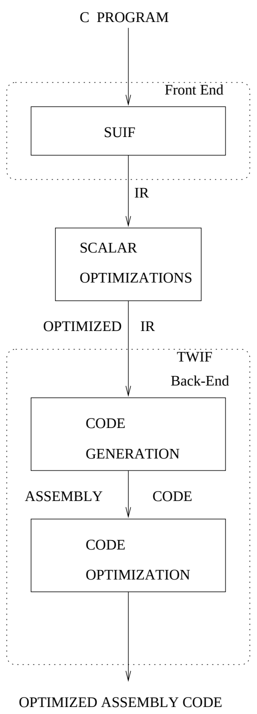

Figure 16: Structure of the SPAM Compiler

- Number of basic blocks
- Dependencies between the statements

The compilation times for most of the benchmarks are small, except for one, which is due to bigger basic block. The most important factor influencing the compilation time is the number of statements(n) in a particular block, which leads to $n!$ possible combination of statements. However some of the combinations are illegal due to the dependencies leading to smaller number of possibilities.

Table 1: Compilation Time for Benchmarks

Sr No.	Benchmark	Compile Time (*in secs*)
1	IIR Biquad	5
2	n realupdate	2
3	fft_duhamel	12
4	nnlearn	2
5	mixfft	2

Some of the important steps of the compilation process are as follows

Statement listing We first generate the TreeDAG decorated with dependent and benefit edges using the framework in Section 5. Then we use an optimal exhaustive algorithm which will find all the legal sequences and return the most beneficial sequence for the code. The new ordering of nodes thus obtained is then fed to the next stage of the SPAM compiler which is the code generator.

Code generator The code generator reads OLIVE [39] expression trees and returns assembly code. The SPAM compiler uses OLIVE as its machine description language. OLIVE is a code generator-generator which takes as input an OLIVE machine description file to generate a code-generator for the target processor. Once the code is generated, some machine dependent peep hole optimizations are performed.

Assembly level optimization After the straight line assembly code is generated by the code-generator, many local compaction techniques such as list scheduling can be used to exploit the available parallelism. Since IIP in the TMS320C25 is restricted to just a few cases, more aggressive compaction techniques may be used to exploit this IIP. In particular, the aggressive clique-covering-based [24] local compaction algorithm is be used to finally generate the optimized assembly code.

The code restructuring framework is limited only to Basic Block. The work is not extended beyond basic blocks since, it was found that doing the work beyond basic block, usually leads to increase in code size. Code size reduction being our primary goal to fit the code in the limited amount of memory, global restructuring (across basic blocks) does not seem to be useful.

7 Performance Evaluation

We tested our framework on loop kernels extracted from standard benchmark codes such as Image processing, FFTW and kernel programs from the DSPstone benchmark suite. Some of these benchmark programs implement frequently used DSP operations such as complex number multiplication, matrix multiplication and convolution etc.

7.1 Evaluation Parameters

We considered the following important parameters while benchmarking our framework.

Performance: We executed the compiled codes on TI's TMS320C25. We recorded the number of clock cycles saved while executing the code generated using our framework. The run time performance of the generated code is a significant issue due to the real time performance requirements of the embedded systems.

Code Size: We observed the static code size of the generated assembly which is a good measure of the DSP code quality. The generated code size should be as small as possible because of the limited memory present on the embedded systems. For example, TMS320C25 DSP has only 544 16-bit words of RAM and 4K 16-bit words of program ROM. Thus code size reductions is important in case of limited memory DSPs such as TMS320C25.

Table 2: Results

Unroll Factor	No Code Restructuring: Number of Instructions	No Code Restructuring: Memory Required	No Code Restructuring: Dyn Instr Count	Code Restructuring: Number of Instructions	Code Restructuring: Memory Required	Code Restructuring: Dyn Instr Count	%Improvement: Code Size	%Improvement: Dyn Instr Count
IIR Biquad – Data Req: *16*								
NU	93	95	363	91	93	355	2.2	2.2
2	182	184	361	177	179	351	2.7	2.8
4	360	362	360	349	351	349	3.1	3.1
n realupdate – Data Req: *65*								
NU	43	49	523	43	49	523	0	0
2	74	80	515	73	79	507	1.4	1.6
8	260	266	509	253	259	495	2.7	2.8
16	498	499	498	483	484	483	2.6	2.6
fft_duhamel – Data Req:*34*								
NU	304	307	9449	297	300	9255	2.3	2.1
2	589	592	9289	574	577	9049	2.6	2.6
4	1159	1162	9209	1128	1131	8961	2.7	2.7
nnlearn – Data Req: *200*								
NU	123	130	2896	117	124	2608	4.8	9.9
2	177	186	2776	168	177	2560	5.1	7.8
3	231	242	2736	219	230	2544	5.2	7.1
4	285	298	2716	270	283	2536	5.3	6.6
6	393	410	2696	372	389	2528	5.3	6.2
8	501	522	2686	474	495	2524	5.4	6.0
12	717	746	2676	678	708	2520	5.4	5.8
16	933	970	2671	882	919	2518	5.5	5.7
24	1365	1418	2666	1290	1343	2516	5.5	5.6
mixfft – Data Req: *117*								
NU	176	203	3904	169	196	3734	4.0	4.4
2	285	326	3750	274	325	3574	3.9	4.7
4	511	580	3710	485	554	3502	5.1	5.6
8	939	1064	3594	893	1018	3410	4.9	5.1
NU indicates No Un-rolling								

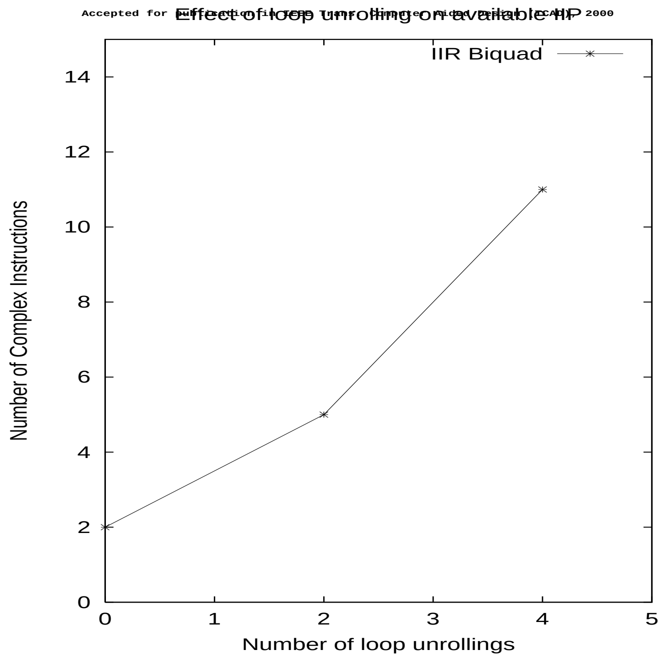

Figure 17: Effect of loop unrolling on complex instruction generation for IIR-Biquad

7.2 Benchmarks

DSPstone Benchmark-I This benchmark is a loop extracted from the `iir_biquad_N_sections` code in the DSPstone [40] benchmark suite.

DSPstone Benchmark-II This is another benchmark from the DSPstone kernel suite. It performs N real updates and is used for filter benchmarking.

Image Processing Benchmark-I The function `nnlearn` is another Image Processing code, which is used for propagating values in the network.

FFT-I The function `mixfft` is taken from the benchFFTW suite. It is used for initializing tables, while taking the fast fourier transforms.

FFT-II The function `FFT_duhamel` is another kernel from the benchFFTW suite. It is one of the methods of getting split-radix FFT.

7.3 Discussion of Results

The results are obtained on the 5 benchmarks described earlier in the section. The static code size, memory required and the dynamic instruction count for the various benchmarks were calculated, with and without unrolling combining with/without our framework. All the results are tabulated in Table 2. `NU` in the table indicates `No Unroll`. In the graphs 17, 18, 19, 20 and 21, the number of complex instructions generated are plotted against the number of loop unrolling. The graphs show

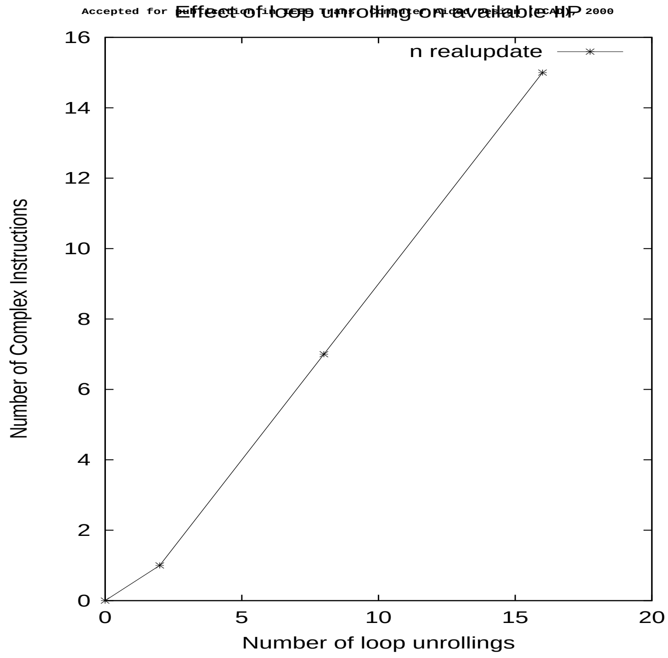

Figure 18: Effect of loop unrolling on complex instruction generation for n realupdates

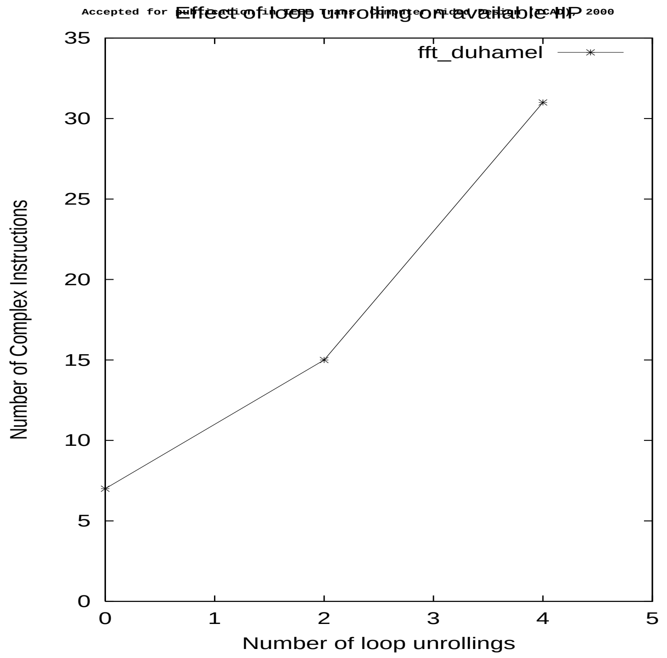

Figure 19: Effect of loop unrolling on dynamic instruction count for fft_duhamel

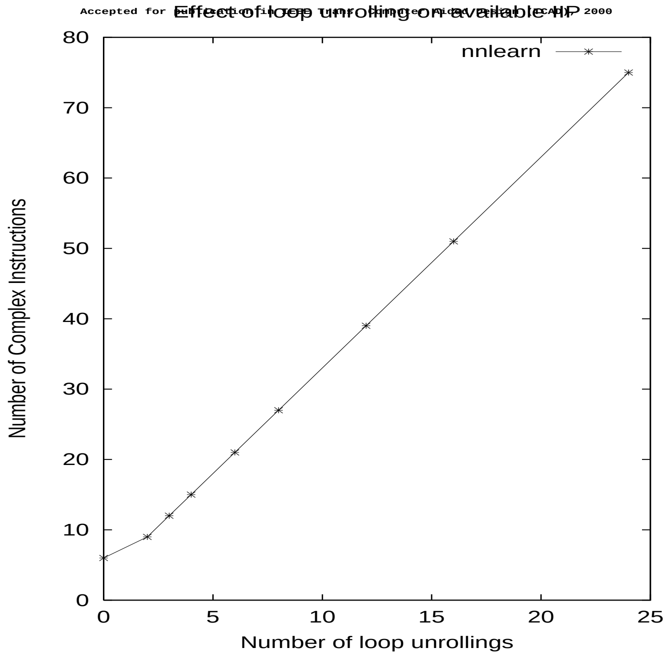

Figure 20: Effect of loop unrolling on complex instruction generation for nnlearn

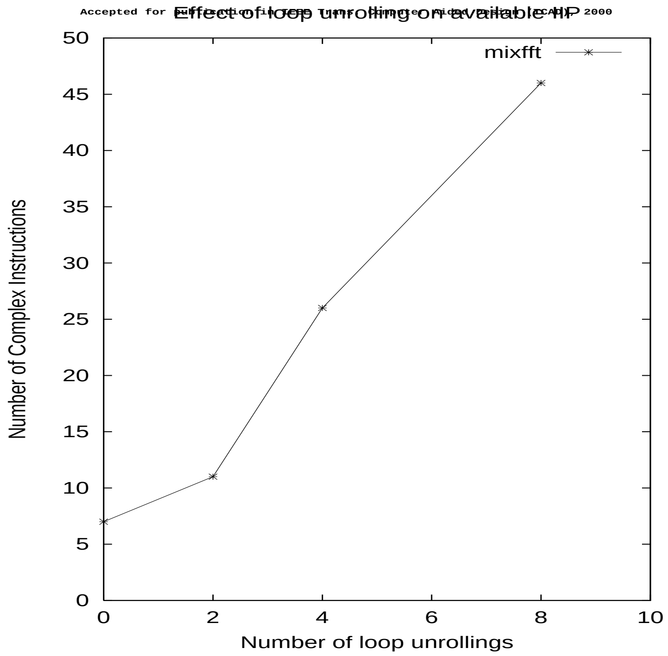

Figure 21: Effect of loop unrolling on complex instruction generation for mixfft

a steady increase in the number of complex instruction generation, showing that these techniques scale well with respect to amount of un-rolling.

The following comparisons are done in order to show the improvement in reduction of dynamic instruction count. The code generated with unrolling and restructuring is compared to the code

- without restructuring or unrolling (native code)
- without restructuring, but with unrolling
- with restructuring, but without unrolling

These comparisons are plotted in Figures 22, 23, 24, 25 and 26.

We obtain improvement in terms of code size upto 5.5%. A decrease in code size leads to decrease in the memory required. The TMS320C25, provides a total of 544 16-bit words, of which 288 is always used for data, while the rest 256 words can be used either for data or for program memory. The TMS320C2X can address upto 64 K of memory, with the first 1K reserved for on-chip memory. We have indicated the instances where restructuring after loop-unrolling helped in fitting of the code in either 256, 512 or 1024 word limit of the memory. Although the decrease in code size over unoptimized version is low, there are certain important advantages of code size reduction:

1. It forms a significant portion of the available on-chip memory for holding instructions. Due to limited amount of memory available in typical DSPs, even a small improvement can make a large difference.
2. For better execution, code can be unrolled for certain number of times so as to fit the memory. By reducing the size of code, we can unroll the loops for more number of times leading to better execution efficiency. As shown in the Table 2 by the highlighted rows, due to the limit on the size of the program memory(multiples of 256 words), the unrolled loop couldn't fit into the on-chip program memory without restructuring. In other words, restructuring allowed code compaction which in turn facilitated more unrolling and compaction.
3. One more advantage of our restructuring is more scope to fit the data required for the program. There has been work done for using the on-chip memory to store spill values [9] or as ultra-fast cache [31]. Reducing the amount of memory occupied by the code allows the on-chip memory to be used for other optimizations such as above.

Summarizing, we have obtained an decrease in dynamic instruction count of upto 9.9% which leads to speedup of 10%.

7.4 Loop Unrolling

One of the characteristics of DSP applications is that the majority of the processing efforts are often concentrated in a few relatively small pieces (loops) of the program which have small loop bodies. Since, the loops are small, they can be unrolled without significantly increasing the code size. The amount of intra instruction parallelism is limited within the loop body. If we unroll a loop, the opportunities for exploiting intra instruction parallelism increase.

Definition 2 *Let the loop body be represented by the graph $G(V, E_b, E_d)$ where V is the set of n nodes and E_b is a set of directed benefit edges and E_d is a set of directed dependence edges. Let $G'(V', E'_b, E'_d)$ be the new DAG representing the k-times unrolled loop body where V' is the set of nodes and E'_b is the new set of directed benefit edges and E'_d is the new set of directed dependence edges.* □

Observation 1 *Given two directed acyclic graphs $G(V, E_b, E_d)$ (original loop body) and $G'(V', E'_b, E'_d)$ (k-times unrolled loop body). The number of nodes in set V' is k times the number of nodes in set V. The number of benefit edges in E'_b could be more than k times the number of benefit edges in E_b. This could lead to opportunities for more IIP.*

$$n(V') = k * n(V) \quad and \quad n(E'_b) \geq k * n(E_b) \quad and$$

$$\sum_{e'_b \in E'_b} b'(e'_b) \geq k * \sum_{e_b \in E_b} b(e_b)$$

Proof: Since the loop body has been unrolled k times, the number of nodes $n(V')$ (statements)increases by k-fold as well. The number of benefit edges (representing the available instruction level parallelism) may increase by more than k-fold since some new benefit edges may now be generated in addition to the previously existing benefit edges. As a result, even the total benefit associated with the k-times unrolled loop could also increase by more than k-fold since the total benefit $\sum_{e'_b \in E'_b} b'(e'_b)$ is the sum of the benefits associated with the benefit edges e'_b in the new DAG $G'(V', E'_b, E'_d)$, showing that more IIP opportunities could be available in unrolled loop. □

7.4.1 Results for loop unrolling

We studied the effect of loop unrolling on the amount of intra instruction parallelism available for a loop body. Although un-rolling results in decrease in the number of dynamic instruction count, resulting in better execution efficiency, it also results in increased code size. Most of the DSPs have the inherent problem of low on-chip memory. Some amount of on-chip memory is required for the data segment of the loop in execution, leaving lesser amount of memory for the code. In our work we have limited the number of un-rollings on the basis of code-size and the on-chip memory available for the loop, after taking into consideration the data requirements. The TMS320C25 provides a total of 544 16-bit words of on-chip data RAM (288 words are always for data and the other 256 words can be configured as program or data). The TMS320C26 has comparatively larger amount of memory, 1568 16-bit words, of which 32 words are exclusively for data while the others can be used for data as well as program memory. Our results are generated on both TMS320C25 as well as TMS320C26 DSP processor. We unroll a loop to the extent that generated code occupies available on-chip memory facilitating more complex instruction to be generated. In Figures 17, 18, 19, 20 and 21, the number of complex instructions generated are plotted against the number of the times a loop is unrolled. The plots show that the number of complex instructions generated (which is a good measure of the available intra instruction parallelism (IIP))increase as the loop is unrolled. In most cases, it is observed that as we unroll more, the rate at which complex instructions are generated increases. Unrolling causes the code size to increase, but the increase is not proportional to the unroll factor. For example, if a loop

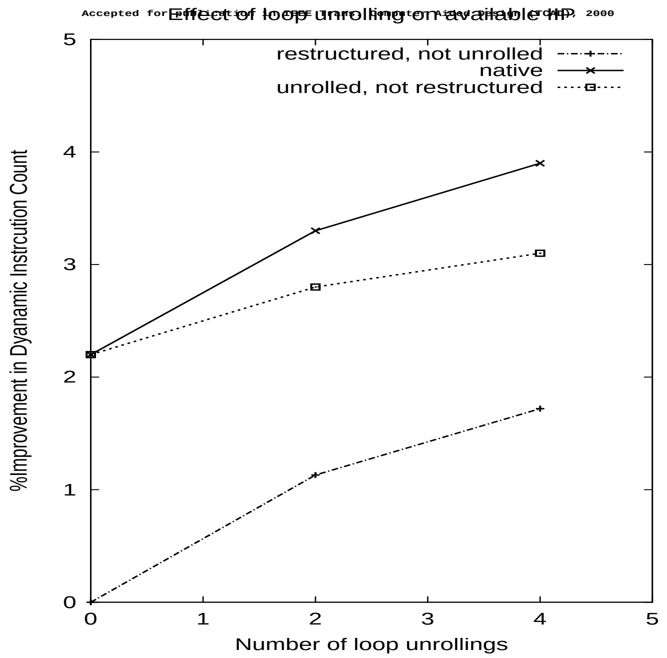

Figure 22: Effect of loop unrolling on dynamic instruction count for IIR-Biquad

is unrolled twice, the code size is not doubled, because due to unrolling, more opportunities are available for complex instruction generation. As a result the increase in code size is lesser than proportional.

Unrolling also has a positive effect on dynamic instruction count. We see that that in most of the cases, there is an increase in the reduction of dynamic instruction count as the loop is unrolled. In Figures 22, 23, 24, 25 and 26, the percentage improvement in run-time performance is plotted against the number of loop unrollings. In most of the cases, unrolling shows an improvement in reduction of dynamic instruction count. We again see that the performance improves as the loop is unrolled in most of the cases. However after a certain number of loop unrollings, there is no further significant gain in performance. This *saturating effect* can be attributed to decreasing code restructuring transformation opportunities with the amount of unrolling. From the graphs, we can conclude that it is practical to unroll the loops only to the extent until there is a significant improvement in performance to avoid code size penalty. The amount of unrolling that is sufficient ranges from as low as 2-3 to as high as 8-9. Although, we have not studied this aspect, loop unrolling also opens up the possibility of using Liao's dictionary-based code compression technique [11].

8 Related Work

Efficient techniques for instruction scheduling have been the focus of code optimizations for both super-scalar as well as VLIW architectures. Trace scheduling introduced by Fisher [14] is one of the first *global* scheduling techniques that was highly successful. One of the criticisms of trace scheduling is code growth; however, as shown by Freudenberger et al. [16], there are ways to avoid

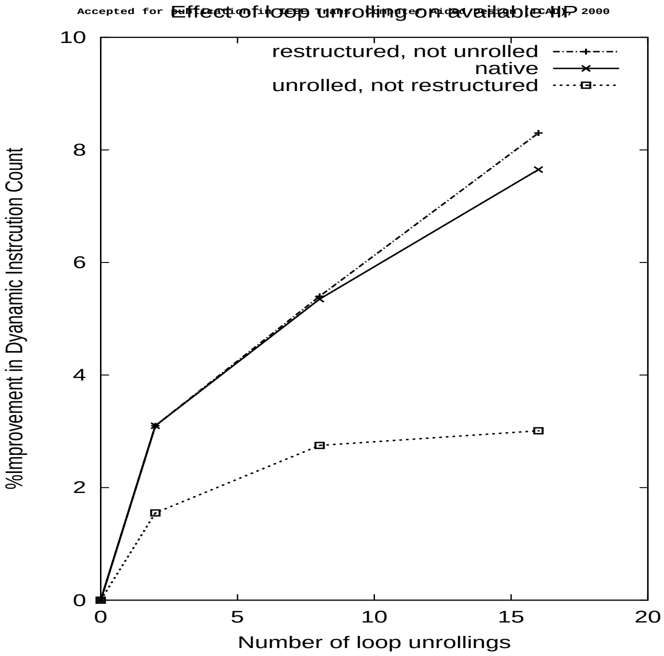

Figure 23: Effect of loop unrolling on dynamic instruction count for n realupdates

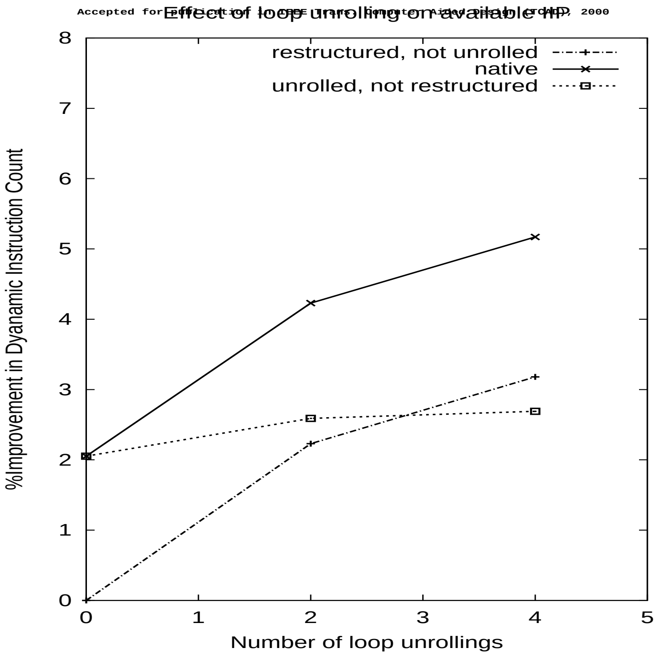

Figure 24: Effect of loop unrolling on dynamic instruction count for fft_duhamel

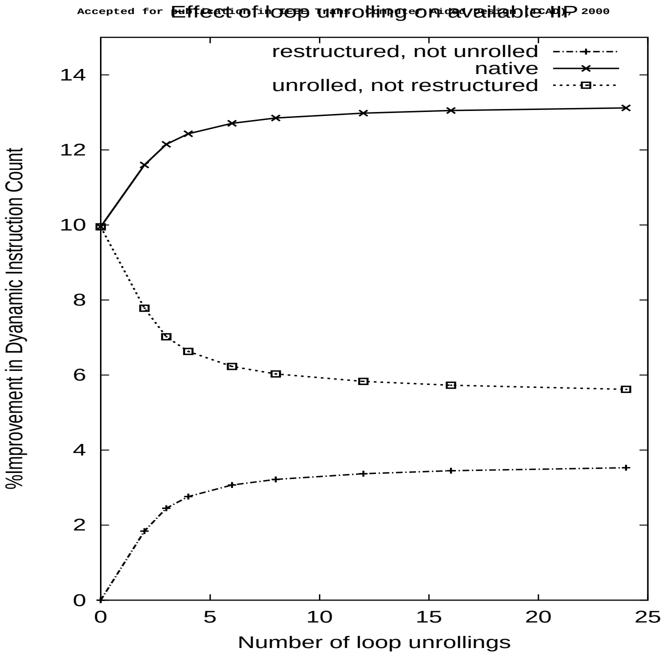

Figure 25: Effect of loop unrolling on dynamic instruction count for nnlearn

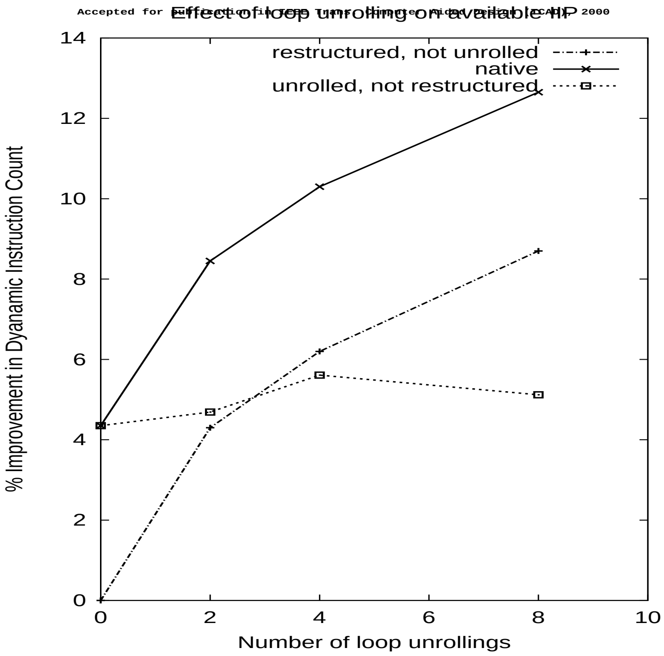

Figure 26: Effect of loop unrolling on dynamic instruction count for mixfft

generation of compensation code. Region scheduling was proposed by Gupta and Soffa [20] that significantly improves performance through duplication. Recently, Cooper and Schielke [8] have proposed two code scheduling methods: extended basic block scheduling (EBBS) and dominator path scheduling (DPS) to exploit ILP in a global scope.

Kemal Ebcioglu et al. [12] have designed a set of new algorithm for reducing path length and removing stalls due to branches, namely speculative load-store motion out of loops, unspeculation, limited combining, basic block expansion, prolog tailoring. Wen-mei W. Hwu et al. [29] have developed a set of techniques to exploit ILP across basic block boundaries. Rajiv Gupta et al [30] use value prediction to remove data dependencies and improve the instruction schedule, but at the cost of code growth in the form of compensation code.

However, these techniques do not deal with the issue of complex and irregular data paths which is one of the significant issue in code generation especially on digital signal processors (DSPs), due to the ASIP nature of the processors.

Some work has been done in the area of loop pipelining, which primarily focuses on scheduling of instructions. Our work is different from this since it focuses on issue of complex instruction generation and not on instruction scheduling since we address the issues of code size and efficiency together. In this work, we attempt to reduce the dynamic instruction count by generation of as many complex instructions as possible. One of the key objectives of this work is to unroll the loop as many times as possible so as to fit it in the small amount of memory. By generating complex instructions we try to reduce the number of instructions in a particular loop. As the number of instructions are reduced, more code can fit in the same amount of memory leading to more un-rolling and thus better performance.

We discuss the various code generation techniques targeted towards embedded systems especially those related to code size optimization. Many compiler optimization techniques make use of special architectural features [28, 1]. Liao et al. [25, 4] developed a framework in which the compiler delays storage allocation of variables, moving it from the front-end to the code generation step that selects addressing modes, thus increasing opportunities to use auto-increment/auto-decrement modes. This optimization helps in subsuming the address arithmetic into auto-increment and auto-decrement modes which leads to an improvement in the performance and reduction in the size of the generated code. Sudarsanam et al. [37] further enhanced this work by comprehensively analyzing the performance benefits and hardware costs due to an auto-increment/decrement feature that varies from $-l$ to $+l$, and allowing access to k address registers in an address generator. This analysis computes the values of l and k that results in the minimum code size. The above two techniques were based on the *offset assignment problem* initially formulated by Liao et al. [25]. Amit Rao and Santosh Pande [33] further extend this work by presenting heuristic techniques to optimize the access sequence of variables by applying algebraic transformations (commutativity and associativity) on expression trees to obtain the *least cost offset assignment*. Bowman et al. [7] have attempted to decrease process memory requirements by overlapping uninitialized global variables with other static data, compressing code by overlapping instructions via cross jumping and abstraction of code portions. They achieve a significant reduction in process memory requirements by applying these techniques.

Ernst et al. [15] suggest code compression as a way of improving total performance even when CPU has to decompress or interpret the compressed code. They describe real world scenarios where code compression can save space and total time when transmission of code over wire is the bottleneck and introduce a wire VM code and an interpretable VM code. Liao et al. [11] address the problem of code size minimization for embedded DSP processors by using data compression methods. They present a framework for code size minimization where the compressed code consists of a dictionary and a skeleton.

Landskov et al. [23] have discussed various techniques for local microcode compaction, in which they have presented algorithms for compaction based on machine independent optimizations. These algorithms, being generic, might not be able to capture all the possibilities of complex instruction generation. In our work the work is targeted towards a specific machine, which considers data paths traversed and suitably rearrange them such that parallelism between data paths can be exploited by suitably generating a complex instruction. The motivation is to decrease code size and increase code efficiency. However this demands intimate knowledge of register transfer paths traversed by the instructions and can be done as machine specific optimization for which the generic techniques of microcode compaction are not useful.

Our work most closely compares with Liao's work. Liao et al. [24] formulate the problem of instruction selection in code generation for embedded DSP processors as directed acyclic graph (DAG) covering. Traditional code generation methods result in inefficient code on these processors since these DSPs have irregular datapaths. The authors give a binate covering formulation for optimal code generation on a one-register machine. The authors have shown that optimal instruction selection in the case of accumulator-based architectures requires a partial scheduling of nodes in the DAG. In addition to the selection of complex instructions the authors have also tried to minimize the reloads and spill. Their method detects opportunities for combining neighboring simple instructions. However, limited opportunities exist if restricted to combining neighboring simple instructions. One could expose far more opportunities for combining such instructions through code restructuring transformation obeying data dependencies. Consider the code shown in Figure 27, the addition of `b * c` and `d` from statement 1 and multiplication of `e` and `h` in statement 3 can be done in parallel.

```
(1)  a = b * c + d;
(2)  h = f + g;
(3)  n = e * h;
```

Figure 27: Example C code

In most of the cases, the neighbouring operations are considered for generation of complex instructions and thus, for the example shown in Figure 27,no complex instructions were generated. We need to do some code restructuring in order to exploit the ILP present in the above code. In this work, we focus on a new problem: what is the best code restructuring transformation to effectively exploit intra-instruction parallelism through complex instruction generation for embedded DSPs, given constraint of data paths and dependencies?. We utilize concurrency in data paths using register-transfer graphs to exploit this intra-instruction parallelism and propose framework based on code motion and loop unrolling.

Corina Lee et.al [26] use the Multiply and Accumulate(MAC) operation is to improve the execution speed for multiplying the elements of the array, but this leads to some amount of code growth especially for multiplying non-vectors. In this work we restructure the code to identify situations where a complex instruction can be generate by moving the appropriate add and multiply operations next to each other.

9 Conclusions and Future Work

Most of the modern DSP architectures feature intra instruction parallelism, which is the ability of the architecture to execute individual operations in parallel. This is reflected by the presence of complex instructions in the instruction set architecture (ISA). An improvement in performance and reduction in code size can be achieved by exploiting the available intra instruction parallelism in the DSP ISA.

In this work, we have presented a code restructuring transformation framework that maximally exploits the IIP present. We have formulated this problem as maximum benefit code restructuring transformation problem which can be reduced to a precedence constrained traveling salesman problem and hence is NP-hard. We have developed a benefit model based on the data path parallelism using the register transfer graph (RTG) representation of the DSP ISA. Using this benefit model, we have proposed a code restructuring transformation to maximize the available benefit (IIP) . Our code restructuring transformation framework improves upon the earlier techniques [24] by looking for code restructuring transformation opportunities within the entire basic block and not just the neighboring statements.

We evaluate the performance of the proposed optimal framework and obtain results for a number of benchmarks from DSPstone benchmark suites, FFTW and Image processing codes. The performance measured as a number of clock cycles (dynamic instruction count and code size) improves for all the studied cases. We also analyze trade-offs between IIP and loop unrolling and conclude that on an average 4-5 unrolling could significantly improve IIP opportunities. The number of unrollings largely depends on the size of the on-chip memory in the processor. Another usefulness of loop unrolling is that some types of register allocation optimizations leading to a reduction in the number of load/spills can be performed leading to a further reduction in the code size [22].

Currently we are extending this work to exploit ILP in the latest embedded processor architectures such as the Texas Instruments TMS320C6X.

Acknowledgments

We would like to thank the Princeton SPAM group for providing us with the source code of SPAM compiler. Santosh Pande's work was partially supported by DARPA Contract Number ARMY DABT63-97-C-0029 and NSF Grant Number CCR-9767129. J. Ramanujam was supported in part by an NSF Young Investigator Award CCR-9457768.

References

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers-Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] S. Anantharaman and Santosh Pande. An efficient data partitioning method for limited memory embedded systems. In *ACM SIGPLAN Workshop on Languauges, Compilation and Tools for Embedded Systems*, pages 205–218, June 1998.

[3] G. Araujo. *Code Generation Algorithms for Digital Signal Processors*. PhD thesis, EE Dept., Princeton University, June 1997.

[4] G. Araujo, S. Devadas, S. Liao K. Keutzer, S. Malik, A. Sudarsanam, S. Tjiang, and A. Wang. *Challenges in code generation fo embedded processors*, pages 48–64.

[5] G. Araujo and S. Malik. Optimal Code Generation for Embedded Memory Non Homogeneous Register Architectures. In *Proceedings of International Symposium on System Synthesis*, 1995.

[6] G. Araujo, S. Malik, and M. Lee. Using Register-Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures. In *ACM IEEE Design Automation Conference(DAC)*, number 33, pages 591–596, June 1996.

[7] R. L. Bowman, E. J. Ratliff, and D. B. Whalley. Decreasing process memory requirements by overlapping program portions. In *Hawaii International Conference on System Sciences*, pages 115–124, 1998.

[8] K. D. Cooper and P. Schielke. Non-local instruction scheduling with limited code growth. In *ACM SIGPLAN Workshop on Languauges, Compilation and Tools for Embedded Systems*, pages 193–207, June 1998.

[9] Keith D. Cooper and Timothy J. Harvey. Compiler-controlled memory. In *8th ASPLOS*, October 1998.

[10] Jack W. Davidson and Sanjay Jinturkar. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. In *MICRO-28:28th Annual International Symposium on Microarchitecture*, December 1995.

[11] Srinivas Devadas, Stan Y. Liao, and Kurt Keutzer. Code density optimization for embedded DSP processors using data compression techniques. In *Adavanced Research in VLSI*, 1995.

[12] Kemal Ebcioglu, Randy D. Groves, Ki-Chang Kim, Gabriel M. Silberman, and Issac Ziv. VLIW Compilation Techniques in a Superscalar Environment. In *Programming Languages Design and Implementation*, 1994.

[13] J. Eyre and J. Bier. DSP Processors Hit the Mainstream. *IEEE COMPUTER*, 31(8):51–59, August 1998.

[14] J. A. Fisher. Trace Scheduling : A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C–30(7):478–490, July 1981.

[15] Christopher W. Fraser, Steven Lucco, Jens Ernst, William Evans, and Todd A. Code compression. In *Programming Languages Design and Implementation*, 1997.

[16] S. Freudenberger, T. Gross, and P. G. Lowney. Acm transactions on programming languages and systems. In *Avoidance and suppression of compensation code in a trace scheduling compiler*, volume 16(4), pages 1156–1214, July 1994.

[17] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H.Freeman and Co, 1979.

[18] R. Gerber, S. Hong, and M. Saksena. Guaranteeing Real-Time Requirements with Resource-Based Calibration of Periodic Processes. In *IEEE Transactions on Software Engineering*, pages 579–592, July 1995.

[19] SPAM Research Group. *SPAM Compiler User's Manual.* http://www.ee.princeton.edu/spam, September 1997.

[20] R. Gupta and M. L. Soffa. Region Scheduling: An Approach for Detecting and Distributing Parallelism. In *IEEE Transactions on Software Engineering*, volume 16(4), pages 421–431, April 1990.

[21] Texas Instruments. http://www.ti.com/sc/docs/dsps/literatu.htm, 1996.

[22] David J. Kolson, Alexandru Nicolau, Nikil Dutt, and Ken Kennedy. Optimal register assignment to loops for embedded code generation. In *ACM Transactions on Design Automation of Electronic Systems*, pages 251–279, April 1996.

[23] D. Landskov, S. Davidson, B.D. Shriver, and P.W. Mallett. Local Microcode Compaction Techniques. *ACM Computing Surveys*, pages 261–294, September 1980.

[24] Stan Liao, Srinivas Devadas, Kurt Keutzer, and Steve Tjiang. Instruction selection using binate covering for code size optimization. In *International Conference on Computer-Aided Design*, 1995.

[25] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steve Tjiang, and Albert Wang. Storage assignment to decrease code size. In *ACM-SIGPLAN Languages, Compilers and Tools for Real Time System*, 1995.

[26] Wen-Yen Lin, Corina G. Lee, and Paul Chow. An Optimizing Compiler for TMS320C25 DSP Chip. In *International Comference on Signal Processing Applications and Technology*, number 5th, pages I–689 – I–694, October 1994.

[27] Y. A. Liu and Gustavo Gomez. Automatic accurate time-bound analysis for high-level languages. In *ACM SIGPLAN Workshop on Languauges, Compilation and Tools for Embedded Systems*, pages 31–40, June 1998.

[28] Peter Marwedel. *Code generation for embedded processors-an introduction*, pages 14–31.

[29] Wen mei W.Hwu, Scott A. Mahlke, William Y. Chen, Pohua P.Cheng, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellete, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. In *Kluwer Publishing*, 1993.

[30] Tarun Nakra, Rajiv Gupta, and Mary Lou Soffa. Value Prediction in VLIW Machines. In *International Symposium on Computer Architecture*, 1999.

[31] Alex Nicolau, P.Panda, and N. Dutt. Memory Organization for Improved Data Cache Performance in Embedded Processor. In *International Symposium on System Synthesis*, pages 384–409, November 1996.

[32] P. Raghavan, M. Charikar, R. Motwani, and C. Silverstein. Constrained TSP and Low-Power Computing. http://theory.stanford.edu/ rajeev/papers.html.

[33] A. Rao and S. Pande. Storage Assignment Optimizations to Generate Compact and Efficient Code on Embedded DSPs. In *Programming Languages Design and Implementation*, pages 128–138, 1999.

[34] Salvatore Ricciardelli, Lucio Bianco, Aristide Mingozzi, and Massimo Spadoni. Exact and heuristic procedures for the traveling salesman problem with precedence constraints, based on dynamic programming. Manuscript.

[35] M. Schlett. Trends inEmbedded Microprocessor Design. *IEEE COMPUTER*, 31(8):44–49, August 1998.

[36] The Stanford SUIF Compiler Group, http://suif.stanford.edu. *The SUIF Library*.

[37] A. Sudarsanam, S. Liao, and S. Devadas. Analysis and evaluation of address arithmatic capabilities in custom DSP architectures. In *Proc. Design Automation Conference*, 1997.

[38] Texas Instruments. *TMS320C2x User's Guide*, 1993.

[39] S. Tjian. An olive twig. Technical report, Synopsys Inc, 1993.

[40] V. Zivojnovic, J. Velarde, and C. Schlager. Dspstone: A dsp-oriented benchmarking methodology. In *Proceedings of the 5th International Conference on Signal Processing Applications and Technology*, October 1994.