View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarBank@NUS

AUTOMATED APPLICATION-SPECIFIC INSTRUCTION SET GENERATION

XU CE

(M.Eng, NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF ENGINEERING

(ACCELERATED MASTER PROGRAM)

DEPARTMENT OF ELECTRICAL AND

COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgements

Pursuing a master degree by research is a difficult journey. The shortened candidature period as a consequence of the accelerated master program (AMP) makes the journey even tougher. I would like to express my thankfulness to all those who have assisted me along the way. Without these help I could not have made the journey through.

I would like to dedicate this dissertation to my parents. I am so thankful to their unconditional love and support, from the first day I left home and started my own journey.

I would like to thank my supervisor, Prof. Tay Teng Tiow, for his patience, guidance, and inspiring advices. I am most grateful that Prof. Tay not only allowed me the complete freedom to experience my research, but also provided constructive suggestions through weekly discussions.

I would also like to thank my colleagues, Xia Xiaoxin, Zhao Ming, Pan Yan, and many more, for sharing their information and knowledge with me.

Last but not least, I thank my girlfriend, for her sustained understanding and support along the way. Especially during the last few weeks before the deadline, she has been taking care of my living with all her love.

Table of Contents

Table of Contents **1**
List of Tables **3**
List of Figures **4**
Abstract **7**
Chapter 1: Introduction **8**
1.1 Related Work 10
1.1.1 Identification 11
1.1.2 Selection 12
1.1.3 Mapping 13
1.2 Thesis Contribution 14
1.3 Thesis Organization 17
Chapter 2: Trace generation and DFG construction **18**
2.1 Introduction 18
2.2 Data Flow Graph generation 19
2.3 MISO & MIMO patterns 23
Chapter 3: Pattern Enumeration **27**
3.1 Introduction 27
3.2 Region and Pattern 28
3.3 Upward cone and downward cone patterns 30
3.4 Pattern enumeration by cone extension 34
3.5 On the complexity of the enumeration algorithm 36
Chapter 4: Pattern Selection **39**
4.1 Introduction 39
4.2 Adjacency matrix representation of graphs 39
4.3 Canonical Label and the nauty package 41
4.4 Complete pattern representation 42
4.5 Hash key generation 44
4.6 Instance list 45
4.7 Software latency, hardware latency and speedup 47
4.8 Optimal custom instruction selection: ILP formulation 49
4.9 Custom instruction selection: greedy algorithm 51
4.10 Maximally achievable speedup as the priority function 52
4.11 Branch-and-Bound algorithm 54
4.12 Conclusion 62
Chapter 5: Application Mapping **64**
5.1 Introduction 64
5.2 Sub-graph isomorphism 64
5.2.1 Ullmann's graph isomorphism algorithm 66
5.2.2 Pruning strategies 71
5.2.3 Convexity checking 79
5.3 Optimal instruction cover 81

5.3.1 Problem formation 81
5.3.2 Pre-processing 83
5.3.3 Heuristically search for an initial solution. 84
5.3.4 Lower bound calculation 85
5.3.5 Sub-problem formation 88
5.3.6 The branch-and-bound algorithm for optimal cover 89
5.4 Code emission 90
5.5 Conclusion 91
Chapter 6: Experimental Results 92
6.1 Environment, libraries and third-party packages 92
6.2 Benchmark programs 92
6.3 Speedup ratio calculation 94
6.4 The effects of input output constraints 94
6.4.1 Input constraint 98
6.4.2 Output constraint 98
6.5 Effects of number of custom instructions 99
6.6 Cross-application mapping 100
6.7 Case study: H.264/AVC encoder 104
6.8 Conclusion 109
Chapter 7: Conclusion 111
Bibliography 114
Appendix 120
Appendix A 120

List of Tables

Table 1: disassembled basic block from “sha” benchmakr............................22

Table 2: content of the creator table...22

Table 3: Instance lists examples...47

Table 4: Software and hardware latency models of common operations 48

Table 5: List of benchmark programs 93

Table 6: the list of cross-compilations 101

Table 7: H.264 building blocks, function names and address range............106

Table 8: the simulation results for H.264/AVC..107

List of Figures

Figure 1: The structure of the automated hardware compiler system............10

Figure 2: pseudo code for DFG construction..21

Figure 3: the constructed DFG..23

Figure 4: Simplified DFG by omitting inputs and grouping similar instructions..25

Figure 5: MISO and MIMO patterns 26

Figure 6: basic blocks can be separated into disjoint regions........................29

Figure 7: Upward cone generation..32

Figure 8: Overlapped upward cones results in repeated patterns 33

Figure 9: Part of a DFG from rijndael benchmark. All nodes are "+" instructions..38

Figure 10: Equivalent graphs have different adjacency matrix representations 40

Figure 11: The setword representation of adjacency matrix..........................43

Figure 12: The complete representation of a pattern graph 44

Figure 13: Pattern instances that are overlapping..46

Figure 14: the greedy algorithm on pattern selection 52

Figure 15: Maximum achievable frequency: the pattern T and instances C1-C7..59

Figure 16: The binary search tree associated with the example in figure 15. 61

Figure 17: Algorithm that calculates the priority of each pattern..................62

Figure 18: the output constraints that must be satisfied for custom instruction matching..70

Figure 19: sub-graph isomorphism without pruning 71

Figure 20: the refinement procedure..73

Figure 21: the library graph and subject graph and the initial permutation matrix. 74

Figure 22: Pruning of binary search tree..77

Figure 23: sub-graph isomorphism that violates the convexity constrain 79

Figure 24: the complete sub-graph isomorphism algorithm..........................80

Figure 25: Cover matrix and pre-processing 83

Figure 26: The algorithm to find the initial cover..85

Figure 27: the greedy algorithm that finds an independent subset of the rows X...88

Figure 28: the branch-and-bound algorithm that finds the optimal cover 90

Figure 29 : dijistra: speed up vs. different input-output constrains. 95

Figure 30: patricia: speed up vs. different input-output constrains. 95

Figure 31: FFT: speed up vs. different input-output constrains....................95

Figure 32: crc: speed up vs. different input-output constrains.......................96

Figure 33 : sha: speed up vs. different input-output constrains....................96

Figure 34 : rawcaudio: speed up vs. different input-output constrains..........96

Figure 35: rawdaudio: speed up vs. different input-output constrains...........97

Figure 36: bitcnts: speed up vs. different input-output constrains................97

Figure 37: basicmath: speed up vs. different input-output constrains. 97

Figure 38: effects of custom instruction set size...100

Figure 39: Speedup ratios of selected cross-compilation 1 102

Figure 40: Speedup ratios of selected cross-compilation 2 102

Figure 41: Basic coding structure for H.264/AVC for a macroblock. 104

Figure 42: Four most popular patterns for DCT and Quantization 107

Figure 43: Four most popular patterns for Motion Estimation 108

Figure 44: Four most popular patterns for Motion Compensation 108

Figure 45: Four most popular patterns for Debloking Filter 108

Figure46: Four most popular patterns for Arithmetic Coding (cabac) 109

Abstract

Large complex embedded applications require high performance embedded processors to complete the tasks. While traditional DSP processors are difficult to meet these stringent demands, extensible instruction-set processors are shown to be effective. However, the performance of such reconfigurable processors relies on successfully finding the critical custom instruction set. To reduce this intensive task which is traditionally performed by experts, an automated custom instruction generation system is developed in this research.

The proposed system first explores the application's data flow graph and generates all valid custom instruction candidates, subjected to pre-configured resource constraints. Next a custom instruction set is selected using a greedy algorithm, guided by intelligent speedup estimation of each candidate. Finally, the system optimally maps any given application onto the newly generated custom instruction set.

The MiBench benchmark is used to study the effects on speedup ratios by varying input-output constraints, custom instruction set size and cross-application compilation. A case study on H.264/AVC is performed and results are presented. Experiments show the proposed system is able to identify the critical patterns and almost all applications can benefit from custom instructions, achieving 15%-70% speedup.

Chapter 1: Introduction

In the last three decades, the performance of traditional general purpose microprocessors has been improving by taking advantage of advanced silicon technology and architectural improvements such as pipelining and media instruction extension (e.g. MMX, SSI), etc. However, fast growth in consumer electronics market demands stringent properties including low power consumption and high performance, which conventional general purpose microprocessors are difficult to meet. Digital Signal Processor (DSP), driven by the market force, appeared in the early 80's and has become popular since ever. DSPs achieve high performance in certain niche application areas by introducing additional function units such as adder, multiply-accumulator (MAC), etc, as a new architectural choice. DSPs have been successfully applied to numerous application domains, including mobile phones, routers, voice-band modems, etc. However, there are many new emerging areas such as portable multimedia communication device, personal digital assistants (PDAs), which are difficult to apply standard DSP architectures. In the last decade, System-on-Chip (SOC) processors gain full attention as these processors are specifically designed for target applications, hence achieving better performance-cost ratio. At the early stage of this application-specific instruction set processors (ASIPs) approach, the practice is to re-design the complete processor structure. The major drawback of this approach is the complexity of redesigning the entire instruction set and its associated development toolset. As the market is changing rapidly, fast re-design turnaround

time is desired, thus limiting the use of ASIPs in SOCs. Recently, the focus has been shifted to configurable or extensible instruction set microprocessors, which offer a tradeoff between efficiency and design flexibility. These processors typically contain one standard core processor with tightly coupled hardware resources that can be customized. The goal is to configure the custom data-path to optimize towards specific applications, subjected to the area and latency constrains.

Sophisticated extensible processors such as Xtensa [11] from Tensilica release the designer's burden by providing a set of development tools. However, it has been a common practice that an expert is needed to find out the custom data-path. The expert must fully understand the application and the available resources provided by the extensible processor. The task becomes complicated when the application software is large. Moreover, design constrains such as die area, clock frequency limit, number of available read-write ports, etc, further complicate the problem.

In this research work, we propose a methodology that automatically detects and selects custom instruction candidates to achieve optimal or sub-optimal speed up for a given application. After the library patterns are generated, the automation algorithm takes another instance of the application software (may or may not be the same software model as the one used for library generation) and detect all possible instruction clusters that match a custom library pattern. Finally the

automation algorithm generates the optimal code that makes the best use of library patterns. The complete program flow is shown in Figure 1 below. In Figure 1, if application program 1 is the same as application program 2, it is called native compilation; otherwise it is called cross-compilation.

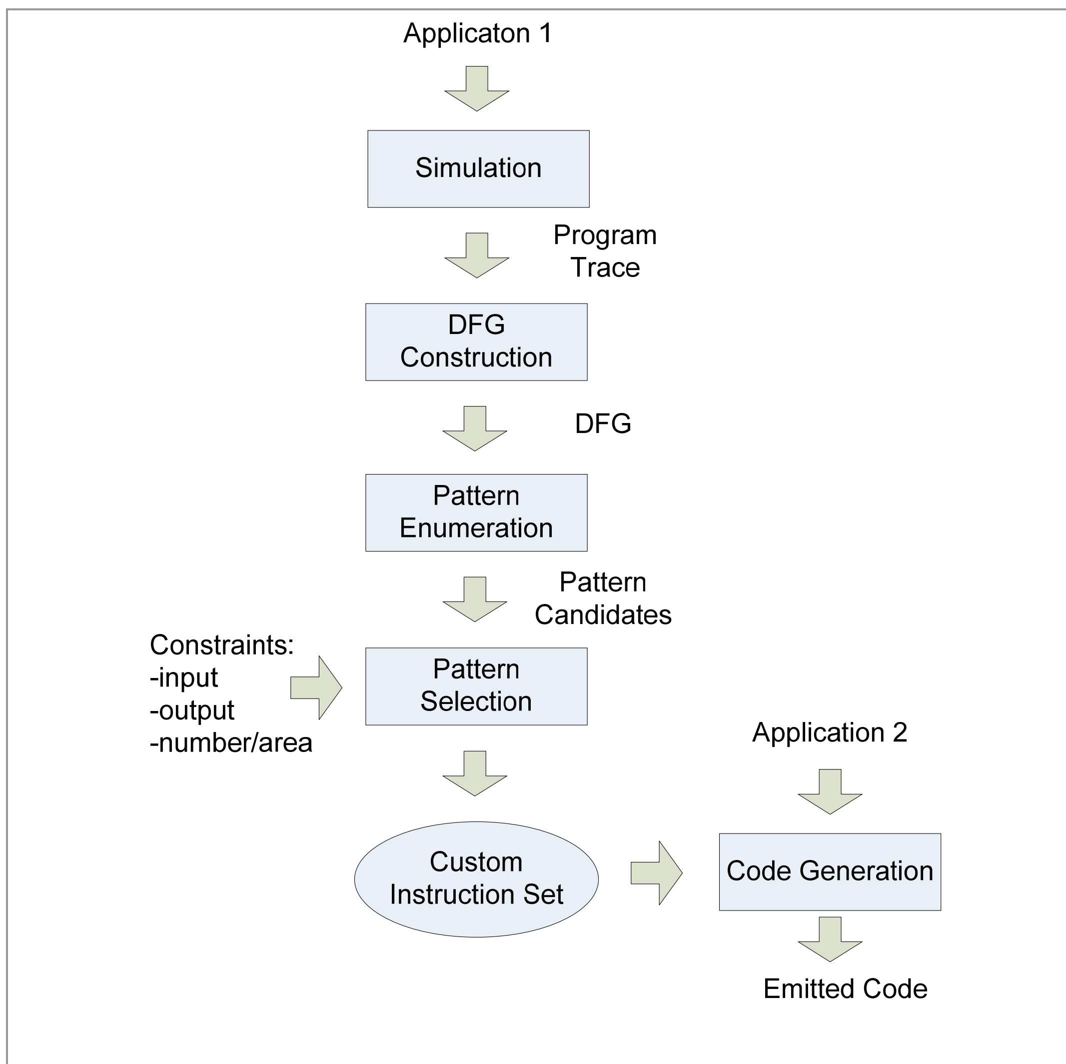


Figure 1: The structure of the automated hardware compiler system

1.1 Related Work

We provide an overview of the related work done in this field. Application specific custom instructions have been extensively studied before. The complete

system in general can be partitioned into three stages: identification, selection and mapping.

1.1.1 Identification

In the first step, the target application's data-flow graph (DFG), usually on a basic block basis, is generated and pattern candidates are picked up by looking at the sub-graphs of the DFG. Complete sub-graph enumeration, however, is exponential to the total number of nodes in the DFG. Many works try to by-pass this problem by heuristically explore a subset of the design space. In works of Sun et. al[4] and Nathan et. al[26], patterns grow from selected seeds and a heuristic guide function is used to limit the growth. In Cong's work [5], only cone-type or multiple-input-single-output (MISO) type patterns are considered. Atasu, et. al [1], on the other hand, exhaustively generate all possible patterns including disjoint patterns. They applied simple pruning strategies to limit the search space exploration. Pan et. al [29] proposed an improved algorithm to generate all feasible connected patterns by extending cone-type patterns into multiple-input-multiple-output (MIMO) type patterns.

Typically the custom instructions can be classified according to execution cycles, input-output constrains, connectivity and whether overlapped patterns are allowed.

Execution Cycles: In early works such as Huang et. al [14], only single cycle

complex instructions are generated. Choi et. al [3] extended to multi-cycle complex instructions but they put an artificial limit on critical path length. Recent works almost all focus on multi-cycle instructions as these instructions in general offer more potential for speedups.

Input-Output constraints: The core processor register file has limited read and write ports, hence it is apparent to apply input output constraints during custom instruction generation. Moreover, these constraints can be effectively used to prune the search tree.

Connectivity: In most works [4], [5], [29], only connected patterns are generated. However, in [3], instructions are first packed in parallel and then grow in depth. They applied subset-sum solver to generate custom instructions. The problem is that the effectiveness of parallel and depth combination is not well known. The exhaustive enumeration in [1] also combines disjoint patterns together to form large patterns.

Overlap: Although patterns in general do not overlap in the final code, it is important to generate all overlapped patterns so as no to artificially constrain the pattern selection stage.

1.1.2 Selection

In the pattern selection stage, the goal is to choose an optimal set of custom instructions out of a large pool of generated patterns, subjected to system constraints such as die area or number of custom instructions. If overlapping patterns are allowed, as what is in [4], pattern selection can be formulated as 0/1 knapsack problem. However, if overlapping patterns are not allowed, then the 0/1 knapsack formulation would contain dynamic values, since selecting one pattern causes the values of overlapping patterns to change. An ILP formulation can be set up to find the optimal custom instruction set [26]. However, in many cases heuristic-based method is preferred as the search space is often unacceptably large for ILP-based approach, especially for large programs. In [4] a simple greedy algorithm is used to select the patterns, taking the overlapping into consideration.

1.1.3 Mapping

Most previous work, however, did not consider application mapping, but simply placed the selected custom instructions in the code immediately after instruction generation and selection, to calculate performance gain [26], [30]. Similarly, Cong et. al [4] did not consider custom instruction matching, but they used binate covering method to address optimal code generation. In the software-hardware co-design context, the application to be run on the custom processor may be frequently modified and updated, and it can even be different applications in the same domain. It is necessary to derive a methodology that properly map any given application onto the custom instruction set.

1.2 Thesis Contribution

This work presents a complete framework to address customer instruction set design and application mapping.

In Chapter 4, we proposed an innovative algorithm to calculate the maximally achievable speedup of each pattern candidate. Given the speedup and total frequency of a pattern candidate, the maximally achievable speedup of this candidate is not simply the product of those two numbers. In practice, not all instances of a candidate can be realized simultaneously because instances can be overlapping. Due to the large number of instances, standard binary search algorithm is not practical. We formulate the problem of finding the maximally achievable speedup of each candidate as a parallel branch-and-bound algorithm. The entire instance list of the candidate is partitioned into disjoint groups such that instances from different groups never overlap. Branch-and-bound algorithm is applied to each individual group and the results are summed to get the actual potential speedup. This strategy effectively transforms the initial problem into sub problems that can be easily tackled.

In Chapter 5, we presented our 2-pass solution to application mapping and code generation problem, which was rarely addressed before due to its complications. After the custom instruction set is selected, the last step of our system is to map the application onto the union of the core processor's basic instruction set and the

newly selected custom instruction set. This is done in a two-pass process. The first pass is library matching: the DFG is constructed for each basic block and it is checked against the custom instruction library to find any possible utilization of those custom instructions. The second pass is optimal code generation: the optimal DFG cover using both custom instructions and core processor instructions is selected.

Code generation against custom instruction set in general is a non-trivial problem, and traditional approaches are to break the DFG into forest (disjoint trees) and perform tree pattern matching against the instruction set. Although in this method the optimality of the generated code is heavily dependent on the partitioning method, in practice it is widely adopted in compiler design due to its attractive complexity. The incentive behind is that tree matching can be easily converted to string matching and linear time string matching automaton is readily available. Unfortunately, this method cannot be applied to a custom instruction set which contains arbitrary complex instruction patterns. In our system, the custom instructions are not limited to tree patterns; in fact, they are directed acyclic graphs (DAG). The matching problem is essentially a sub-graph isomorphism problem from each custom instruction to the subject DFG. It is known that sub-graph isomorphism of digraphs is as difficult as that of regular graphs and the latter is NP-Hard [10]. Nevertheless, in the case of instruction matching there are two constraints that greatly reduce the theoretical exponential search space. The

first constraint is that both DFG and custom instructions are acyclic graphs. The second constraint is that for a match to be valid, each matched node pairs in the subject graph and the library graph must be the same operation type. Ullmann [27] proposed a general graph matching algorithm which travels in a depth first manner in the search space. The algorithm achieves attractive runtime by applying a refinement procedure at each search node, despite that the worst case is still exponential to the number of nodes in the subject graph. We use Ullmann's algorithm as a basis and added additional refinement steps to further reduce the run-time complexity.

After the matches are detected, it still remains a problem to optimally select a subset from all the matches such that every instruction in the subject graph is covered and the total execution latency is minimized. It is well known that such optimal DAG covering is a NP-hard problem. However, in practice, the custom instruction set size is limited due to resource constrains, unless for huge basic blocks (over a few hundred instructions), there are hopes for efficient algorithms that find the optimal covering. In our systems, we implemented a branch-and-bound (bnb) algorithm to perform instruction covering. To reduce the runtime complexity, the pruning techniques proposed by Coudert and Madre [8] are applied. In addition, the custom instructions do not overlap, and can be used as another pruning constraint to greatly reduce the search space.

1.3 Thesis Organization

The thesis is organized as follows. Chapter 2 discusses application trace generation and DFG construction. Chapter 3 describes the pattern enumeration algorithm. Chapter 4 provides a detailed description on pattern selection, including the data structure for pattern representation, the speedup estimation and the custom instruction selection algorithm. Chapter 5 introduces Ullmann's graph isomorphism algorithm and how it is incorporated into our branch-and-bound algorithm to solve the code generation problem. Chapter 6 presents the experiment results. Chapter 7 gives the conclusion and the direction for future work.

Chapter 2: Trace generation and DFG construction

2.1 Introduction

In this work, the core processor is assumed to be RISC-like and the ISA is similar to the MIPS [23] instruction set. In the MIPS ISA, instructions are classified into the following major categories: memory, integer computation, floating point computation, and control instructions. In this context, integer computation instructions are of particular interests to be implemented in custom hardware logics. Floating point instructions, on the other hand, are not very popular due to the fact that in most applications they take a small fraction only. Another reason is float-point instructions usually span multiple clock cycles, which makes it difficult to be put in custom hardware.

Integer instructions are further classified into operation types: addition, subtraction, multiplication, division, shift, logic, etc. The latencies for those instructions are assumed to be 1 except for division, which is assumed to be 10.

We use the SimpleScalar [2] PISA toolset as the framework. SimpleScalar is a popular simulation package which comes with compiler, assembler, debugger and simulator. Moreover, new simulators can be crafted without much difficulty. The SimpleScalar PISA ISA is compatible with the MIPS IV ISA; hence it provides a

good working environment for our system.

The target application is assumed to come with a standard reference software model; examples are Momusys for MPEG-4 and JM for H.264/AVC, etc. The software model is compiled to the SimpleScalar architecture and it is simulated using a modified fast simulator with standard input dataset. The simulator is crafted to record both static and dynamic information of the software model. Static information includes program text symbols and their associated address range; each basic block's starting address, instructions, and size. Dynamic information mainly contains the run-time accessing count of each basic block.

2.2 Data Flow Graph generation

Definition 1: *source, sink, forward-dependency*

If instruction i updates register $\$r$ and instruction j uses $\$r$ as one of its inputs later, we say instruction i is the source of instruction j, and instruction j is the sink of instruction i. There is a forward dependency from instruction i to j.

The selected basic blocks are represented in Data Flow Graphs. The DFG $G(V, E)$ represents the relationship, more specifically the inter-dependency, among the instructions in a basic block. Each instruction is represented as a node $v \in V$ in the DFG and the edge $e: u \rightarrow v$ represents that there is a forward dependency from node u to node v. In other words, the output of the instruction

represented by node u is one of the inputs of the instruction represented by node v. A DFG is necessarily a directed acyclic graph (DAG). A DFG is a parameterized graph: it stores the instruction type at each node, but there is no parameter associated with the edges. In this work, we use a node array L of size $|G|$ to represent the node parameter, for instance $L[v]$ is the instruction type associated with node v. As mentioned before, there are constraints on instruction types for custom hardware. Those that can be included into the custom hardware are called valid operations and all others are called invalid operations.

$$\text{Valid operations: } \{add, sub, mul, div, shift, logic, lui, slt\}$$

$$\text{Invalid operations: } \{load, store, branch, float, etc...\}$$

Since invalid operations are not taken into consideration for custom instructions, we label them as belong to one class "invalid". To conclude, the operation type associated with each node is one of the following:

$$\{add, sub, mul, div, shift, logic, lui, slt, invalid\}.$$

To create the DFG, we maintain a register value creator table to record which instruction is the last modifier of each register. In the MIPS compatible architectures, there are 32 general registers and 32 floating point registers. The floating point registers are ignored in this case. Each MIPS instruction at most takes 3 registers as inputs and updates up to 2 registers as outputs.

We scan through the basic block and add one node to the DFG for each instruction.

We check the input registers, if the corresponding creation table for that register is not empty, there is a dependency from the creator to the current instruction and we add one new edge in the DFG accordingly. The outputs of current instruction are used to update the creation table. The algorithm that builds the complete DFG is shown in Figure 2 below:

```
<DFG_construction>
01 Graph G = empty Graph;
02 for instruction i = 1,2,...n
03     node v = G.add_node(op_type(i));
04     for input reg. j = 1,2,3
05         if creater(j) ≠ 0 then
06             G.add_edge(creater(j), v);
07         end
08     end
09     for output reg. j = 1,2
10         creater(j) = v;
11     end
12 end
```

Figure 2: Pseudo code for DFG construction

Table 1 shows a disassembled basic block from MiBench's [13] "sha" benchmark. Table 2 shows the content of the register value creator table and how it changes as instructions are processed. Finally, Figure 3 shows the initially constructed DFG. The label beside each node is the instruction number same as that of table 1 and the label inside the node is the instruction type. The inputs with "$" prefix are registers and the inputs with "#" prefix are immediate values. It is worth noting that the DFG is not necessarily connected, as a matter of fact, it often consists of a few connected components and singular nodes. In this example, there are three

connected components and four singular nodes:

$$\{\{1,2,3,4,5,6,7,13,16,17\},\{10,11,12\},\{15,19,20\},\{8\},\{9\},\{14\},\{18\}\}.$$

Table 1: disassembled basic block from "sha" benchmark

Basic Block 280		
1	sll	r3,r10,5
2	srl	r2,r10,27
3	or	r3,r3,r2
4	xor	r2,r8,r7
5	xor	r2,r2,r11
6	addu	r3,r3,r2
7	addu	r3,r3,r12
8	addu	r12,r0,r11
9	addu	r11,r0,r7
10	sll	r7,r8,30
11	srl	r2,r8,2
12	or	r7,r7,r2
13	lw	r2,0(r4)
14	addu	r8,r0,r10
15	addiu	r9,r9,1
16	addu	r3,r3,r2
17	addu	r10,r3,r5
18	addiu	r4,r4,4
19	slti	r2,r9,40
20	bne	r2,r0,0xffffff68

Table 2: content of the creator table

Registers	Creator Instructions
r0	
r2	$2 \rightarrow 4 \rightarrow 5 \rightarrow 11 \rightarrow 13 \rightarrow 19$
r3	$1 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 16$
r4	18
r5	
r7	$10 \rightarrow 12$
r8	14
r9	15
r10	17
r11	9
r12	8

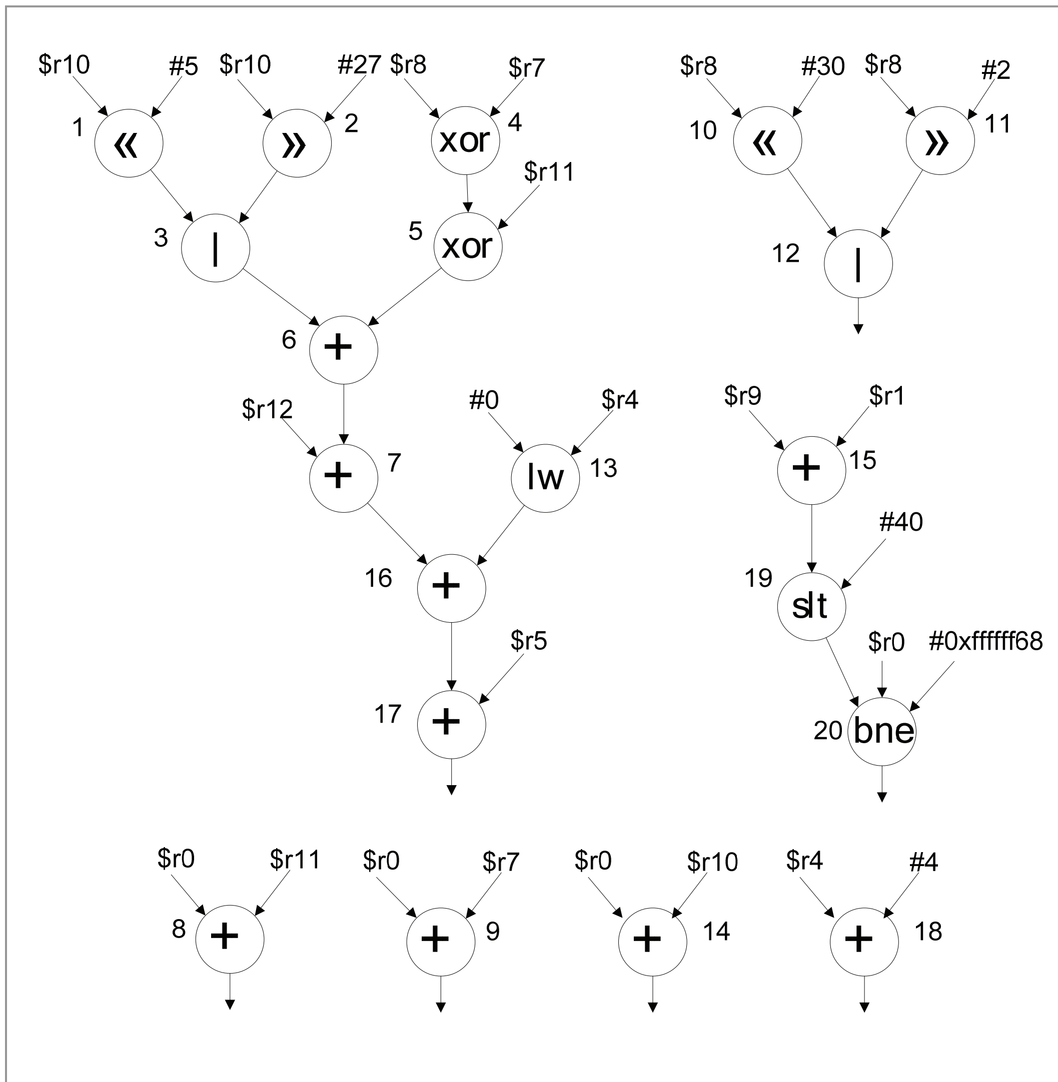


Figure 3: The constructed DFG

2.3 MISO & MIMO patterns

Definition 2: *pattern*

A pattern $P(V', E')$ is a sub-graph of the DFG, such that

$$\begin{aligned} &V' \subseteq V \\ &E' = (V' \times V') \cap E \\ &L(v) = L(v) \quad if \quad v \in V' \end{aligned} .$$

In this work, only connected patterns are considered. Each instruction itself is a special type pattern called "trivial pattern". Each pattern has incoming edges and

outgoing edges. The set of nodes in P that are connected to incoming edges are called input nodes. Similarly, the set of nodes in P that are connected to outgoing edges are called output nodes.

For pattern generation, the exact register and immediate inputs to each node can be omitted in the DFG representation. The rationale behind is that register and immediate inputs are dynamically allocated by the compiler and these information are not needed for custom instruction generation.

In addition, in this work, we assume similar instructions can be executed in one piece of custom hardware. For example, all logic operations, including *and, or, nor,* and *xor*, can be implemented on a logic hardware unit. We assume the specific operation is encoded as signature bits in the custom instruction format and it can be recognized by the custom hardware automatically. Similarly, a shift unit is able to perform *left shift, right shift, left shift arithmetic* and *right shift arithmetic*. However, *add* and *sub* are treated differently, although in some practical systems it might be desirable to group them onto a single custom hardware. Figure 4 shows a simplified DFG derived from the one in Figure 3.

Definition 3: *MISO and MIMO pattern*

MISO patterns are patterns that contain exactly one output node. Conversely, MIMO patterns contain at least two output nodes.

Examples of MISO and MIMO patterns are shown in figure 5. Figure 5(a) shows a MISO pattern with 4 inputs and 1 output node; Figure 5(b) shows a MIMO pattern with 4 inputs and 2 output nodes.

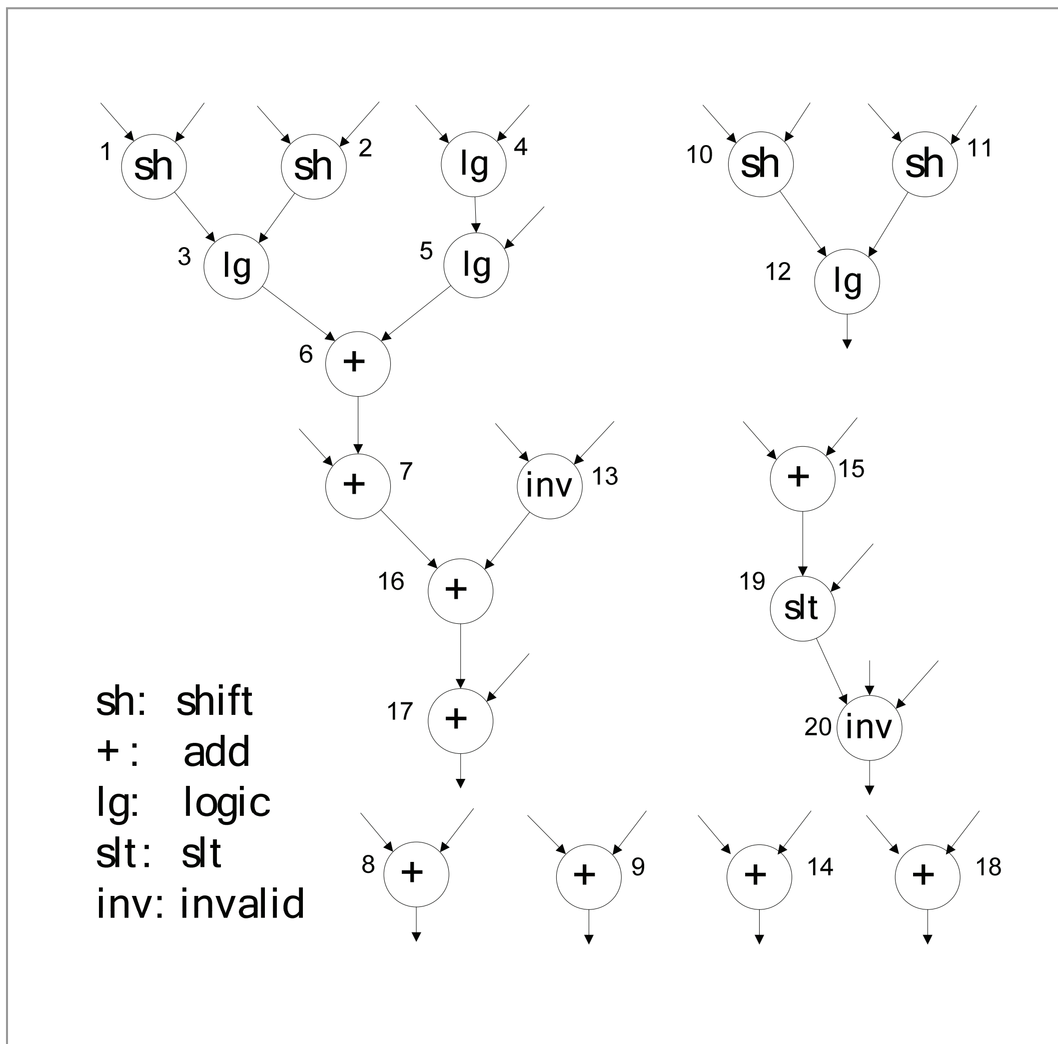


Figure 4: Simplified DFG by omitting inputs and grouping similar instructions

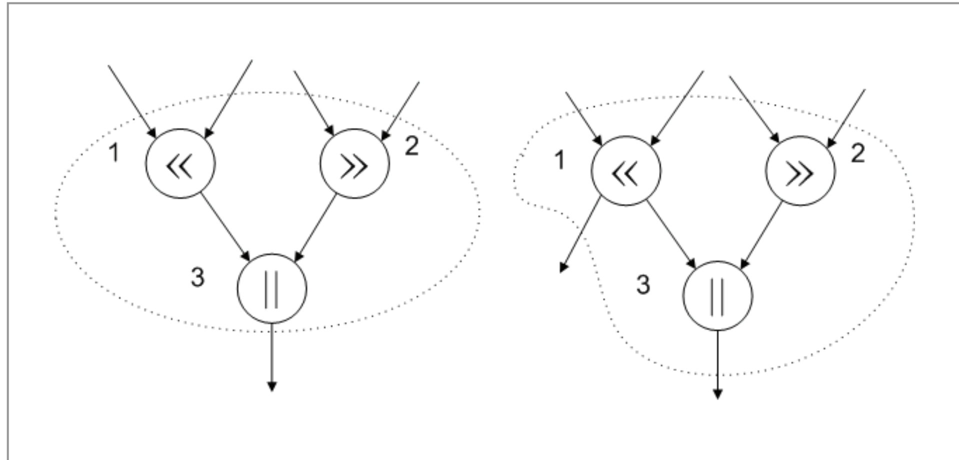


Figure 5: MISO and MIMO patterns

In this work, the number of inputs (not input nodes) and the number of output nodes are used for hardware constraint checking.

Chapter 3: Pattern Enumeration

3.1 Introduction

To provide sufficient information for later stages, all possible patterns in a DFG should be enumerated. However, theoretically the complexity of enumerating all patterns is proportional to 2^N, where N is the total number of nodes in the DFG. To bypass this difficulty, works such as [4], [26] generate a subset of all possible patterns. Although these approaches are attractive in practical implementations when efficiency is an important concern, the optimality is not guaranteed. Moreover, it is apparent to have a system that generates all patterns so that the performance of those heuristic methods can be evaluated. In Atasu's work, all possible patterns that satisfy convexity constrain are generated. However, as no other constrains are imposed, this method is not efficient enough to be applied to large basic blocks. Pan [29] proposed an improved method that generates MIMO patterns by extending cone-type patterns. Their method is attractive because the complexity is proportional to 2^K, where K is the number of extension ports. In practice, the limit of K is closely related to the fan-in/fan-out at each node. As the fan-in at each node is limited to 3 due to the nature of DFGs, usually there is only one case that prevents the use of this complete enumeration method. That is, when there is at least one node have a large number of fan-outs (typically > 20). In other cases, the runtime of the full enumeration method is very much acceptable.

Cong et. al [5] also applied full enumeration method except that in their framework, the custom instructions to be considered are MISO patterns.

3.2 Region and Pattern

In our work, we adopted Pan's algorithm to perform pattern enumeration. The pattern enumeration, however, is not directly performed on the entire DFG. Since invalid nodes are not included into custom instructions, it is very likely that the entire DFG can be partitioned into multiple regions, separated by invalid nodes. It is only necessary to perform pattern enumeration in each region. Region partitioning is a simple yet efficient strategy that helps to reduce the graph size to work on. Here the same definition of region as in [29] is used:

Definition 4: Region

Given a DFG $G(V, E)$, a region $R(V', E')$ is defined as a maximum sub-graph of G such that:

(1) $\forall v \in V'$, v is valid node.

(2) There exists an undirected path between any two nodes in R.

(3) There does not exist any edge between a node $v \in V'$ and another node $u \in V - V'$.

The definition of pattern in previous chapter can be refined to:

A pattern $P(V', E')$ is a sub-graph of a region in a DFG. It is important to note

that not all sub-graphs are valid patterns. A pattern is convex if there exists no path between any two nodes $u, v \in P$ such that the path contains a node $w \notin P$. Patterns that do not satisfy convexity are invalid as there is a circular dependency between the pattern P and the node w. This can be easily understood: on one hand, there is an edge from a node in P to w, thus there is a forward dependency from P to w; on the other hand, there is an edge from w to a node in P, thus there is a forward dependency from w to P.

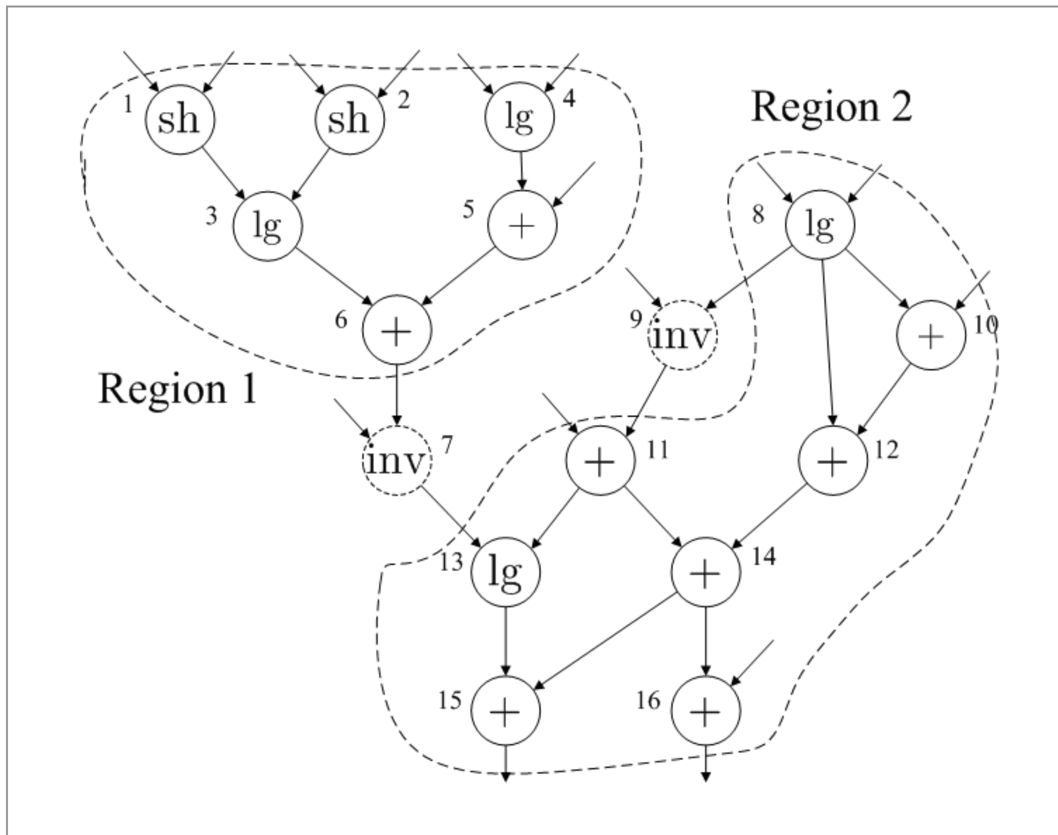


Figure 6: Basic blocks can be separated into disjoint regions

Figure 6 gives an example where a connected DFG is separated into two regions by node 7 and 9. Examples of non-convex patterns are {8,12}

and $\{8,10,11,12,14\}$. In pattern $\{8,12\}$, there is a path from node 8 to node 12 through node 10, which is a valid node but it is not in the pattern. In pattern $\{8,10,11,12,14\}$, the node that causes violation is node 9. It is worth noting that node 9 is an invalid node and it does not belong to any regions.

3.3 Upward cone and downward cone patterns

Two special pattern types are defined:

Definition 5: *Upward Cone, Downward Cone*

Upward cone: The upward cone of node v, denoted as $UC(v)$, is a convex pattern that contains node v, and for all other nodes $u \in UC(v)$, there is a path from u to v. In other words, v is the only sink node in $UC(v)$. Let the set of all upward cones of node v be denoted as $UC_Set(v)$

Downward cone: The downward cone of node v, denoted as $DC(v)$, is a convex pattern that contains node v, and for all other nodes $u \in UC(v)$, there is a path from v to u. In other words, v is the only source node in $DC(v)$. Let the set of all downward cones of node v be denoted as $DC_Set(v)$

Take node 14 in Figure 6 as an example, the set of its upward cones are $\{14\}$, $\{11,14\}$, $\{12,14\}$, $\{11,12,14\}$, $\{10,11,12,14\}$, etc. Similarly, the set of its downward cones are $\{14\}$, $\{14,15\}$, $\{14,16\}$, and $\{14,15,16\}$.

The enumeration algorithm requires the DAG being topologically sorted.

Definition 6: *Topological Sort*

A topological sort of the vertices of G is a linear ordering of the vertices such that for every pair of distinct vertices v_i and v_j , if $v_i \rightarrow v_j$ is an edge in G, i.e., $(v_i, v_j) \in E$, then v_i appears before v_j in the ordering.

It is easy to prove if the order of each node in the DFG is assigned using the corresponding instruction sequence number in the basic block, then this ordering is readily a topological ordering. The same holds even after the DFG is partitioned into regions: the nodes in each region are still topologically ordered except the orders are not continuous.

The enumeration algorithm contains two phases. In the first phase the set of upward and downward cones at each node is identified. To identify the upward cones, the DAG is traversed in topologic order. The set of upward cones at node v can be obtained by selectively union the upward cones of its predecessors and node v itself. Let $v_1, v_2, ..., v_k$ be the predecessors of node v, as the DAG is traversed in topologic order, by the time node v is reached, the set of upward cones of $v_1, v_2, ..., v_k$ are all known. If we pick $i(0 \leq i \leq k)$ predecessors out of k, say $u_1, u_2, ..., u_i = v_1, v_3, ..., u_{k-2}$, and pick one upward cone from each

of $UC_Set(u_1), UC_Set(u_2), ..., UC_Set(u_i)$ and union these upward cones together with node v, the resultant pattern is an upward cone of node v. This can be easily proven: since $u_1, u_2, ..., u_i$ are predecessors of node v, for any node $u \in UC_Set(u_1) \bigcup UC_Set(u_2) \bigcup ... \bigcup UC_Set(u_i)$, there is a path from u to v through one of $u_1, u_2, ..., u_i$.

For example, in Figure 7, the set of upward cones for node 3 and node 5 are $\{\{3\},\{1,3\},\{2,3\},\{1,2,3\}\}$, $\{\{5\},\{4,5\}\}$ respectively. Therefore the set of upward cones for node 6 is the union of the following:

(a) Itself: $\{\{6\}\}$

(b) Select predecessor node 3 only: $\{\{3,6\},\{1,3,6\},\{2,3,6\},\{1,2,3,6\}\}$

(c) Select predecessor node 5 only: $\{\{5,6\},\{4,5,6\}\}$

(d) Select both predecessors:

$\{\{3,5,6\},\{1,3,5,6\},\{2,3,5,6\},\{1,2,3,5,6\},\{3,4,5,6\},\{1,3,4,5,6\},\{2,3,4,5,6\},$
$\{1,2,3,4,5,6\}\}$

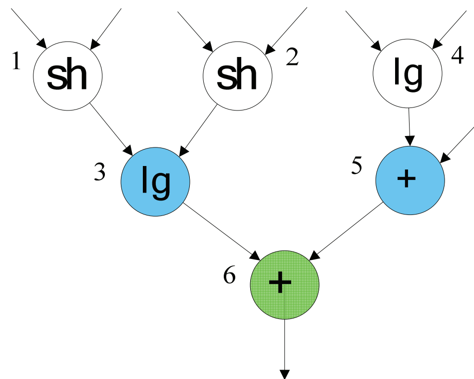


Figure 7: Upward cone generation

However, the above procedure may generate invalid patterns and repeated patterns. For upward cone generation, invalid patterns are those do not satisfy convexity or input constrains. These patterns can not be used for pattern extension and can be eliminated. It is shown in [29] the elimination is safe and it does not prevent any valid patterns to be generated. It is worth noting patterns that do not satisfy output constrains are not eliminated, since those patterns have potential to be extended to valid patterns.

Repeated patterns can be generated if the upward cones of the predecessors overlap. Consider the DAG in Figure 8, the set of upward cones for node 3 and node 4 are $\{\{3\},\{1,3\},\{2,3\},\{1,2,3\}\}$, $\{\{4\},\{1,4\},\{2,4\},\{1,2,4\}\}$ respectively. It is easy to observe union $\{1,3\},\{4\},\{5\}$ or $\{3\},\{1,4\},\{5\}$ results in the same upward cone $\{1,3,4,5\}$ of node 5. Therefore before a generated pattern is added to the upward cone set, it is checked to ensure the upward cone set does not contain duplicates.

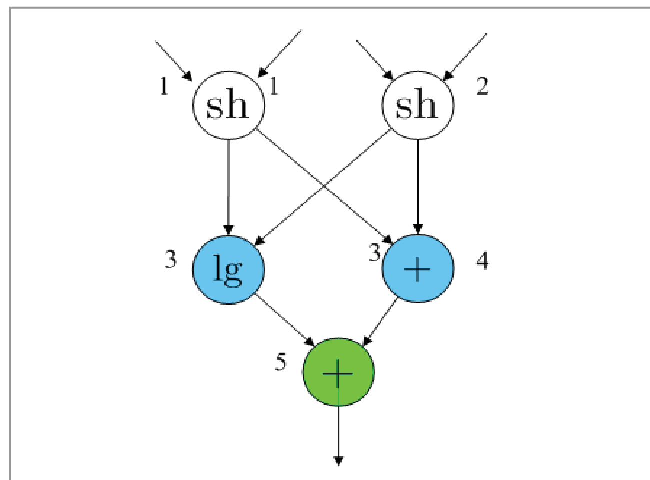


Figure 8: Overlapped upward cones results in repeated patterns

The generation of downward cones is similar to that of upward cones, except that the region DAG is traversed in the reverse topologic order. Moreover, the definition of invalid downward cones is not satisfying convexity constrain or output constrain.

3.4 Pattern enumeration by cone extension

The second phase of pattern enumeration is to extend the cone type patterns to form general shaped patterns. If we choose upward cones as initial pattern, the region DAG is traversed in the reverse topologic order. On the other hand, if we choose downward cones as initial pattern, the DAG should be traversed in topologic order. These two approaches are equivalent and in this work we use the former method. As the DAG is traversed, all the patterns that contain a particular node are generated after that node is visited.

A maximum upward cone (MAX_UC) of node v is defined as the union of all its upward cones. An important property that is associated with the MAX_UC is any upward cones of node v can only be extended along the output nodes of MAX_UC. Those nodes along witch patterns are extended are called extension points.

The pseudo code of pattern enumeration is shown below:

1. For each node v in reverse topological order, its UC_Set is added to the

pattern pool: $Pattern(v) += UC_Set(v);$

2. Find the set of extension points ext by checking $MAX_UC(v)$.
3. If ext is not empty, perform pattern extension:

   $Pattern(v) += UNION(Pattern(v), ext, down);$

The $UNION(core, ext, direction)$ procedure is a recursive routine that extends the set of core patterns through the extension point along the direction specified. If direction=1, the core will be extended downwards and otherwise upwards.

In the UNION procedure, new patterns are generated in a manner similar to that of UC_Set and DC_Set generation. We briefly describe the process below:

1. Find all possible i combinations $(0 \le i \le |ext|)$ of extension points,

say $\mathrm{A} = \{\alpha_1, \alpha_2, ..., \alpha_i\} \subseteq ext$.

2. Selected a subset $P \subseteq core$, such that $A \subseteq P$ and $(ext - A) \bigcap P = \varnothing$;
3. Form a temporary set by cross-product the upward cones or downward cones of the selected extension points:
   a) if direction is downwards, $tmp := DC_Set(\alpha_1) \times ... \times DC_Set(\alpha_i)$;
   b) if direction is upwards, $tmp := UC_Set(\alpha_1) \times ... \times UC_Set(\alpha_i)$;
4. Select one pattern each from P and tmp, generate the new pattern pat_tmp using union operator. If direction is downwards, check convexity and output constrains of pat_tmp. If direction is upwards, check convexity

and input constrains of pat_tmp. Let the set of newly generated patterns being new_core, add the pat_tmp to new_core if it is valid.

5. After all new patterns for current set of extension points are generated, find the extension points new_ext for new_core and recursively call $UNION(new_core, new_ext, \neg direction)$

3.5 On the complexity of the enumeration algorithm

Although the pattern enumeration algorithm is still exponential to the number of nodes in the DAG, its average runtime is a few magnitudes lower than exhaustive enumeration. In practice, we found the runtime is heavily dependent on the DAG structure. More specifically, if the DAG contains some nodes which has a large number of fan-outs, the algorithm would stuck as early as in the downward cone generation phase. Take a simple example, suppose a node generates 20 forward dependencies, which may happen in very large basic blocks (e.g. rijndael from MiBench), the algorithm needs to union all possible combinations of 1, 2, up to 20 successors' DC_Sets. Note even if under the extreme conservative assumption that each DC_Set contains only one pattern, the number of possible combinations is $\binom{20}{0}+\binom{20}{1}+...\binom{20}{20}\approx 100M$. The same problem may cause trouble for pattern extension phase as well.

The generation of UC_Set, however, does not have this problem. This is due to the DFG property that each node has maximally 3 inputs. In fact, for all those

instructions that are valid to be included into custom instructions, i.e. add, sub, mul, div, shift, logic, lui, and slt, each has a fixed number of inputs equal to 2.

Fortunately the exponential enumeration problem for DC_Set and pattern extension may be tackled in most practical applications. Observations from experiments show that a DFG containing nodes with such large number of forward dependencies normally possesses high degree of regularity in its DAG structure. An example in Figure 9 shows a partial DFG from the rijndael benchmark. Here all nodes are “addition” instructions hence the labels are omitted. The algorithm fails to generate all possible DC_Sets in acceptable time if no special care is taken, since there are more than 30 fan-outs at node 370, 372, and 374. However, if we take a close look at the DAG structure, we notice node 380, 388, 400…1136 are equivalent, similarly the sub-graphs rooted at node 372 and 374 are equivalent. In other words, this DAG is highly symmetric and most of its sub-graphs are identical under isomorphism. Since our task is to generate all possible patterns for custom instructions, isomorphic patterns need only be generated once. Using this strategy, the number of patterns to be checked can be greatly reduced. However, in order to identify the nodes that are images of each other under isomorphism, efficient algorithms are required. As this topic is not addressed in this work, we just bring up this point and briefly discuss its usefulness in generating patterns for difficult DAGs. Interested reader may refer to [20] for a comprehensive discussion on graph isomorphism.

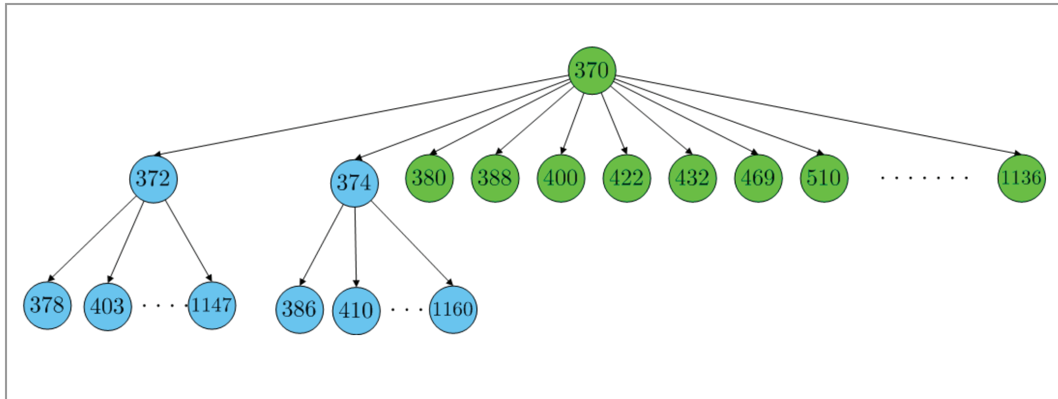


Figure 9: Part of a DFG from rijndael benchmark. All nodes are “+” instructions.

Chapter 4: Pattern Selection

4.1 Introduction

After pattern candidates from each basic block are generated, we need a proper representation so that equivalent patterns can be recognized. The nauty package [31] on graph isomorphism is employed to compute the canonical label of each pattern graph. We combine the canonical label, the operation types and the output ports together to uniquely represent each pattern. A hash function is applied to this pattern representation and a 32-bit hash code is generated. The hash code is indexed into a hash table which keeps a count and a list of its instances in the basic blocks for individual patterns. The hash table is dumped for pattern selection after all basic blocks are processed. We apply a greedy algorithm to select the optimal set of custom instructions, subjected to resource constrains.

4.2 Adjacency matrix representation of graphs

A graph $G(V,E)$ can be represented by adjacency lists or adjacency matrix. Although the adjacency lists representation is more economic in terms of memory usage, adjacency matrix is often preferred as edges between any two nodes can be checked in $O(1)$ time. In this work, the adjacency matrix representation is used. The adjacency matrix **M** for a graph with n nodes is a nxn binary matrix. $M(i,j)=1$ if there is an edge from node i to node j, otherwise $M(i,j)=0$.

However, structurally equivalent graphs may not have the same adjacency matrices. This is illustrated in Figure 10. Figure 10(a) and Figure 10(b) show two graphs that are equivalent, but their adjacency matrices are different. The difference comes from the non-uniqueness of topological ordering: both orderings in Figure 10(a) and Figure 10(b) satisfy topologic conditions. In fact our ordering is directly obtained from instruction sequence, and instruction 1 may appear before instruction 2 or after instruction 2, thus this ambiguity cannot be resolved easily.

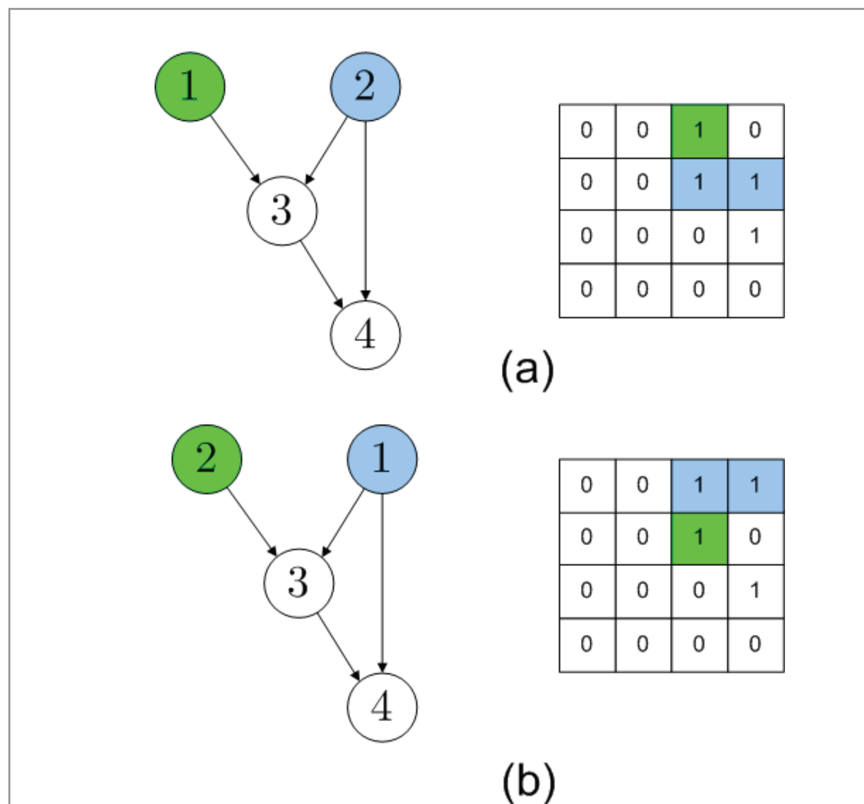


Figure 10: Equivalent graphs have different adjacency matrix representations

The differences in adjacency matrices, despite the fact that the graphs are equivalent, would generate different hash code and recognized as different

patterns if not handled. Those patterns are isomorphic with each other and an algorithm that re-labels isomorphic graphs to obtain a common adjacency-matrix representation is needed.

4.3 Canonical Label and the nauty package

Let $G(V, E)$ be a graph, γ be a permutation of V, $v \in V$ .Then v^{γ} is the image of v under γ, G^{γ} is the graph in which vertices x^{γ} and y^{γ} are adjacent if and only if *x* and *y* are adjacent in *G*.

Definition 7: *Automorphism Group*

The *automorphism group* of a graph G is the set of all permutations γ such that $G^{\gamma} = G$.

Definition 8: C*anonical Labelling*

A *canonical labelling map* is a function C such that, for any graph *G*, and permutation γ of V, we have:

(a) $C(G) = G^{\delta}$ for some permutation δ

(b) $C(G^{\gamma}) = C(G)$

Informally, graphs generated by permutations from the same automorphism group are structurally identical and their canonical labels are identical. By computing the canonical labels of all the generated pattern graphs, we are able to group

structurally identical patterns together.

The nauty package [31] developed by professor B. D. McKay is one of the fastest algorithm that perform graph isomorphism detection and canonical label generation. We applied this package to our system.

4.4 Complete pattern representation

The adjacency matrix only encodes the pattern graph's structure, which is not sufficient to uniquely represent a pattern graph. For instance, two pattern graphs may have the same structure but different instruction type at each node. Even if both structures and instruction types are the same, we need to check the output nodes before we conclude those two patterns are equivalent.

The complete pattern representation thus contains three parts: the adjacency matrix, the operation type array and the output port array. To reduce storage and hash code computation, instead of using integer arrays, we pack the adjacency matrix into a much more compact form called setword.

A setword essentially is a 16-bit short integer. A set with size n can be represented by $m=n/16+1$ setwords. Each bit in the m setwords corresponds to one element of the set and it can be set to 0 or 1 to indicate the absence/presence of the element. The adjacency matrix of a graph with n nodes can be represented by nxm setwords.

The i-th set gives the adjacencies from node v_i to all other nodes, for $1 \le i \le n$. When graph size is not multiple of 16, there would be unused bits in each set, they are set to 0s.

Figure 11 shows an example of adjacency matrix represented using setwords. The graph has 18 nodes, hence each node needs 18/16+1=2 setwords. The total memory storage used is 18x2=36 short integers. On the other hand, if short integer array is used, the storage required is 18x18=324 short integers. This shows a great storage saving can be achieved by using setwords. In Figure 11, the shaded bits represent the adjacency matrix and bits that are not shaded are set to 0s.

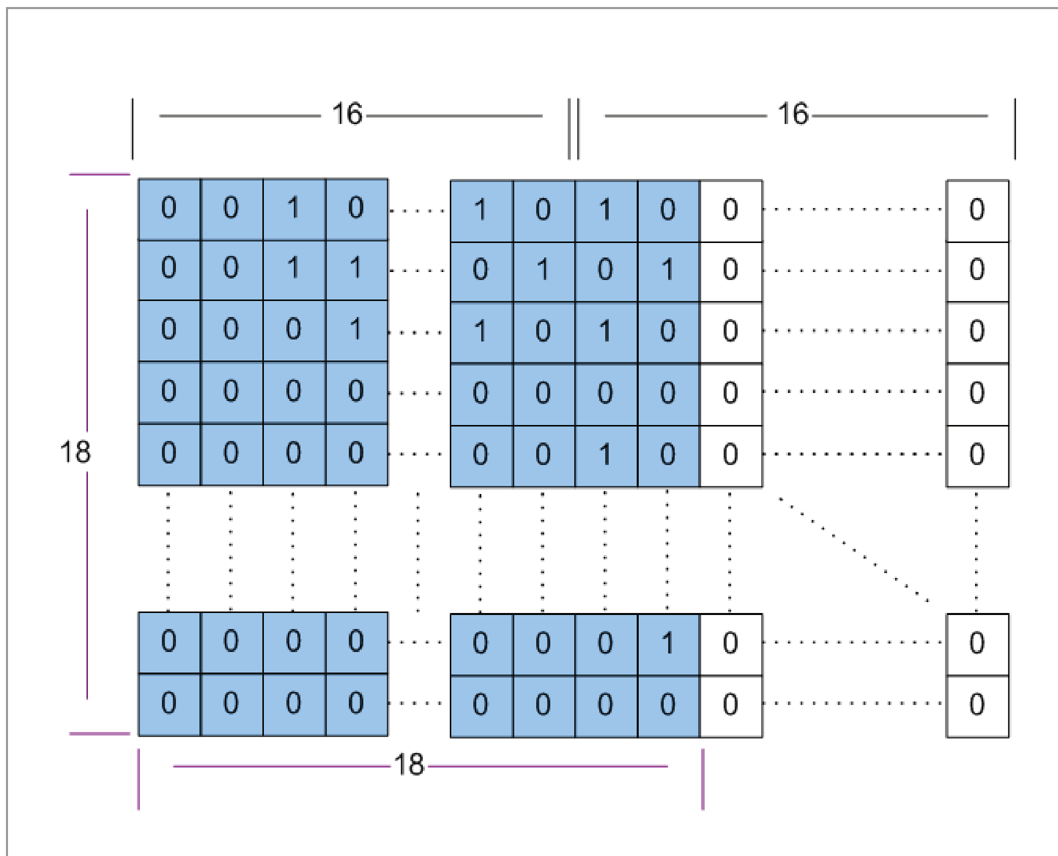


Figure 11: The setword representation of adjacency matrix

We use short integers {1, 2, 3, …, 8, 9} to represent the instruction type {add, sub, mul, div, shift, logic, lui, slt, invalid}. For a graph of size n, n short integers are required to encode the instruction types.

Finally, we use an array of size MAX_OUT to store the output nodes. If a pattern has less than MAX_OUT output nodes, the unfilled slots in the array are set to -1.

The above three arrays are stacked together to form a larger short-integer array. This is illustrated in Figure 12 and the labels below the bar diagram indicates the size of each part in terms of short integers.

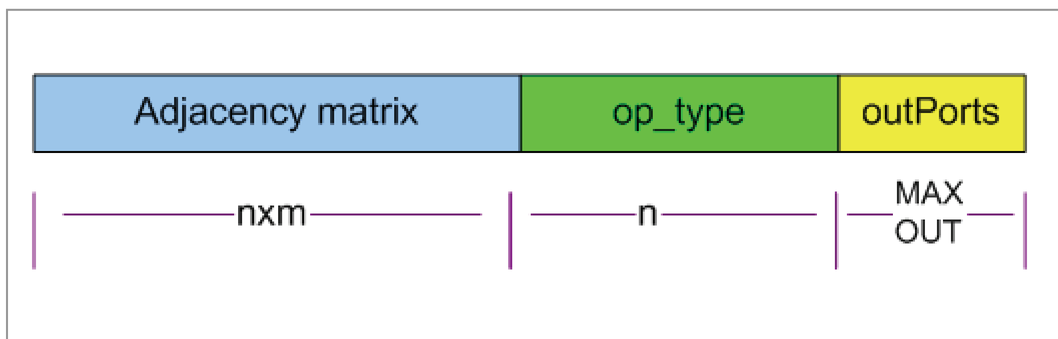


Figure 12: The complete representation of a pattern graph

4.5 Hash key generation

The complete representation discussed in the previous section is generated for each pattern instances C_i in each basic block. A simple hashing function is defined to take the complete pattern representation as input and generates a 32-bit

hash key. Ideally identical patterns generate the same hash key and different patterns generate different hash key. However, there are chances that different patterns generate the same hash key and this problem is resolved by chaining mechanism.

For each pattern instance, after the hash key is generated, the content of the hash table indexed by that key is updated. In this work, we defined a C++ class called "Candidate" and the hash table is an array of the "Candidate" class. The "Candidate" class keeps a complete list of pattern instances that are hashed into the current location. In addition, it records the total frequency of the pattern.

4.6 Instance list

It is important to note that simply record the total frequency of each pattern is not sufficient for pattern selection. The reason is instances of different patterns may overlap and including one pattern into custom instruction set would change the frequency of other patterns whose instances are overlapping with the selected one. If we simply record the total frequency and use it as the selection metric, the generated custom instruction set would be biased as this policy favors overlapped patterns from high frequency basic blocks.

To solve this problem, an instance list or instance table is defined in the "Candidate" class. Each element in the list contains three fields: the original basic

block id "BB_ID", the array of node numbers in the original basic block "images[]", and the execution count of the basic block "frequency". Figure 13 shows an example where two patterns P1 and P2 (other patterns are ignored) are generated from two basic blocks BB1 and BB2. There are two instances for P1, denoted as P1:C1, P1:C2 and three instances for P2, denoted as P2:C1, P2:C2, P2:C3. The instance lists in Candidate (P1) and Candidate (P2) are shown in Table 3.

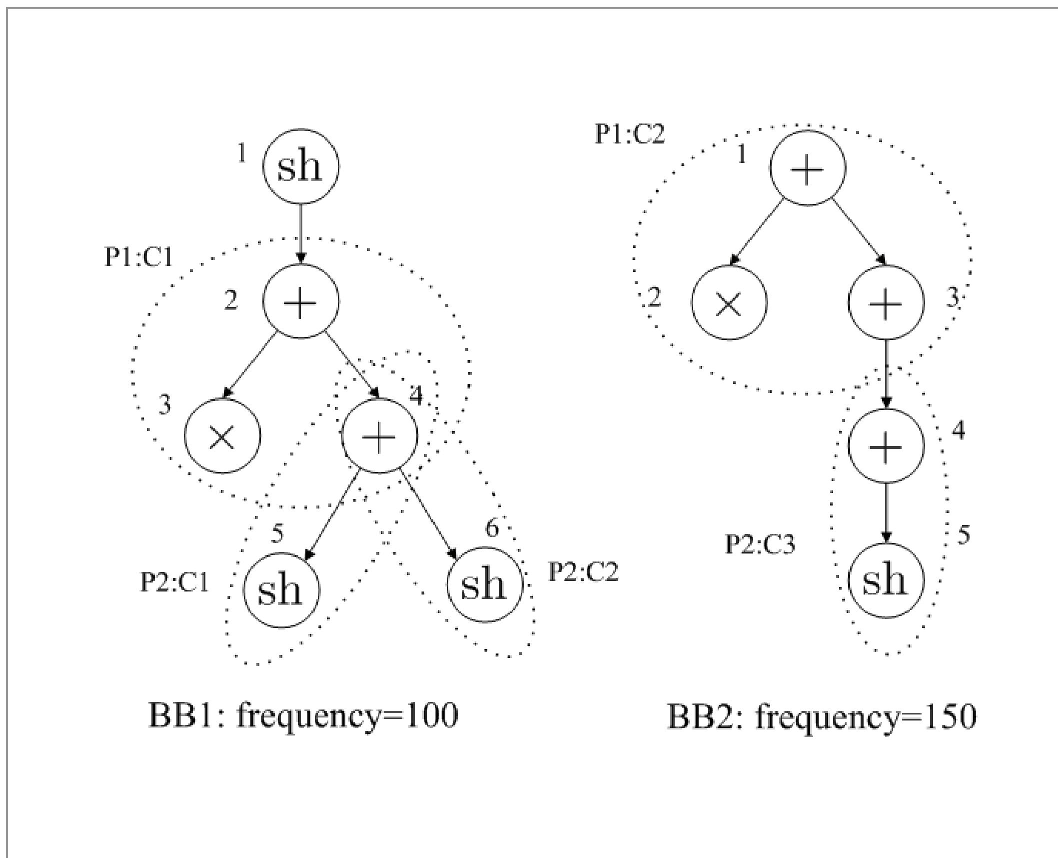


Figure 13: Pattern instances that are overlapping.

Table 3: Instance lists examples

<table>
<tr><th></th><th>BB ID</th><th>Images[]</th><th>Frequency</th><th>Total Frequency</th></tr>
<tr><td rowspan="2">Candidate P1</td><td>1</td><td>2, 3, 4</td><td>100</td><td rowspan="2">250</td></tr>
<tr><td>2</td><td>1, 2, 3</td><td>150</td></tr>
<tr><td rowspan="3">Candidate P2</td><td>1</td><td>4, 5</td><td>100</td><td rowspan="3">350</td></tr>
<tr><td>1</td><td>4, 6</td><td>100</td></tr>
<tr><td>2</td><td>4, 5</td><td>150</td></tr>
</table>

4.7 Software latency, hardware latency and speedup

The *software latency* of a custom instruction is the overall execution time of its primitive instructions, assuming single-issue pipelined microprocessor architecture. The execution time of trivial patterns is given in Table 4. We assume all primitive instructions that can be included into custom instructions, except division, require 1 machine cycle to execute. Division requires 10 machine cycles to finish. The software latency of non-trivial patterns is the summation of individual instructions, as we assume all the instructions in a pattern need to be executed sequentially in a single-issue pipelined processor. Thus, for a pattern P, we have:

$$T_{sw}(P) = \sum_{v \in P} T_{sw}(v)$$

The *hardware latency* of a custom instruction is the required cycle number of execution on customized hardware logic. Accurate estimation of hardware latency of each pattern requires logic synthesis and post-synthesis technology mapping. In our system, since all candidate patterns that satisfy constrains are enumerated, it would be inefficient to perform cycle-accurate logic synthesis for each individual

patterns. Instead, we estimate the hardware latency from the pattern's critical path and hardware latencies of individual operations. The hardware latency model is synthesized using standard cells from a popular library and is mapped to 0.18 μm CMOS technology [Ataas]. This is also shown in Table 4.

In some studies, the hardware latency is calculated in an additive manner: the summation of hardware latencies of individual nodes along the pattern graph's critical path and then rounded up to the nearest integer. We believe the more precise definition should be the maximal latency along all possible critical paths.

$$T_{hw}(P) = \left\lceil \max_{\forall cp(P)} \left\{ \sum_{v \in cp(P)} T_{hw}(v) \right\} \right\rceil$$

The reason is that a given pattern graph may contain more than one critical paths. A simple example is the pattern P1 in figure 14. Both $+ \rightarrow \times$ and $+ \rightarrow +$ are critical paths of length 2, the latency of the entire pattern graph should be calculated as $\lceil \max(0.25+1, 0.25+0.25) \rceil = 2$.

Table 4: Software and hardware latency models of common operations

	ADD	MUL	DIV	SHIFT	LOGIC
Software Latency	1	1	10	1	1
Hardware Latency	0.25	1	9.61	0.16	0.02

The potential speedup of a custom instruction is the difference between its software latency and hardware latency, i.e.

$$speedup(P) = T_{sw}(P) - T_{hw}(P)$$

Using this formula, the speedups for P1 and P2 in Figure 14 are both 1.

A side note is that shift and logic operations can be easily optimized in current FPGA/ASIC technologies and can be executed in almost zero time. Thus it would be advantageous to implement custom instructions in hardware if the application contains a large percentage of shift/logic operations (As can be seen later, applications in security domain are able to achieve high degree of speed up).

4.8 Optimal custom instruction selection: ILP formulation

Suppose there are N unique patterns over all the basic blocks and they are denoted as $P_1, P_2, ..., P_N$. For each pattern, there are n_i instances $C_1^i, C_2^i, ..., C_{n_i}^i$ and each instance has an associated execution frequency $f_1^i, f_2^i, ..., f_{n_i}^i$. For each pattern P_i, we use R_i to denote the resource requirement and S_i to denote the speedup. The resource requirement of a pattern can be calculated in an additive manner: the summation of all its instructions' resource requirements. For some extensible processor, the number of custom instructions is the only restriction and in that case $R_i = 1$. We further define two set of binary variables $B_1, B_2, ..., B_n$ and $b_1^i, b_2^i, ..., b_{n_i}^i$. B_i is associated with pattern P_i: if $B_i = 1$, P_i is included in the final selection otherwise excluded. Similarly, b_j^i is associated with instance C_j^i: if $b_j^i = 1$, C_j^i is selected to cover the instructions. It is important to note that if $C_j^i = 1$ $(1 \le j \le n_i)$, then the pattern P_i is automatically included into the final

selection, i.e. $if \quad \exists j, s.t. \quad b_j^i = 1, then \quad B_i = 1$.

The objective function to be maximized is the overall speedup, i.e.

$$F = \sum_{i=1}^{N} \sum_{j=1}^{n_i} (b_i^j \times S_i \times f_i^j)$$

However, the optimization must be done under the constrain each instruction is covered by at most one instance (it may not be covered by pattern instances at all, i.e. it is covered by trivial patterns instead). This constrain is expressed as follows: if an instruction can be covered by pattern instances $C_{i1}^{j1}, C_{i2}^{j2}, ..., C_{ik}^{jk}$, then

$$b_{i1}^{j1} + b_{i1}^{j1} + ... + b_{i1}^{j1} \leq 1 \quad (1)$$

Note for all instructions that can be covered by pattern instances, there is an equation in the form of (1) associated with it. Thus the number of constrain equations is huge.

Besides the (1) constrain equation, there is another constrains equation on the hardware resource. The following equation simply ensures the resources used for custom instructions are with the limit:

$$\sum_{i=1}^{N} B_i \times R_i \leq R_{total}$$

Although the optimal custom instruction set selection problem can be formulated as ILP problem perfectly, it is often of little interests. The reason is even for applications with small basic blocks, the number of pattern instances can easily be very large. Moreover, the number of constrain equations is almost proportional to the number of valid nodes in the application. In practice, pattern selection is done

using heuristic algorithms that try to achieve sub-optimal solutions. The remaining sections of this chapter are devoted to our heuristic algorithm on pattern selection.

4.9 Custom instruction selection: greedy algorithm

The objective of pattern selection is to choose an optimal set of patterns $T = \{T_1, T_2, ... T_M\}$ out of a huge number of valid pattern candidates $P = \{P_1, P_2, ..., P_N\}$, subjected to area or quantity constrains.

The core of the greedy algorithm is to design a priority function that estimates the maximally achievable speedup for each pattern. The greedy algorithm then sort the patterns according to their priorities and select P^*, the one with highest priority. The selection of P^* in general necessarily affects the achievable speedup of the remaining pattern candidates if they are overlapping with P^*. As a consequence, the priority of those remaining patterns must be recalculated. The greedy algorithm continues the above procedure until resource constrains are reached or no more candidates is available.

In this section, we discuss the overall structure of the greedy algorithm, as shown in Figure 14. The priority calculation, which is non-trivial, will be discussed in the following sections. In Figure 14, Line 3-8 calculates the priority of each pattern and adds the one with highest priority to the finalist. Line 9-14 checks all the remaining pattern's instances, any instance that is overlapping with the selected

one will be removed from the instance list.

$input: set \ \ of \ \ unique \ \ patterns \ \ P$
$output: set \ \ of \ \ patterns \ \ T \ \ to \ \ be \ \ implemented \ \ in \ \ hardware$

01. $T \leftarrow \varnothing;$
02. $while \ \ P \neq \varnothing \ \ do$
03. $\quad for \ \ each \ \ P_i \in P, \ \ calculate \ \ priority(P_i);$
04. $\quad select \ \ P^* \ \ s.t. \ \ prio(P^*) = \max_{\forall P_i \in P}\{prio(P_i)\};$
05. $\quad if \ \ R + R(P^*) < R_{total} \ \ then$
06. $\quad\quad P \leftarrow P - P^*;$
07. $\quad\quad T \leftarrow T \cup \{P^*\};$
08. $\quad end$
09. $\quad for \ \ all \ \ P_i \in P$
10. $\quad\quad for \ \ all \ \ instances \ \ C_i^j \ \ of \ \ P_i$
11. $\quad\quad\quad if \ \ P^* \cap C_i^j \neq \varnothing \ \ then \ \ C_i \leftarrow C_i - C_i^j$
12. $\quad\quad\quad end$
13. $\quad\quad end$
14. $\quad end$
15. end
16. $return \ \ T;$

Figure 14: The greedy algorithm on pattern selection

4.10 Maximally achievable speedup as the priority function

In the pattern selection phase, we need a priority function for each pattern so that the one with the highest priority is selected first. Naturally, the maximally achievable speedup is used as the priority function. However, given the pattern speedup and total frequency, the maximally achievable speedup is not simply the product of those two. The reason is not all instances of the pattern can be mapped to the custom instruction simultaneously.

Let's again look at the example in Figure 13. For P1, its instances C1 and C2 are from different basic blocks and thus they are independent. If we implement P1 in custom hardware, both C1 and C2 can be mapped to the hardware logic and the maximally speed up for P1 is indeed $1 \times (100 + 150) = 250$. On the other hand, although P2 has a total frequency of 350, not all instances are realizable concurrently. For example, P2:C1 and P2:C2 have node 4 as the overlapped node. Thus if P2:C1 is mapped to the custom hardware, P2:C2 can no longer be mapped. P2:C3 on the other hand, is not affected since it is from another basic block. The maximally achievable speedup for P2 is achieved by either mapping P2:C1 and P2:C3 or P2:C2 and P2:C3 to the custom hardware, which is $1 \times (100 + 150) = 250$. Note however, in some systems, such as [Cong], overlapped instances are allowed.

The instance list in each pattern is used to calculate the maximally achievable speedup. It is quite frequent that an instance list is of quite large size, say, containing more than 50 elements. It would be very inefficient if exhaustively enumeration is used. The entire instance list is first partitioned into disjoint groups so that instances from different groups never overlap. A simple Branch-and-Bound algorithm is then applied to each group to obtain the maximally achievable frequency of that group. The overall maximum frequency is calculated by summing up the frequencies of each group and the priority is simply the maximum frequency times the single pattern speedup.

4.11 Branch-and-Bound algorithm

We briefly give a description on *branch-and-bound* algorithms because it will be applied to not only this section, but also optimal code generation. *Branch-and-bound* is an approach developed for solving discrete and combinatorial optimization problems. The discrete optimization problems are problems in which the decision variables are discrete values; when this set is set of integers, we have an integer programming problem. The combinatorial optimization problems, on the other hand, are problems of choosing the best combination out of all possible combinations. Most combinatorial problems can be formulated as integer programs. Our problem of selecting a subset from the pattern instances to achieve a maximum total frequency is exactly an example of both discrete optimization problems and combinatorial optimization problems.

As stated by Murty [24], the major difficult of solving these problems is we don't have any optimality conditions to directly check whether a given solution is optimal or not. In other words, unlike other linear or non-linear optimality problems where the target to be optimized can be expressed as a function of the decision variables, there is no way of applying traditional analytic methods to discrete and combinatorial optimization problems. In general, optimality for such problems can be assured only if all feasible solutions are enumerated and compared against each other. However, in practice, it is often possible to avoid

enumerating some feasible solutions if there is sufficient reason to believe it is safe to do so. The brand-and-bound algorithm is such an approach that generates partial enumeration of all possible alternatives without losing optimality.

The essence of the branch-and-bound approach is the following observation: in the total enumeration tree, at any node, if it can be proved that optimal solution cannot occur in any of its descendents, then there is no need to consider any of those descendent nodes. This is known as search tree pruning. It is important to note the optimality is never compromised as those solutions in the leaves of the pruned branches can not be the optimal solution, according to the definition of pruning. Thus, the *branch-and-bound* approach is not a heuristic procedure, but an exact optimizing procedure.

Let's assume a feasible solution has been found, either by heuristic methods or by a depth-first search to reach the first leaf in the search space. Since this solution is so far the best solution available, we assign it to the global threshold. Then at any node of the search tree, if we can compute a bound on the best possible solution that can expected from the descendents of that node, we can compare the bound with the global threshold. If what we have on hand, the global threshold, is better than what we can expect from any of the descendents, then it is safe to stop branching from that node. In other words, all the descendents can be pruned.

It is trivial that the performance of the branch-and-bound algorithm or the actual runtime complexity depends on the prune techniques. In general, the quality of the prune techniques often boils down to how well the lower/upper bound (depends on whether minimum cost or maximum value is to be found) of the decedent trees is estimated, given the current position in the search tree. The tighter the lower/upper bound, the more the search tree can be pruned. As will be mentioned soon, we calculate the upper bound by summing up the frequencies of un-chosen pattern instances. The method is attractive for its simplicity and experiments show it works fine for all benchmarks and applications tested. It is worth noting the method proposed in [8] cannot be applied to this problem as it is applicable to lower bounds only. Moreover, the method in [8] is trying to heuristically find an independent subset of un-chosen candidates, whereas in our problem, all patterns in each disjoint group are from a dependent set.

The Branch-and-Bound algorithm recursively splits the original problem into two sub-problems and finds the maximally achievable frequency over a group of instances. The algorithm is as follows:

(Denote the list of instances as L, the chosen list as L_chosen, the current global maximum frequency as GMF, and the sum of frequencies along the binary search three to the current node as CPF. Denote the branch-and-bound procedure as $< L, L_chosen, CPF, GMF >$)

Step 1: Initialize variables to

$L_chosen := empty;$
$GMF = 0$
$CPF = 0;$
$call < L, L_chosen, CPF, GMF >$

Step 2: In $< L, L_chosen, CPF, GMF >$ procedure:

Case 1: L is empty. This indicates we are at the leaf node of the search tree.

If CPF>GMF, update GMF=CPF.

If CPF<GMF, the current search path is worse the GMF obtained earlier, take no action.

Case 2: L is not empty. We check the up-bound of the additional achievable frequency (UAF) from the remaining instances in L: $UAF = \sum_{i \in L} frequency(i)$.

Note the calculation of UAF ensures that the remaining achievable frequency is no more than that. Hence if CPF plus UAF is smaller than GMF, there is no reason to continue search along the current direction. Thus we bound.

If CPF+UAF > GMF, there is potential to obtain better total frequency by continuing the branching. We pick one pattern instance C_i from L and form two sub-problems that recursively called:

Sub-Problem 1: considering including C_i into the chosen list, update CPF by adding the frequency of C_i to it. Next we scan through all the remaining instances in L: if they are overlapping with C_i, remove them from the L.

$$L = L - \{C_i\} - overlap_group(C_i);$$
$$L_chosen = L_chosen + \{C_i\};$$
$$CPF = CPF + frequency(C_i);$$
$$call < L, L_chosen, CPF, GMF >$$

Sub-Problem 2: considering not including C_i into the chosen list, thus we simply remove C_i from L, and call the BnB algorithm with the same CPF and chosen list, i.e.

$$L = L - \{C_i\};$$
$$call < L, L_chosen, CPF, GMF >$$

After the BnB algorithm terminates, the maximally achievable frequency of that group is returned in GMF and the corresponding instances are given in L_chosen.

To illustrate this algorithm, an example is shown in Figure 15. On the right of Figure 15 is the pattern structure and we assume all its instances are from one single basic block, whose DFG is shown on the left. There are seven instances labeled as *C1-C7*. In this example, all the patterns are from the same disjoint group. Since only one basic block presents, we can simply look at the size of each feasible solution and ignore the basic block frequency.

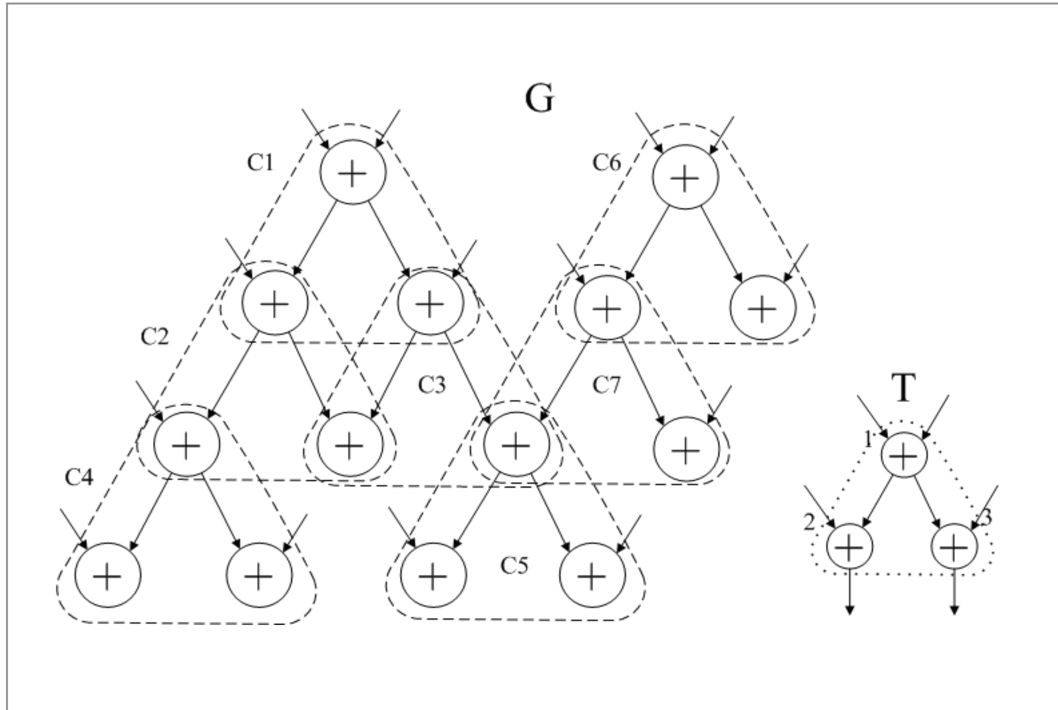


Figure 15: Maximum achievable frequency: the pattern T and instances C1-C7.

The corresponding binary search tree is shown in Figure 16. It is clear that this tree is not a full binary search tree since when we formulate sub-problem 1, all instances that are overlapping with current selected instance are removed. At each node of the search tree, the left branch corresponds to sub-problem 1, i.e. including this instance into finalist; whereas the right branch corresponds to sub-problem 2. In Figure 16, instance C1 is considered first. If C1 is selected, C2 and C3 will be removed and the next pattern to be considered is one of C4, C5, C6, and C7. At level 2, assume C4 is considered. The left branch corresponds to selecting C4 and no instances are removed. Now we are at level 3 and have C5, C6 and C7 left. Assume C5 is considered at level 3. The left branch corresponds to selecting C5, and C7 will be removed. Finally the algorithm reaches level 4 where the only one to consider is C6. The left branch corresponds to selecting C6 and we

reach the first leaf node, hence a feasible solution {C1, C4, C5, C6} is obtained. The size of this solution is 4 and the global maximum frequency GMF is updated. The right branch at level 4 corresponds to not selecting C6 and a feasible solution {C1, C4, C5} is obtained as well. However, this solution is discarded as the size is only 3.

Now consider the right branch (not selecting C5) at level 3, due to the natural of recursive call (depth-first like), when it is processed, the solution {C1, C4, C5, C6} is already obtained. The path frequency associated with this node is 2 ({C1, C4}), and the remaining instances to choose are {C6, C7}, thus the additional achievable frequency is 2. By now it is safe to say the best solution can be obtained by exploring the descendants is 2+2=4. Since the current solution in hand already has a frequency of 4, it is not necessary to continue traversing the descendants.

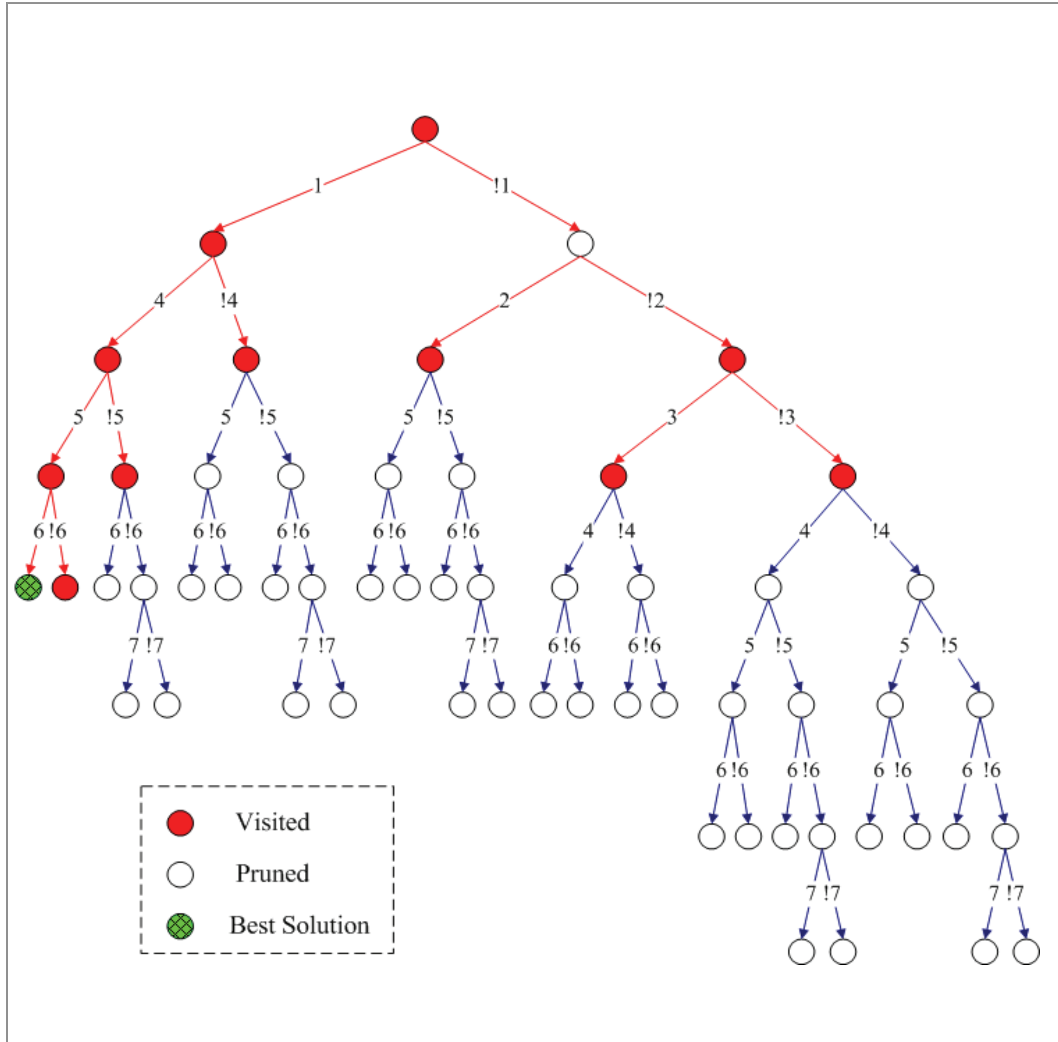


Figure 16: The binary search tree associated with the example in figure 15.

The shaded nodes in Figure 16 correspond to visited nodes whereas the blank nodes are pruned. It is worth noting only 2 out of 29 leaves are visited, indicating the efficiency of this simple pruning technique. It is interesting to note in this example, the first feasible solution is the final optimal solution as well. Although this is a coincident, the order of sub-problem 1 and sub-problem 2 does affect how fast the best solution can be obtained. For instance, consider at each level we branch to the right sub-tree first, the first feasible solution is simply {C7}. Although not always true, for this problem, it is almost always better to branch to

sub-problem 1 first.

The overall algorithm that finds the maximally achievable frequency for each pattern is summarized in Figure 17. (L denotes the entire instance list, L_group denotes each disjoint group, total_freq denotes the total achievable frequency of the pattern). Line 3-13 identifies one disjoint group and line 14-15 calls the BnB algorithm to find the maximum achievable frequency over this group.

```
< Priority(L) >
01.   total_freq = 0;
02.   while  L ≠ ∅  do{
03.       L_group ← ∅;
04.       C ← L.pop();
05.       L_group ← L_group ∪ {C};
06.       while new instances are added to (L_group) && L ≠ ∅
07.           for all C' ∈ L
08.               if isOverlap(C', L_group) then
09.                   L ← L − {C'};
10.                   L_group ← L_group ∪ {C'};
11.               end
12.           end
13.       end
14.       L_chosen ← ∅;  GMF = 0;
15.       call  < L_group, L_chosen, &GMF >;
16.       total_freq ← total_freq + GMF;
17.   end
18.   return  priority = total_freq × speedup;
```

Figure 17: Algorithm that calculates the priority of each pattern.

4.12 Conclusion

In this section, we first introduced the adjacency matrix representation of graph

structure followed by the generation of canonical labels of isomorphic graphs. A unique representation combing graph adjacency matrix, instruction type and output ports is designed. We define a hash function that converts the pattern representation into a 32-bit hash key and instances of the same pattern produce identical hash keys. The instance list of each pattern is updated as instances are indexed into the hash table. When all the basic blocks are processed, the software latency, hardware latency and hence speed up of each pattern are calculated. A priority function that estimates the maximally achievable speedup is defined and a branch-and-bound algorithm is designed to calculate the priority. Finally, we use a greedy algorithm to iteratively select a custom instruction set under the resource constrains.

Chapter 5: Application Mapping

5.1 Introduction

Many previous works stop right after pattern selection. We believe application mapping is an essential part of a practical extensible ISA system. Given a set of custom instructions, or the library instructions, we detect all possible matches from the application's DFGs to the library instructions. The algorithm we use is a modified version of Ullman's subgraph isomorphism algorithm. Properties of digraphs are incorporated into the refinement procedures to prune the search space. After all matches are generated, we cast the optimal mapping problem into a special version of set covering problem and developed a branch-and-bound algorithm to find the best solution.

5.2 Sub-graph isomorphism

For each basic block's data flow graph G, we want to match it against the custom instructions {T1, T2 …TM}. This is decomposed into finding the matches from G to each Ti. Let T being the DAG representation of any custom instruction Ti, the matching problem is the same as detecting subgraph isomorphism from T to the subject graph G.

It is well known subgraph isomorphism is NP-Complete [10]. Till today, it still remains an open question whether polynomial time algorithms can be found for

graph and sub-graph isomorphism. As for digraphs, the problem is as difficult as regular graphs. Although the problem can be solved using exhaustive enumeration, the complexity is exponential: $|G|^{|T|}$. We summarize the efforts on DAG isomorphism and subgraph isomorphism as follows:

Many works [9][12][16][18][19][28] focused on special type DAGs. For some restricted DAG types, polynomial or linear time algorithms exist. Rooted DAG is discussed in [16] and the time complexity is further reduced to $O(|E(P)| \times |V(T)| + |E(T)|)$ in [9]. Series Parallel (SP) digraphs are discussed in [19 28] and biconnected outerplanar graphs are discussed in [18]. However, as DFGs in general are not special digraphs, the above approaches are not applicable.

For normal graphs, most algorithms developed are based on state-space search and backtracking. The earliest work is dated to Corneil and Gotlieb's algorithm [7]. The major improvement was introduced by Ullmann[27]. In Ullmann's work, a backtracking algorithm with a refinement procedure was proposed. The refinement procedure effectively reduces the search space need explored. The above algorithms are developed for one-to-one subgraph isomorphism detection. Recently, Messmer and Bunke [22] proposed an algorithm for library based matching. Their method computes all possible isomorphic graphs of model graphs in the preprocessing step and representing the computed results in a decision tree. The decision tree is then directly used to detect graph or subgraph isomorphisms

from the input graph to the model graphs in time that is only quadratic in the size of the input graph. However, the attractive run-time complexity comes from exponential time complexity during preprocessing stage and the exponential size of the decision tree. Unfortunately this algorithm is only applicable for subgraph isomorphism from subject graph to library graphs whereas in our system, the reverse has to be done.

As mentioned, most works on instruction set extensible processors did not address application to custom instruction set mapping. To our knowledge, the only work that applied such mapping is done by Clark et. al [4]. Other works directly used the pattern instances information from pattern generation stage to perform optimal instruction covering. In practical systems, after the custom instruction set is fixed, it is likely new applications are required to port to the new ISA. In that situation the possible mapping from the new application to the ISA is completely unavailable and algorithms for optimal code generation can not be performed. To concluded, the matching procedure is an essential part for a complete system. In Clark's work, the vflib [6] graph matching library (also exponential in worst case) is directly applied, thus the matching procedure was not discussed in details.

5.2.1 Ullmann's graph isomorphism algorithm

The core of our approach to tackle custom instruction matching is similar to Ullmann's algorithm as it is fast yet easy to implement. However the refinement

procedure in our work is improved as it utilizes the properties associated with DFG matching to efficiently prune the search space. We will introduce Ullmann's algorithm and followed by our refinement procedure in the following.

Let's denote the library graph as $T < V_T, E_T >$, where V_T, E_T denote the node set, edge set respectively. Let M_T denotes the $m \times m$ adjacency matrix of T, where m is the number of nodes. Let L_T denotes the node array that stores the instruction types. We further define two node arrays $inDeg_T, outDeg_T$. As their names indicate, $inDeg_T[v]$ is the number of input edges of node v and $outDeg_T[v]$ is the number of output edges of node v.

Similarly suppose the subject graph $G < V, E >$ is of size $n, (n \geq m)$, we use matrix M and node array $L, inDeg, outDeg$ to denote its adjacency matrix, instruction types, input degree and output degree.

We define a permutation matrix Φ to be a $m \times n$ binary matrix whose elements are either 0 or 1. In addition, each row of Φ contains exactly one 1 and no column contains more than one 1, i.e.

$$\begin{aligned} &\forall i, (1 \leq i \leq m), \sum_{j=1}^{n} \Phi(i,j) = 1 \\ &\forall j, (1 \leq j \leq n), \sum_{i=1}^{m} \Phi(i,j) \leq 1 \end{aligned} \quad (2)$$

Actually, the permutation matrix specifies a node-to-node mapping from T to G: if $\Phi(i,j) = 1$, node i in T is mapped to node j in G. In a valid sub-graph

isomorphism, each node in T is mapped to exactly one node in G and since $n \geq m$, there might be nodes in G that have no image nodes in T, these are actually formulated as constraint (2) of permutation matrix. A valid sub-graph isomorphism from the library graph to the subject graph can be specified by a permutation matrix Φ that satisfies

$$M_T = \Phi M \Phi' \quad (3)$$

Thus, the problem of finding all isomorphic sub-graphs in G that are isomorphic with T is equivalent to finding the set of permutation matrices $\{\Phi_1, \Phi_2, ..., \Phi_k\}$ for which (3) is true.

Step 1: Construct the initial matrix Φ_0, which encodes all possible node-to-node mappings from T to G. A possible node-to-node mapping must satisfy three conditions:

(1) The instruction types must be the same.

(2) The input degree of the node in the library graph must be smaller than or equal to that of the node in the subject graph. Here the input degree means the number of predecessor nodes, not the number of input edges. For example, in Figure 18, the input degree of node 1 is zero, although it has two input edges.

(3) If a node is not an output node in the library graph, it can only be mapped to a node in the subject graph with the same output degree. If a node is an output node in the library graph, it can be mapped to a node in the subject

graph whose output degree is equal to or greater than it.

To illustrate constraint 3, an example is shown in figure 18. T1, T2 and T3 are the library graphs and G is the subject graph to be mapped. Here we manually identify three sub-graphs m1, m2 and m3 of G. Sub-graph m1 is an isolated pattern and it is covered by the original shift instruction whereas sub-graph m3 can be covered by T2. It is interesting to note m2 cannot be mapped to T1 using $\{2 \rightarrow 1, 3 \rightarrow 2\}$. The reason is node 2 in G has a fan-out towards node 4 where as for the library pattern T1, there is no fan-out at node 1 towards outside of the pattern. On the other hand, m2 can be mapped to T3 perfectly as there is a fan-out towards outside of the pattern at node 1 of T3.

If node i in T and node j in G satisfy the above three conditions, the corresponding entry in the initial permutation matrix is set to 1, i.e.

$$\Phi_0(i,j) = \begin{cases} 1 & if \quad L_T(i) = L(j) \;\; \wedge \;\; inDeg_T(i) \le inDeg(j) \;\; \wedge \\ & \{(isOutNode(i) \wedge outDeg_T(i) \le outDeg(j)) \\ & \vee \;\; (\neg isOutNode(i) \wedge outDeg_T(i) = outDeg(j)\} \\ 0 & otherwise \end{cases}$$

Note Φ_0 is not a proper permutation matrix as it in general does not satisfy constraint (2). However, it will be eventually set to valid permutation matrices, if they exist, in the following steps.

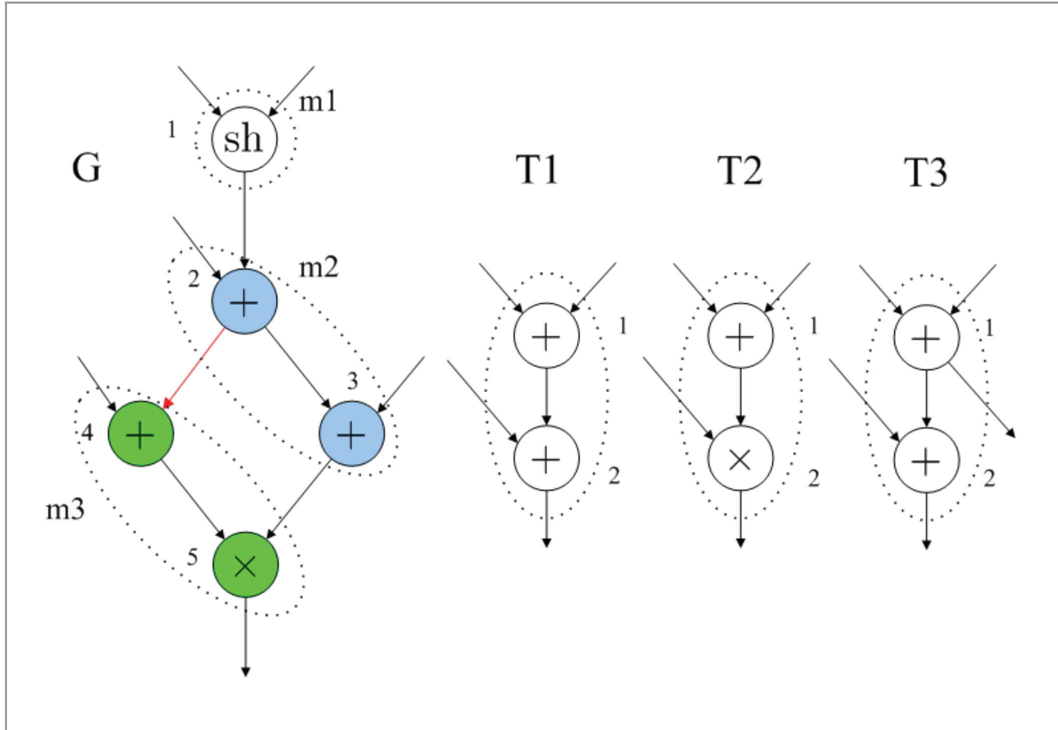


Figure 18: The output constraints that must be satisfied for custom instruction matching

Step 2: If there is at least one node in T that cannot be mapped to any nodes in G, i.e. $\exists i,(1 \le i \le m), \sum_{j=1}^{n} \Phi_0(i,j) = 0$, there is no valid sub-graph isomorphism exist. The program is terminated early. Otherwise, we systematically change all but one of the 1's in each row of Φ_0 to 0, subject to the constraint no column may contain more than one 1. After each row is changed, a refinement procedure is applied to prune the search space.

Step 3: For each resulting matrix from step 2, condition (3) is tested to examine whether it is a valid permutation matrix.

Next we discuss step 2 in detail. Without the refinement procedure, the algorithm

is a full enumeration algorithm that traverses the entire search tree in a depth-first manner, as Figure 19 shows.

We use a length-n binary vector b to record whether a column is occupied ($b[j]=0$) or not ($b[j]=1$). We use a length-m vector K to record for each row, what is the last column that has processed.

1. $d=1, \Phi=\Phi_0, K=\{0\}$;
2. $store \ \ \Phi_d=\Phi$;
3. $k=K[d]$;
   $if\ there\ \ is\ \ no\ \ \ j>k\ \ \ s.t.\ \ \Phi(d,j)=1 \wedge b[j]=1, goto\ \ \ 5.$
   $pick\ \ \ first\ \ \ k\ \ \ s.t.\ \ \Phi(d,k)=1 \wedge b[k]=1;$
   $for\ \ \ all\ \ \ j \neq k, set\ \ \ \Phi(d,j)=0$
4. $K[d]=k$;
   $if\ \ \ d<m,\ \ b[k]=0; d=d+1; goto\ \ \ 2;$
   $else\ \ if\ \ \ M_T=\Phi M \Phi', report\ \ \ \Phi\ \ \ valid;$
   $else\ \ \ \Phi=\Phi_d, goto\ \ \ 3;$
5. $if\ \ \ d=1$ terminate;
   $else\ \ \ d=d-1, k=K[d], b[k]=1, \Phi=\Phi_d, goto\ \ \ 3;$

Figure 19: Sub-graph isomorphism without pruning

5.2.2 Pruning strategies

To reduce the possible search space, we apply a few pruning strategies. Definition: the matrix-truncation operation $Si,j(M)$ on a $m \times n$ matrix M is to delete rows $i+1, i+2, ..., m$ and columns $j+1, j+2, ..., n$ and form a new $i \times j$ matrix.

1. After a new k is picked, and for all other columns $j \neq k$, $\Phi(d,j)$ is set to zero, we check whether the partial permutation matrix is valid up to depth-d using the matrix truncation operation:

$$S_{d,d}(M_T) = S_{d,d}(\Phi M \Phi') \quad (4)$$

If the above condition is not satisfied, there is no need to check rows d+1, d+2, etc. Thus we backtrack: if there is another j>k, such that $\Phi(d,j)=1$ and $b[j]=1$, we continue in the same depth. Otherwise we back to the previous depth d-1, and start from the latest column we have explored in depth d-1.

2. The adjacency constraint.

Suppose we are at any non-terminal node of the entire search tree, i.e. 1<d<m, then all rows of Φ less than d are said to be fixed by the search path and Φ is called a partial permutation matrix. For a partial permutation matrix, we have the follow equation:

$$\forall i, 1 \leq i \leq d, \sum_{j=1}^{n} \Phi(i,j) = 1 .$$

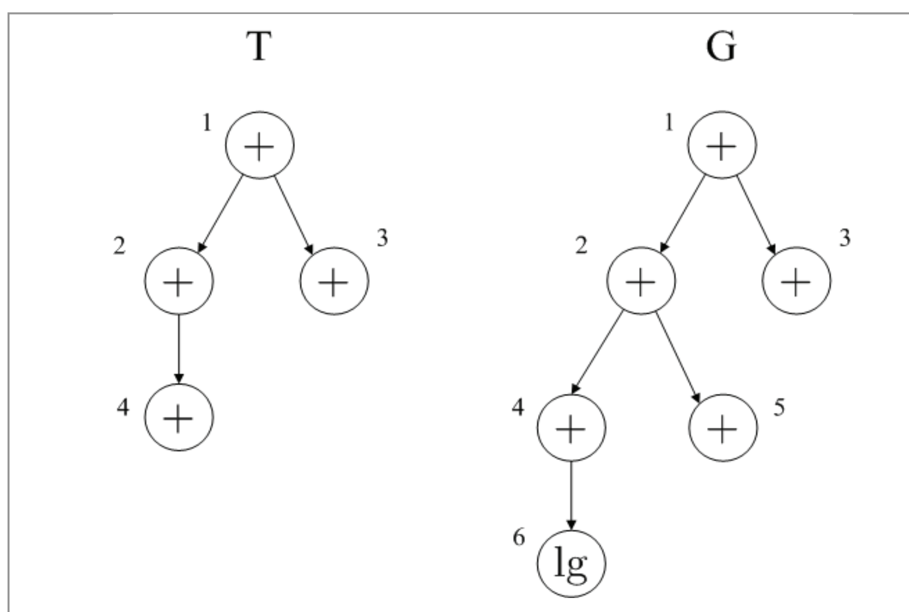
It is important to note the fixed rows 1-d can provide additional constraints to the non-fixed rows d-m. For example, suppose we have $\Phi(a,b)=1, 1 \leq a \leq d$ being fixed, which means v_a the a-th node in T and v_b the b-th node in G are matched. Let $\{v_{a1}, v_{a2}, ... v_{a\alpha}\}$ be the set of nodes that are adjacent to v_a and $\{v_{b1}, v_{b2}, ... v_{b\beta}\}$ be the set of nodes adjacent to v_b. From the definition of sub-graph isomorphism, it is clear that for each $x = 1,2,...,\alpha$, the node v_{ax} has to be mapped to a node v_{by}, where $y = 1,2,...,\beta$. Thus this refinement simply set all those entries $\Phi(ax, j)$ to 0

if node v_j is not in the adjacency set of v_b. The pseudo code is simply expressed as follows:

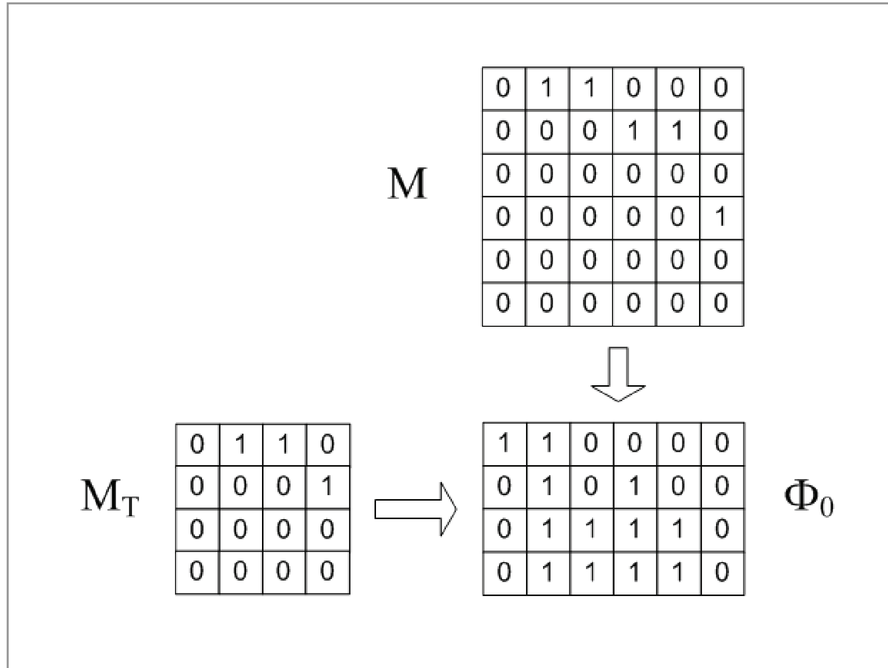
```
< refinment_procedure >
for  i = 0  to  m
    if  M_T(d,i) = 1
        for  j = 0  to  n
            if (M(i, j) = 1 ^ B(k, j) = 0)  M(i, j) = 0;
        end
    end
end
```

Figure 20: The refinement procedure

To illustrate the usefulness of our refinement procedure, we show an example in Figure 21. Figure 21(a) shows the library graph T and subject graph G. Figure 21(b) shows the adjacency matrix for both graphs and the initial mapping matrix Φ_0.



(a)



(b)

Figure 21: The library graph and subject graph and the initial permutation matrix.

Next we illustrate how the search space is being explored and pruned at each depth. The complete search tree is given in Figure 22 and the exploration path and pruning at each depth is clearly labeled. Given the initial permutation matrix as shown in Figure 21(b), the algorithm first enter depth=1 and pick up the first unoccupied column that is one, in this case *k=1*. All remaining columns that are one in the same row are set to 0. The new permutation matrix after step 1 is obtained as follows:

1	0	0	0	0	0
0	1	0	1	0	0
0	1	1	1	1	0
0	1	1	1	1	0

Now the refinement procedure is applied to node 1 in T and node 1 in G. The adjacency list of node 1 of T is {2, 3} and that of node 1 of G is {2, 3}. There are only four possible mappings: $2 \to 2; 2 \to 3; 3 \to 2; 3 \to 3$, hence we can safely eliminate $\Phi(2,4), \Phi(3,4), \Phi(3,5)$. The permutation matrix after the refinement procedure is shown below:

1	0	0	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	1	1	0

Now the algorithm advance to the second row (*d=2*), where there is only one candidate *k=2* left to choose. Again we apply refinement procedure to node 2 in T and node 2 in G. and $\Phi(4,2), \Phi(4,3)$ are set to 0.

1	0	0	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
0	0	0	1	1	0

Next the algorithm advance to third row *d=3*. In this case, although there are two candidates *k=2* and *k=3*, the only valid one to choose is *k=3*, as the second column is already occupied at *d=2*. Note since node 3 in T has no successors, no refinement is needed.

1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	1	0

Finally the algorithm reach the leaf node of the search tree (*d=4*) and it picks *k=4*. A candidate permutation matrix is formed as follows:

1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0

Now the algorithm is at a leaf node of the search tree and the candidate permutation matrix is checked against condition (3). The above permutation matrix is valid and corresponds to the matching {(1,1), (2,2), (3,3), (4,4)}. The match is recorded and the algorithm starts to backtrack to check other candidate permutation matrices.

Figure 22: Pruning of binary search tree.

The original full enumeration algorithm generates $2 \times 2 \times 4 \times 4 = 64$ different candidate permutation matrices and tests each for validity. On the other hand, if the refinement procedure is applied, the search algorithm would reach the leaf of the search tree twice only. Thus the total number of candidate permutation matrices to be test for validity is also reduced to 2. This simple example illustrates the effectiveness of the refinement procedure in pruning the search tree.

In Ullmann's original algorithm, a refinement procedure that does a simple check on adjacencies was presented. The refinement procedure is effective in eliminating invalid node-to-node mappings before any k at each depth is picked. We call Ullmann's refinement procedure "prior-refinement". Interested readers may refer to [Ullmann] for details. On the other hand, our refinement procedure works in a different way: it eliminates invalid node-to-node mappings after a valid k has been picked. We call our refinement procedure "post-refinement". We apply both prior-refinement and post-refinement to prune the search space to the best extend. As both refinement procedures eliminate invalid 1s in Φ, it is possible that after such elimination, some row may not contain any 1 at all, i.e. condition (2) is violated. If such a violation occurs before any search starts, it is sufficiently safe to conclude no valid sub-graph isomorphism exists. On the other hand, if such a violation occurs in the middle of a search path, it indicates there is no necessity to continue searching along this search path and the algorithm backtracks. To facilitate such operations, both refinement procedures return a FAIL status flag if

condition (&) is violated as a consequence of refinement.

5.2.3 Convexity checking

Although a permutation matrix Φ that satisfies conditions (2) and (3) represents a correct sub-graph isomorphism, there is no guarantee on the obtained sub-graph to be logically correct, i.e. the convexity condition may not be satisfied. If care is not taken this may result in generating invalid program code. To illustrate this, an example is presented in Figure 23.

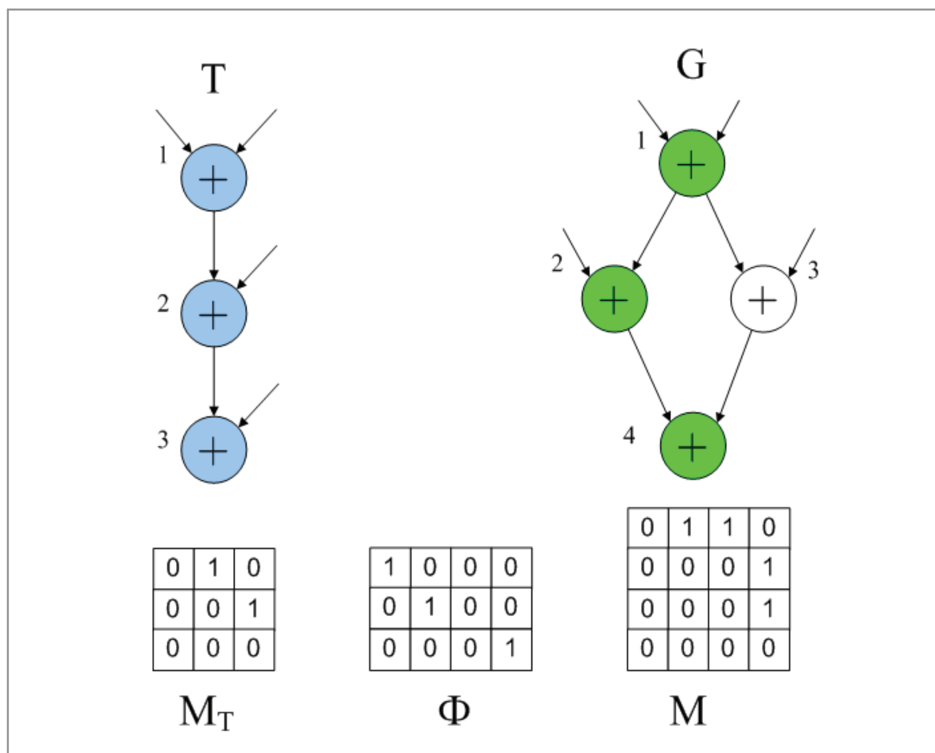


Figure 23: Sub-graph isomorphism that violates the convexity constrain

It is easy to verify the permutation matrix Φ given in Figure 23 satisfies $M_T = \Phi M \Phi'$. This permutation matrix specifies a mapping from the

nodes {1, 2, 3} in T to the nodes {1, 2, 4} in G.. However, the subject graph G has an "O" type structure such that the induced sub-graph G' by the nodes {1, 2, 4} is not a convex pattern. It is obvious that there is a forward dependency from G' to node 3 and vise versa.

To ensure only logically valid matches are generated, a convexity checking is applied after each permutation matrix is found. This convexity checking procedure is similar to the one applied during pattern generation stage.

Finally, the complete sub-graph isomorphism algorithm is presented in Figure 24.

1. $d = 1, \Phi = \Phi_0, K = \{0\}, b = \{1\};$
   $prior_refinement(\Phi), if$ FAIL terminate program;
2. $store \quad \Phi_d = \Phi;$
3. $k = K[d];$
   $if\ there \quad is \quad no \quad j > k \quad s.t. \quad \Phi(d, j) = 1 \wedge b[j] = 1, goto \quad 5.$
   $pick \quad first \quad k \quad s.t. \quad \Phi(d, k) = 1 \wedge b[k] = 1;$
   $for \quad all \quad j \neq k, set \quad \Phi(d, j) = 0$
   $if \quad S_{d,d}(M_T) \neq S_{d,d}(\Phi M \Phi'),$ terminate program;
   $prior_refinement(\Phi),$ if FAIL terminate program;
   $post_refinement(\Phi),$ if FAIL terminate program;
4. $K[d] = k;$
   $if \quad d < m, \quad b[k] = 0; d = d + 1; goto \quad 2;$
   $else \quad if \quad M_T = \Phi M \Phi', report \quad \Phi \quad valid;$
   $else \quad \Phi = \Phi_d, goto \quad 3;$
5. $if \quad d = 1$ terminate;
   $else \quad d = d - 1, k = K[d], b[k] = 1, \Phi = \Phi_d, goto \quad 3;$

Figure 24: The complete sub-graph isomorphism algorithm

5.3 Optimal instruction cover

Given the set of discovered custom instruction matches in the application program, the task of finding the optimal cover for code generation is non-trivial. Since custom instruction matches do not cross basic blocks, the first complexity reduction method is to perform the cover on a per basic block basis.

5.3.1 Problem formation

A subject DAG $G(V,E)$ corresponds to the DFG of a basic block. A pattern that contains only one node is called a trivial pattern. Each node $v_i, 1 \le i \le n$ of G represents a basic instruction and can be covered by either a trivial pattern or a set of custom instruction matches. The complete set of matches that covers any node $v \in V$ is denoted as $m_1, m_2, ..., m_q$. Each match m_j has an associated cost $c(m_j)$ and speedup saving $s(m_j)$. The cost is simply the hardware latency and the saving is the difference between hardware latency and software latency, as discussed in section 3.7. The optimal code generation problem or the optimal DFG covering problem is to select a set of matches $\{y_1, y_2, ..., y_k\} \subseteq \{m_1, m_2, ..., m_q\}$ such that the follow conditions are satisfied:

(1) All the nodes in G are covered: $y_1 \cup y_2 \cup ... \cup y_k = V$;

(2) Any node is covered by exactly one match;

(3) The total cost $\sum_{i=1}^{k} c(y_i)$ is minimized

Clark et. al. use an heuristic approach by assigning an desirability ordering to each custom instructions. If an elementary instruction can be covered by multiple custom instructions, the one with highest order is chosen. Cong et. al recasts the optimal code generation problem to Binate Covering, which was first applied to the DAG covering problem by Rudell [25] and Liao [17], etc. Binate covering is a NP-hard problem; nevertheless, much effort has been spent on finding the exact solution because of its wide applications. However, in our system, binate covering cannot be directly applied as it allows overlapped instructions.

In our work, we define a $n \times q$ cover matrix M whose elements are either 0 or 1. Each row of M represents a node in G and each column of M represents a successful match instance. If match m_j covers node v_i, the corresponding entry is set to 1, i.e.:

$$M(i,j) = \begin{cases} 1 & if \quad v_i \in m_j \\ 0 & otherwise \end{cases}$$

Figure 21 gives an example on cover matrix. On the left side of Figure 21 is the DAG and all possible matches. Note m_1, m_2, m_3 are custom patterns where as $m_4 - m_9$ are trivial patterns. The corresponding cover matrix is on the right upper corner of Figure 25.

A valid cover scheme is represented by selecting columns such that for each row, there is exactly one 1 selected. The cost associated with this cover scheme is hence the sum of individual costs of the selected columns. An optimal cover

scheme is thus the one yields the lowest cost.

We construct a branch-and-bound algorithm to find an exact optimal solution. Note there might be more than one optimal solutions and our algorithm would simply find one of them.

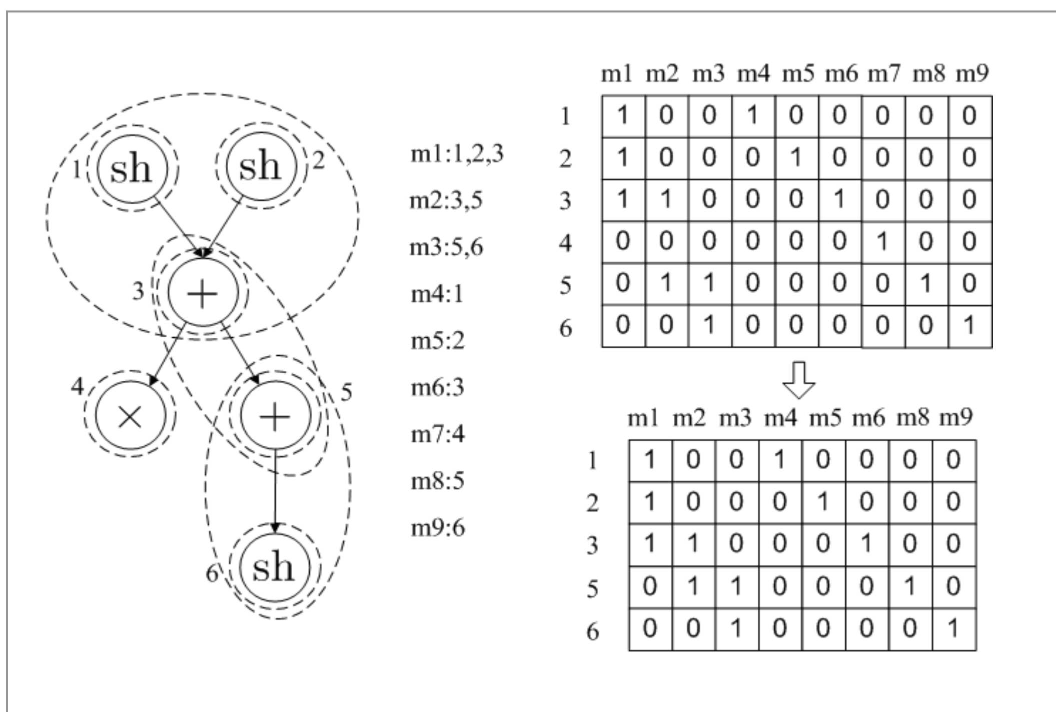


Figure 25: Cover matrix and pre-processing

5.3.2 Pre-processing

Before the optimal covering algorithm is applied, it is often possible to perform cover matrix reduction. The idea is simple, if a node v is not covered by any of the custom instructions, we have no choice but cover it use trivial patterns. The trivial pattern that covers this node corresponds to an essential column. The essential columns and the rows they cover can be removed directly. The optimal covering

algorithm will be performed on the reduced cover matrix. Finally, the cost of the essential columns will be directly added to the cost of the reduced matrix to obtain the actual cost associated with the original cover matrix. This reduction is called pre-processing.

In figure 25, there is no custom pattern covers node 4, hence m_7 is an essential column. After pre-processing, the reduced matrix is shown on the right-bottom of figure 25. The effectiveness of this processing procedure is not very obvious in this example, however, in practice, it is a simple yet useful technique, considering the complexity of the branch-and-bound algorithm is exponential to the number of columns in the worst case.

5.3.3 Heuristically search for an initial solution.

The performance of the *branch-and-bound* algorithm in general depends on the quality of bound estimation and how fast a relatively good solution can be obtained. It would not be wise to start with zero solutions, instead a good guess as the starting point helps reduce subsequent searching efforts. In order to find a high quality initial cover, we apply a greedy selection procedure. The matches are sorted according to saving/cost ratio and the greedy selection procedure chooses from the highest priority to the lowest sequentially. Whenever $y \in m$ is selected, for each node v_i it covers, we remove the corresponding row from the covering matrix. The column corresponds to y is also removed. This step corresponds to

line 4-5 in figure 26 where we use $\text{cov}\{y\}$ to denote the set of rows that are covered by y.

To ensure each node is covered by exactly one match, for each node $v_i \in m_j$, all other columns m_k are checked. If v_i can be covered by m_k as well, the column corresponds to m_k is also removed. This corresponds to line 7-11 in figure 26.

```
< Initial_Cover >
X : {v_1, v_2, ..., v_n};
m : {m_1, m_2, ..., m_q};
01.   Y ← ∅;
02.   while  X ≠ ∅  do
03.       select  y  s.t.  prio(y) = max_{∀m_i ∈ m}{prio(m_i);
04.       X ← X − cov(y);
05.       m ← m − {y};
06.       Y ← Y ∪ {y};
07.       forall  v ∈ y
08.           forall  y' ∈ m
09.               if(v ∈ y')  m ← m − {y};  end;
10.           end;
11.       end;
12.   end;
13.   return  Y;
```

Figure 26: The algorithm to find the initial cover.

5.3.4 Lower bound calculation

We use the variable X to denote the set of rows in the cover matrix, and Y, the set of matches to choose from. As the *branch-and-bound* algorithm traverse the search space, we use CPC to denote the total cost associated with the current path. We further maintain a global minimum cost GMC, which is at first initialized to

the cost of the initial cover in step 1. As the *branch-and-bound* algorithm traverses the search space, if a better solution is found then GMC is updated accordingly. The lower bound of the remaining uncovered nodes is denoted as LBC. The cover problem at any intermediate nodes of the search tree is denoted as: $C = < X, Y, CPC, LBC, GMC >$.

If $CPC + LBC > GMC$, it indicates continuing searching any descendants cannot yield better solutions than the current best solution in hand, thus we bound at the current location. Otherwise we branch. It is straight forward to calculate the cost along the search path; however, it remains a challenging issue to calculate the lower bound. The lower bound should be as tight as possible so as to prune the search tree as early as possible.

Definition 1:

The weight of a row is the minimum cost among all the matches that covers that row.

$$Weight(v_i) = \min_{\forall j, M(i,j)=1} Cost(m_j)$$

Definition 2:

$\tau(x) = X \cap \bigcup_{y \ni x} y$, which is the union of the nodes of each y that covers x. In other words, $\tau(x)$ is the set of nodes that potentially can be covered if we branch to some y that covers x.

Definition 3:

An independent set X' of X w.r.t to τ is a subset of X such that any two different rows $x_1^{'}$ and $x_2^{'}$ satisfies $x_1^{'} \notin \tau(x_2^{'}), x_2^{'} \notin \tau(x_1^{'})$.

From the definition of $\tau(x)$, it is clear that no single column y can cover both $x_1^{'}$ and $x_2^{'}$. Moreover, from the definition of weight, it covers any two rows from the independent set, and the cost is greater than the sum of their weights, i.e. $Cost(\{x_1', x_2'\}) \geq Weight(x_1') + Weight(x_2')$. The following lemma can be proved:

Lemma: $Cost(X) \geq \min\{Cost(X')\} = \sum_{x' \in X'} Weight(x')$

Therefore, the lower bound is set to the cost of covering X'. However, the problem of finding an independent subset that maximizes this bound is NP-hard. In order to obtain a reasonably good lower bound in the shortest time, the optimality of finding maximum independent subset is compromised and a greedy algorithm is applied. It is worth noting that this tradeoff only affects how the search tree is pruned but not the optimality of the DAG covering problem itself.

```
< independent_subset(X) >
1.   X' ← ∅;
2.   while  X ≠ ∅  do{
3.       x' ← an  element  of  X;
4.       X ← X − τ(x');
5.       X' ← X' ∪ {x'};
6.   }
7.   return  X';
```

Figure 27: The greedy algorithm that finds an independent subset of the rows X

The algorithm presented above guarantees the selected rows in X' are independent. Suppose there exists two elements $x_1^{'}, x_2^{'} \in X', s.t.\ x_1^{'} \in \tau(x_2^{'})$, from the definition of $\tau(x)$ there exists an column $y, s.t.\ x_1^{'} \in y, x_2^{'} \in y$, we have $x_2^{'} \in \tau(x_1^{'})$. Line 4 in the above algorithm removes $\tau(x_1^{'})$ if $x_1^{'}$ is selected first, or $\tau(x_2^{'})$ if $x_2^{'}$ is selected first, hence it is impossible for $x_1^{'}, x_2^{'}$ to be both selected. It follows that all the elements in X' are independent.

Finally, the lower bound cost LBC is simply the sum of the weights of all the rows in X':

$$LBC = \sum_{x \in X'} Weight(x)$$

5.3.5 Sub-problem formation

Similar to what is presented in chapter 3, if there is potential to continue branching at any intermediate node of the search tree, the cover problem at that node is divided into two sub-problems. First, the algorithm arbitrarily selects a

column y from the remaining columns, and then the two sub-problems are:

Sub-problem 1:

Consider y being included in the final solution, we shall remove column y from Y and remove all rows that are covered by y from X. In addition, if there is any unselected column $y' \in Y$ that is overlapping with y, it is removed immediately. Thus the sub-problem is represented as:

$$C1 = < X - \text{cov}(y), Y - \{y\} - overlap(y), CPC + \cos t(y), GMC >$$

Sub-problem 2:

Consider y being excluded from the final solution, i.e. y is simply discarded. We shall remove column y from Y, and the sub-problem is:

$$C1 = < X, Y - \{y\}, CPC, GMC >$$

5.3.6 The branch-and-bound algorithm for optimal cover

Figure 23 shows the overall pseudo code of the branch-and-bound algorithm that finds an optimal cover, given an initial cover. Line 2 corresponds to that a better solution is found and the global minimum cost is updated. Line 4-8 corresponds to finding the lower bound. It is worth noting the weight of each row has to be calculated in each recursive call, since the columns are changing. Line 9-21 corresponds to continue branching at the current node.

$$
\begin{array}{ll}
& <\text{branch_and_bound}: X, Y, CPC, GMC> \\
01. & \quad if \quad (X = \emptyset) \wedge (CPC < GMC) \quad then \\
02. & \qquad GMC = CPC; \\
03. & \quad elseif \quad X \neq \emptyset \quad then \\
04. & \qquad X' = \text{independent_subset}(X); \\
05. & \qquad for \quad each \quad x' \in X' \\
06. & \qquad\quad Weight(x') = \min_{\forall y \in Y} \quad Cost(y) \\
07. & \qquad end \\
08. & \qquad LBC = \sum_{x' \in X'} Weight(x'); \\
09. & \qquad if \quad CPC + LBC < GMC \quad then \\
10. & \qquad\quad select \quad y \in Y, Y \leftarrow Y - \{y\}; \\
11. & \qquad\quad duplicate \quad Y' = Y; \\
12. & \qquad\quad forall \quad x \in y \\
13. & \qquad\qquad forall\ y' \in Y \\
14. & \qquad\qquad\quad if \quad x \in y' \quad then \quad Y \leftarrow Y - \{y'\}; \quad end; \\
15. & \qquad\qquad end \\
16. & \qquad\qquad X \leftarrow X - \{x\}; \\
17. & \qquad\quad end; \\
18. & \qquad\quad duplicate \quad X' = X; \\
19. & \qquad\quad call \quad < X, Y, CPC + \text{cost}(y), GMC >; \\
20. & \qquad\quad call \quad < X', Y', CPC, GMC >; \\
21. & \qquad end; \\
22. & \quad end;
\end{array}
$$

Figure 28: The branch-and-bound algorithm that finds the optimal cover

5.4 Code emission

After the optimal DAG cover is obtained, the follow up and also the final step is actual code emission. However, since there is no restriction on custom instruction structures, provided the convexity and input-output constrains are satisfied, reordering of instructions may be necessary. This issue is addressed in [26] and interested readers may refer to it.

5.5 Conclusion

In this chapter, the application mapping problem is discussed. The importance of application mapping is due to the fact the software rarely runs on the custom processor once and away. Thus the approach that combines pattern generation, selection and binary code modification into one shot, as in some pervious works, is of little practical usage. In this work, the application mapping problem is decomposed into two sub-problems: custom instruction matching and optimal code generation. For the former, Ullmann's general graph matching algorithm is employed as the basis and new refinement procedures are added to effectively prune the search space. For the latter, a branch-and-bound algorithm is formulated to find the optimal DAG cover from the pool of custom instruction matches and trivial patterns. Effective pre-processing procedures and lower bound calculation for the branch-and-bound algorithm are presented.

Chapter 6: Experimental Results

6.1 Environment, libraries and third-party packages

The automation system presented in the previous chapters is implemented in a standard Linux/C++ environment. In addition, to facilitate efficient implementation, we used three third-party software packages. Besides the SimpleScalar simulation framework and the Nauty graph isomorphism library, the LEDA graph library is used. As the core of the algorithm is combinatory programming, efficient and easy-to-use data structures for graphs and parameterized graphs are of primary concern. The LEDA library [21] is specially designed for applications in graphs, geometric computations, combinatorial optimization and other. It offers a variety of relevant building blocks that are needed in our system. More specifically, it provides object based data structures including not only graphs, but also queues, linear lists, and hash tables etc.

As a side note, at the time this thesis is written, the newer versions of the LEDA library are commercialized. In our work, we used version 4.2 which is free for academic researchers.

6.2 Benchmark programs

The benchmark programs used in this work comes from two sources: MiBecnh and the H.264/AVC [15] reference software JM8.6. The details of the benchmarks

used in this work are listed in table 5.

Table 5: List of benchmark programs

	Benchmark	Domain	Maximum basic block size
MiBench	dijistra	Network	24
	patricia	Network	46
	sha	Security	31
	crc32	Telecom	14
	FFT	Telecom	57
	IFFT	Telecom	57
	rawcaudio	Telecom	12
	rawdaudio	Telecom	11
	bitcnts	Automotive	46
	basicmath	Automotive	52
H.264/AVC	Encoder	Multimedia	256

Instead of performing the algorithm on each individual basic blocks, we purposely masked out some basic blocks belong to system libraries such as I/O processing (e.g. file processing), memory management (e.g. malloc, memcpy, etc), etc. However, basic blocks belong to arithmetic related libraries, such as the math library, or the low level multiple-precision arithmetic library, are not filtered. This filtering process helps to avoid spending time in non-profiting basic blocks. For example, for benchmark "sha", the number of remaining basic blocks after filtering is only 59 whereas the original total number is 471. In additional, we want to see the true speed up from the application's native code. Hence, the

filtering process also helps to remove system libraries' interferences.

All benchmarks are compiled using the SimpleScalar ported gcc (gcc-2.7.2.3) with their standard compiling options, e.g. –O3 for MiBench.

6.3 Speedup ratio calculation

The speedup ratio is calculated over all valid basic blocks. Using T_{sw} and T_{hw} to represent a basic block's old execution cycles and new execution cycles after custom instruction mapping, we have the following formula:

$$Speedup \quad ratio = \left(\frac{old \quad execution \quad cycles}{new \quad execution \quad cycles} - 1 \right) \times 100\%$$
$$= \left(\frac{\sum_i T_{sw}(BB_i) \cdot freq(BB_i)}{\sum_i T_{hw}(BB_i) \cdot freq(BB_i)} - 1 \right) \times 100\%$$

6.4 The effects of input output constraints

To evaluate the effects of input-output constraints, experiments are performed on the seven benchmark programs from MiBench. The input constraint varies from 3 to 8 and the output constraint varies from 1 to 3. When the output constraint is set to 1, the generated custom instructions are MISO patterns. Figures 29-37 show the speedup against all input-output constraint configurations for each benchmark

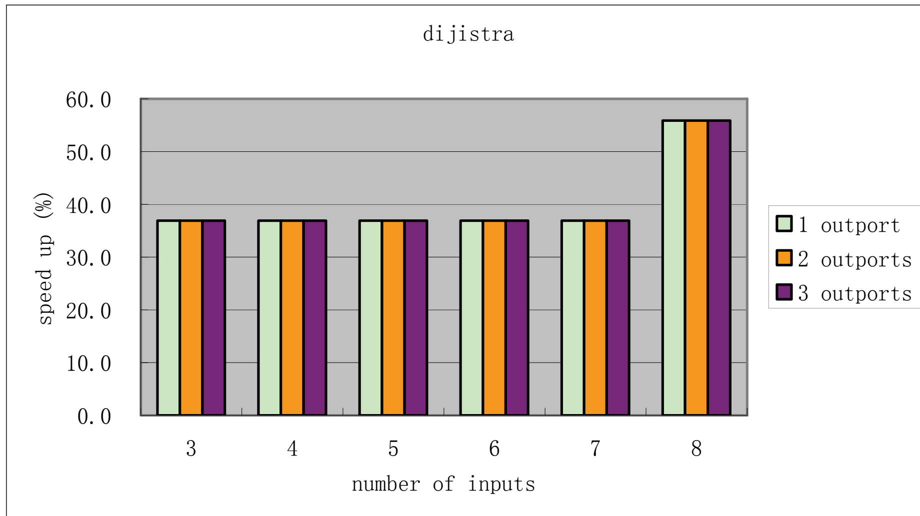


Figure 29 : Dijistra: speed up vs. different input-output constrains.

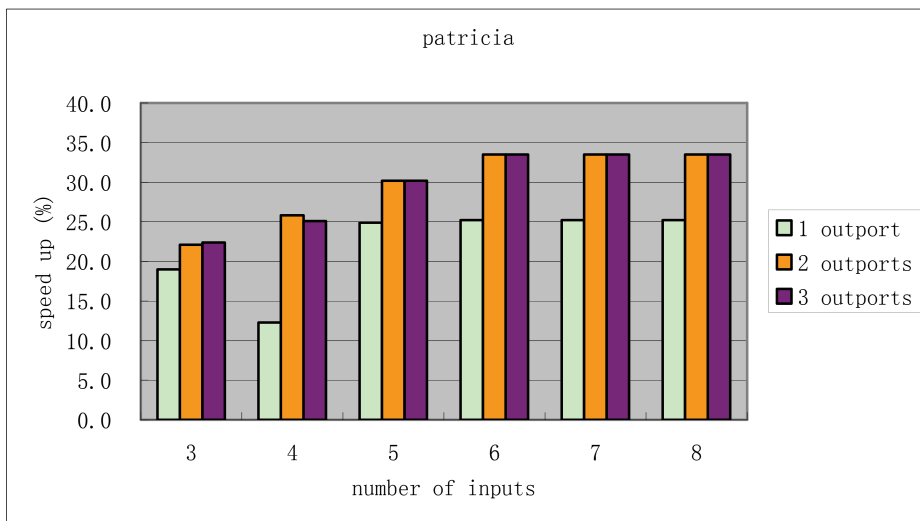


Figure 30: Patricia: speed up vs. different input-output constrains.

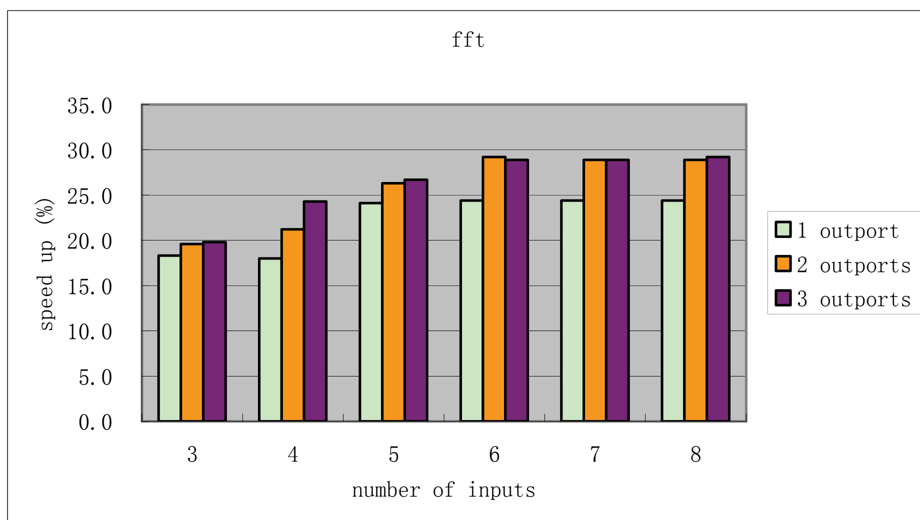


Figure 31: FFT: speed up vs. different input-output constrains.

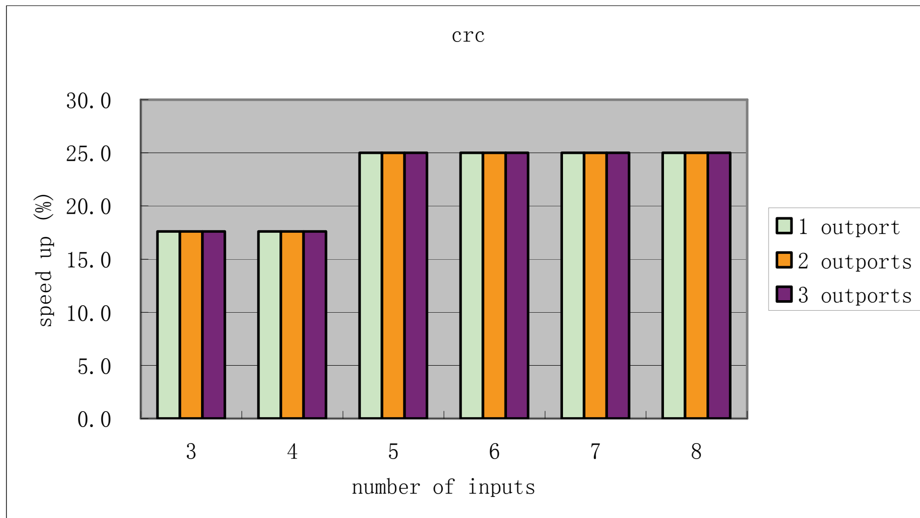


Figure 32: Crc: speed up vs. different input-output constrains.

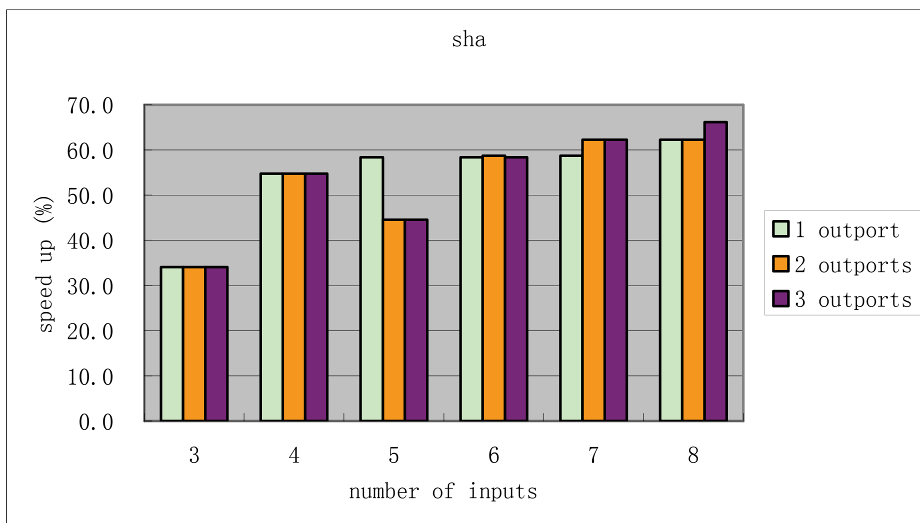


Figure 33 : Sha: speed up vs. different input-output constrains.

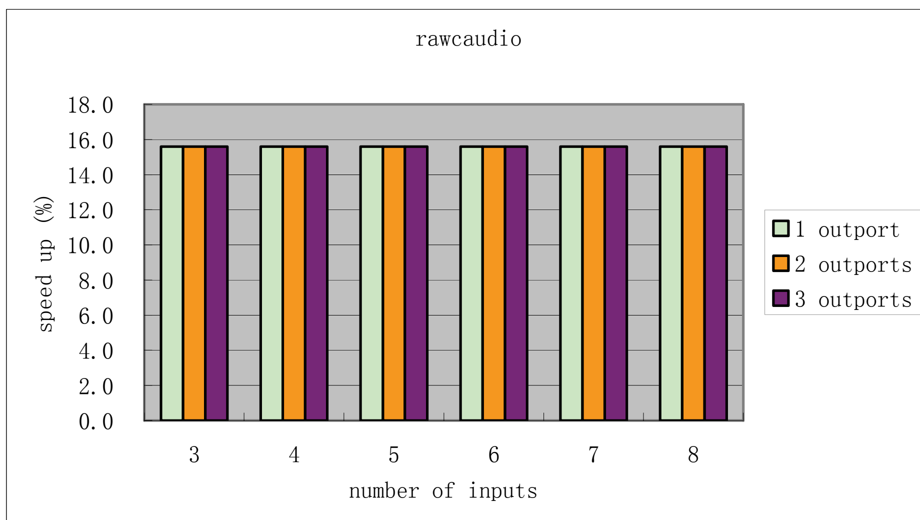


Figure 34 : Rawcaudio: speed up vs. different input-output constrains.

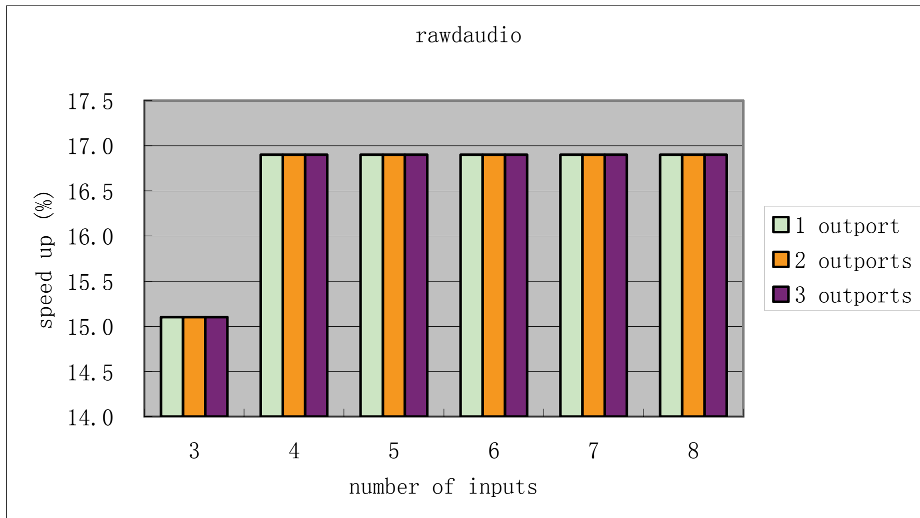


Figure 35: Rawdaudio: speed up vs. different input-output constrains.

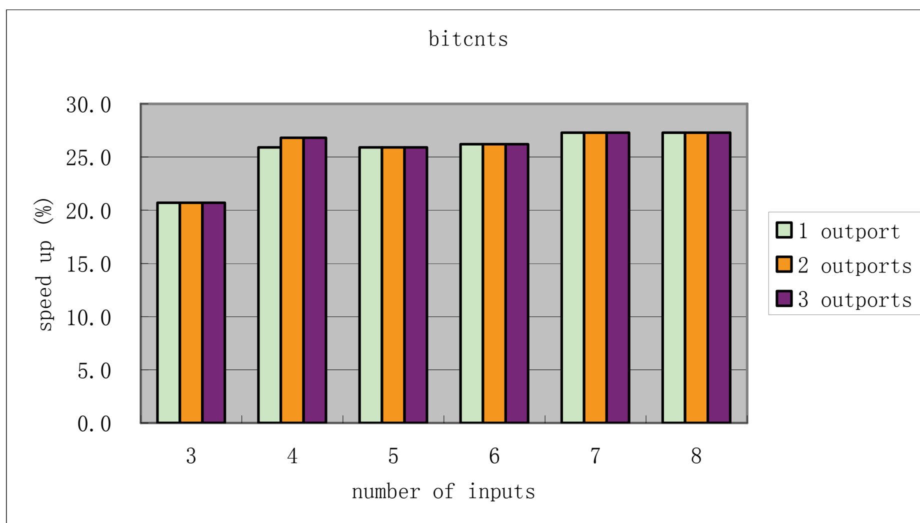


Figure 36: Bitcnts: speed up vs. different input-output constrains.

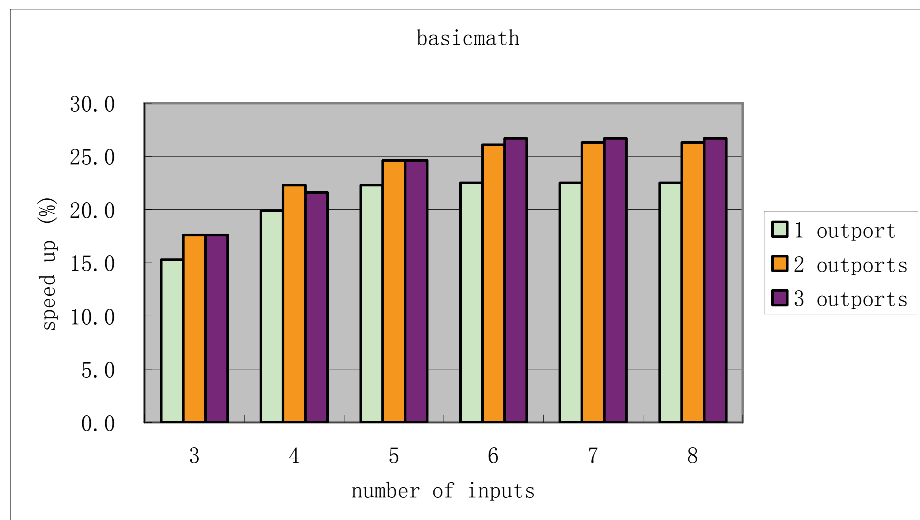


Figure 37: Basicmath: speed up vs. different input-output constrains.

6.4.1 Input constraint

For most benchmark programs, the speedup increases as the input constraint is relaxed, e.g. “patricia”, “FFT”, “sha”, and “basicmath”. Input constraint is closely related to custom instruction size. By relaxing the input constraint, larger patterns can be discovered. In general, large patterns are more “economic” as they pack a large number of instructions in only a few processor cycles. However, speedup becomes saturated as more inputs are allowed. The reason is as pattern becomes larger, it is more difficult to find instructions that can be executed on the custom hardware.

Some benchmarks, such as “crc”, “rawcaudio”, “rawdaudio” and “bitcnts”, the speedup saturates very early. An extreme example is “rawcaudio”, where all custom instructions are found when the input constraint is set to 3. There is no additional speedup obtained as input constraint increases from 3 to 8.

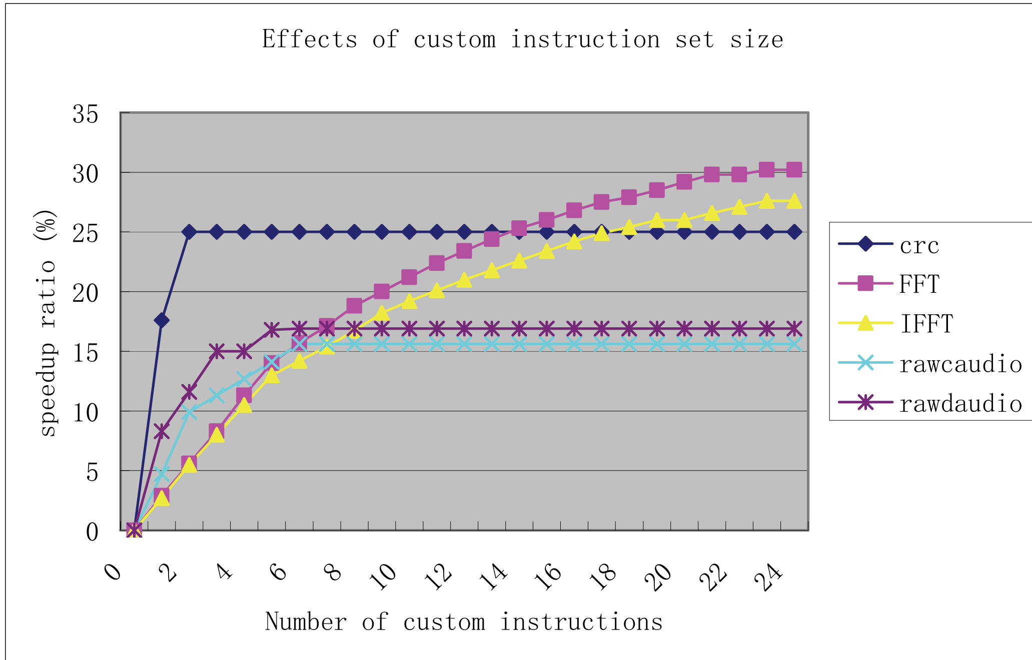
6.4.2 Output constraint

In almost all cases, there are no noticeable speedup differences between 2-output and 3-output custom instructions. This observation suggests for real applications, the output constraint can be set to 2 as an optimal balance between efficiency and accuracy. On the other hand, MISO custom instructions (1-output) may result in inferior performance. For instance, in “patricia”, the speedup of 1-output configuration underperforms 2-output configuration by 10%-15%. In “FFT” and “basicmath”, the measured difference is about 5% for all input configurations.

Although in other benchmarks 1-output configurations performs as well as other configurations, we believe, MIMO patterns should always be used to ensure guaranteed performance.

6.5 Effects of number of custom instructions

In some extensible instruction set processors, there is a limit on the total number of custom instructions. In this simulation, we vary the library size from 1 to 25 and observe the corresponding speedup ratios. To prevent any performance limitation due to input-output constraints, we set it to the maximum case, 8-input-3-output. The results are shown in Figure 38 below. It is clear for all benchmarks, at least 90% of the maximum speedup can be achieved with library size limit set to 25. In general, all the curves rise rather fast for the first few custom instructions (<8), and the rising speed decreases as additional custom instructions are added. This phenomenon matches the nature of our greedy pattern selection algorithm: patterns that have greater speedup potential are selected first. For some benchmarks (crc, rawcaudio, rawdaudio, sha, dijistra, and bitcnts), the speedup ratio saturates in about 10 custom instructions, hence, resulting in very steep curves. On the other hand, the speedup ratio of other benchmarks (FFT, IFFT, patricia, and basicmath) keeps increasing until the maximal library size, but at a much slower increasing rate.



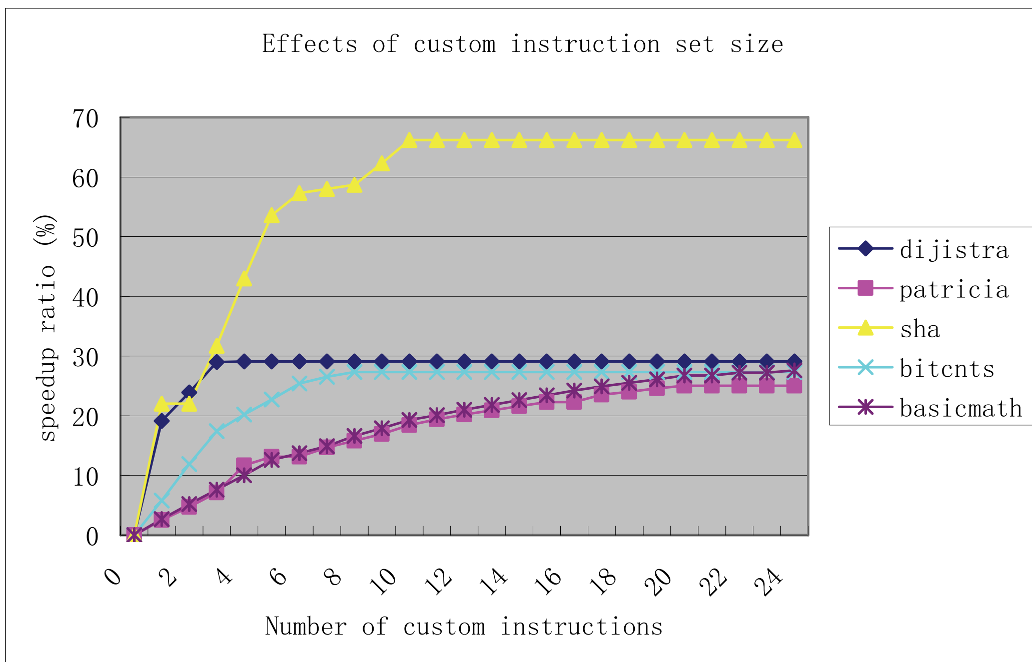


Figure 38: Effects of custom instruction set size

6.6 Cross-application mapping

First addressed in [4], cross-application mapping or cross-compilation can be used to examine the generalizability of custom instructions among applications in the same domain. The domains of the benchmarks are shown in Table 5 and we

perform cross-compilation within each domain accordingly. In addition, a few simulations are performed on selected cross-domain benchmarks. For all simulations in this section, other constraints are relaxed, i.e. maximum input-output constraint is used and there is no limitation on library size. Table 6 bellow gives the complete simulation list. For each source-target cross-compilation pair, the source benchmarks are used to generate the custom instructions and the target benchmark is then mapped to the generated instruction set. The Abbrev column gives the corresponding abbreviation that will be used in the figures later.

Table 6: The list of cross-compilations

Domain	Source	Target	Abbrev.
Network	patricia	dijistra	pat2dij
	dijistra	patricia	dij2pat
Automotive	bitcnts	basicmath	bit2math
	basicmath	bitcnts	math2bit
Telecom	FFT	IFFT	FFT2IFFT
	IFFT	FFT	IFFT2FFT
	rawcaudio	rawdaudio	rawc2rawd
	rawdaudio	rawcaudio	rawd2rawc
	FFT	crc	FFT2crc
	crc	FFT	crc2FFT
	FFT	rawcaudio	FFT2rawc
	rawcaudio	FFT	rawc2FFT
	crc	rawcaudio	crc2rawc
	rawcaudio	crc	rawc2crc
Cross-domain	sha	crc	sha2crc
	crc	sha	crc2sha
	basicmath	FFT	math2FFT
	FFT	basicmath	FFT2math

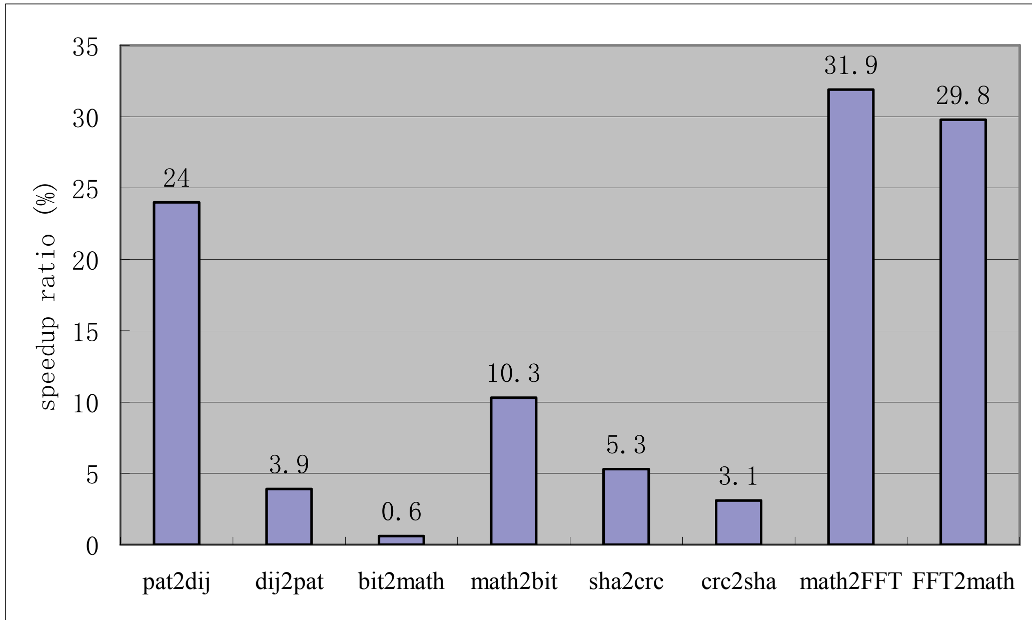


Figure 39: Speedup ratios of selected cross-compilation 1

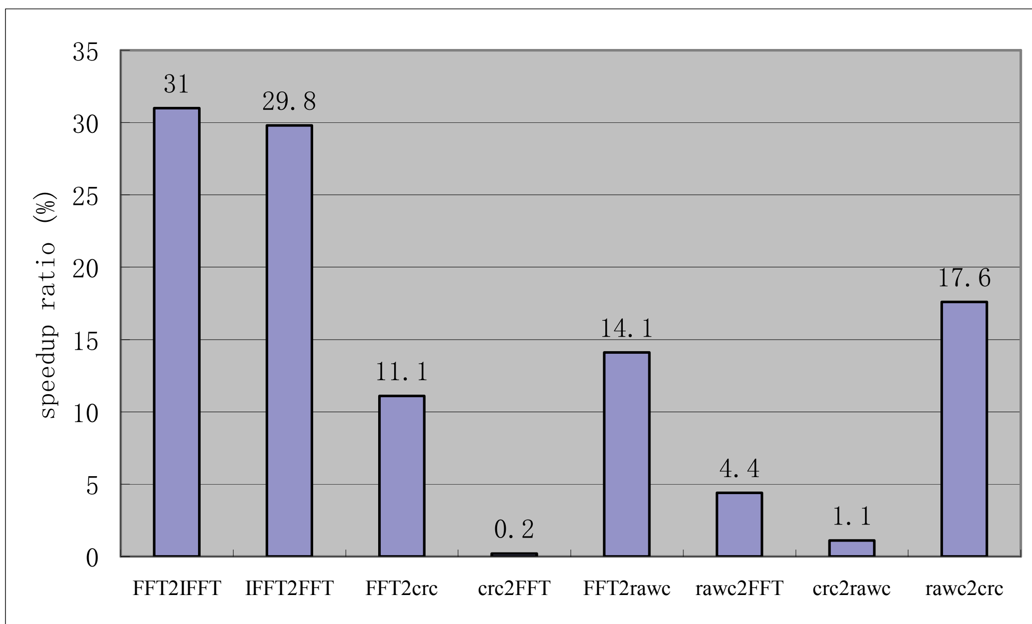


Figure 40: Speedup ratios of selected cross-compilation 2

Figures 39 and 40 show the speedup ratio of each cross-compilation benchmark pairs. Figure 39 covers “Network”, “Automotive”, and “Cross-domain” whereas figure 40 covers “Telecom”. In most cases the speedup-ratio for cross-compilation

is lower than that of native compilation. For instance, a maximum speedup of 55.9% is achieved for “dijistra-dijistra” native compilation whereas only 24% is achieved for “patricia-dijistra” cross compilation. Similarly, the “dijistra-patricia” cross compilation only achieves 3.9% whereas the native compilation achieves 33.5%.

However, there are a few cross-compilation pairs for which, the speedup-ratio is as good as their native compilation. For instance, the native compilation of “basicmath” has a record of 26.7%, whereas the “FFT-basicmath” cross-compilation achieves 29.8%. Considering the maximum library size for native compilation is set to 25 whereas there is no limit for cross compilation, the speedup ratios can be considered close. In additional, similar results are obtained for “FFT-IFFT”, “IFFT-FFT”, “basicmath-FFT”, and “FFT-rawcaudio”. In previous section it is observed the speedup ratio for “FFT” and “basicmath” increases gradually as number of patterns added. In deed, after a close look at the extracted custom instructions, it is observed that “FFT” and “basicmath” each generates a large number of small patterns. Hence, if these two are used for custom instruction generation, there is a higher chance that other applications can benefit from it. On the other hand, other applications, such as “sha” only generates 8 custom instructions and some are of larger size (largest one contains 7 instructions), it is hence much more difficult for other applications to benefit from these custom instructions.

6.7 Case study: H.264/AVC encoder

In this section, a case study on a practical application H.264/AVC is performed. H.264/AVC is an important video coding standard and it covers various application domains ranging from low bit-rate video-conferencing to high-quality multimedia entertainment.

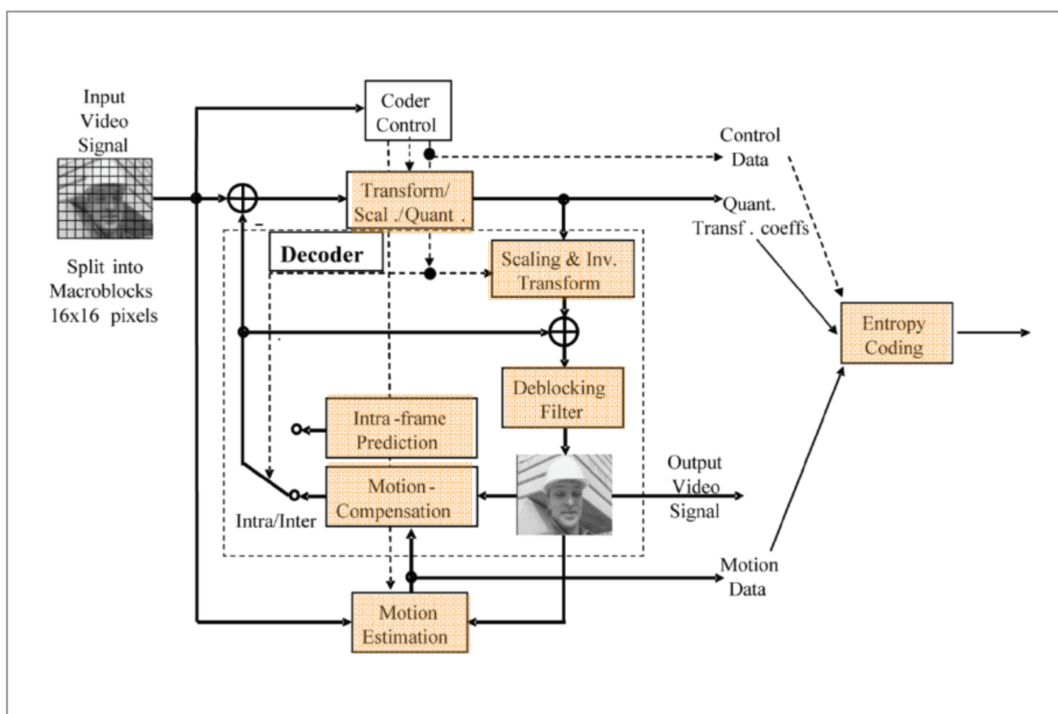


Figure 41: Basic coding structure for H.264/AVC for a macroblock.

Figure 41 shows the basic coding structure of a macroblock. The shaded blocks will be studied for custom instruction. In our experiments, the H.264/AVC standard reference software JM8.6 is complied using SimpleScalar ported gcc-2.7.2.3, with –O2 option. We simulated the encoder using the following configurations: Hadamard transform on, three reference frames for motion estimation, P-frames on, B-frames off, context adaptive binary arithmetic coding

(cabac) on, rate-distortion optimization on. We use the “forman” sequence in QCIF size as input. The “forman” QCIF sequence is a representative testing sequence for low bit-rate applications. There are 50 frames in total encoded.

To study a specific block in the application, we first analyze the source and find out the corresponding C/C++ functions. Since the text symbols are dumped into a file during program trace generation, we are able to identify the starting and ending addresses of each user functions. We manually find the address range of the target functions and use it as an argument of the custom instruction generation engine. The custom instruction generation engine would filter out basic blocks that are not in the address range; effectively it means only the target regions are explored. Similar filtering is performed in the application mapping stage.

We identified five interesting aspects of the H.264/AVC system as targets. The corresponding function names and address are given in Table 7 below. For a complete list of text symbols, their associated address, and size, etc, please refer to appendix A.

Table 7: H.264 building blocks, function names and address range

H.264 Building block		Function names	Address range
DCT, Quantization		dct_luma, dct_luma_sp, dct_chroma, dct_chroma_sp	00404630-0040D808
Motion Vector Estimation	Full-pel	FullPelBlockMotionSearch	00467230-00467FD8
	Sub-pel	SubPelBlockMotionSearch	004688C0-0046A3B0
	SAD	SATD	00467FD8-004688C0
Motion Compensation		LumaPrediction4x4, ChromaPrediction4x4, OneComponentLumaPrediction4x4, OneComponentChromaPrediction4x4, intrapred_luma, intrapred_luma_16x16, IntraChromaPrediction8x8, IntraChromaPrediction4x4	00401F90-00404630 004441C8-00445420 00445BA8-004470F8 00442750-00443328 00441FE8-004425D8
Debloking Filter		DeblockFrame, DeblockMb, GetStrength, EdgeLoop	0043D4A0-0043FAA0
Arithmetic coding (cabac)		biari_encode_symbol, biari_encode_symbol_eq_prob	00400F10-004019C0

The details of the simulation results are given in Table 8. For each simulation, we list out the total number of basic blocks fall in that address range, the maximum basic block size, the old execution cycles and the new execution cycles after custom instruction mapping, and finally the speedup ratio. The one that achieves highest speedup ratio is integer transform (DCT, quantization). The speedup ratio is 32.1% and it is similar to that of FFT in MiBench, as these two are similar in nature.

Table 8: the simulation results for H.264/AVC

H.264 Building block	**Number of BBs**	**Max BB size**	**Old exec. cycles**	**New exec. cycles**	**Speed-up ratio**
DCT, Quantization	231	83	12125777395	9177001147	32.1
Motion Estimation	109	256	17886087760	15723919802	13.8
ME Full-pel	10	57	6999213134	5953800228	17.6
ME Sub-pel	91	40	6124885118	5476646006	11.8
SATD	8	256	4761989508	4290551088	11.0
Motion Compensation	352	62	1313550736	1070812678	22.7
Deblock Filter	176	42	160275540	130182453	23.1
cabac	60	15	4122943229	3446346518	19.6

In the following part, we give the first four popular patterns for each building block. The patterns are arranged in decreasing popularity order from left to right.

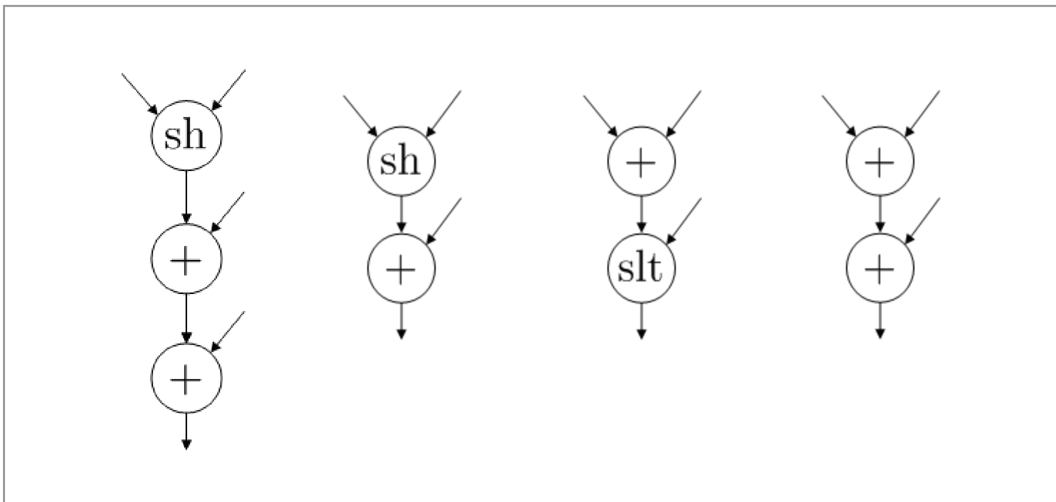


Figure 42: Four most popular patterns for DCT and Quantization

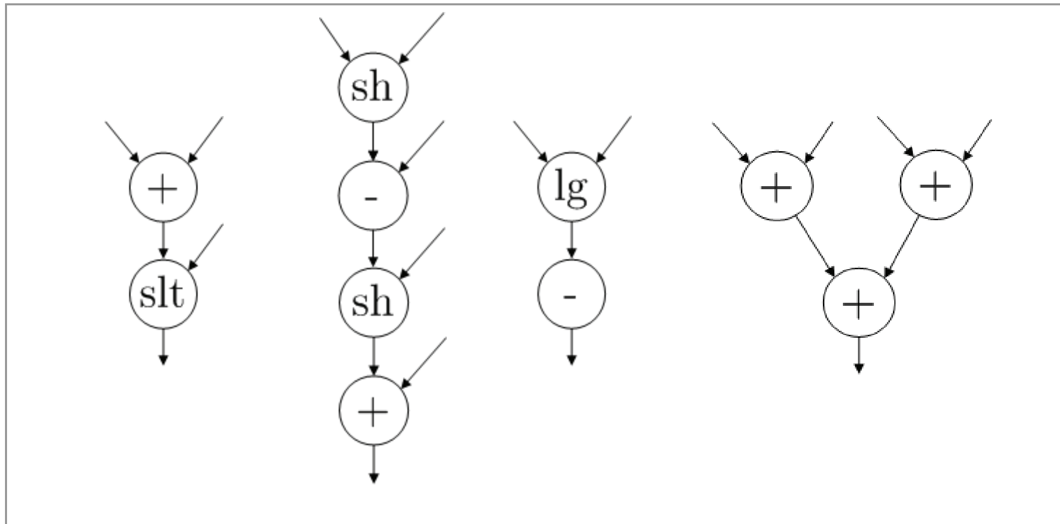


Figure 43: Four most popular patterns for Motion Estimation

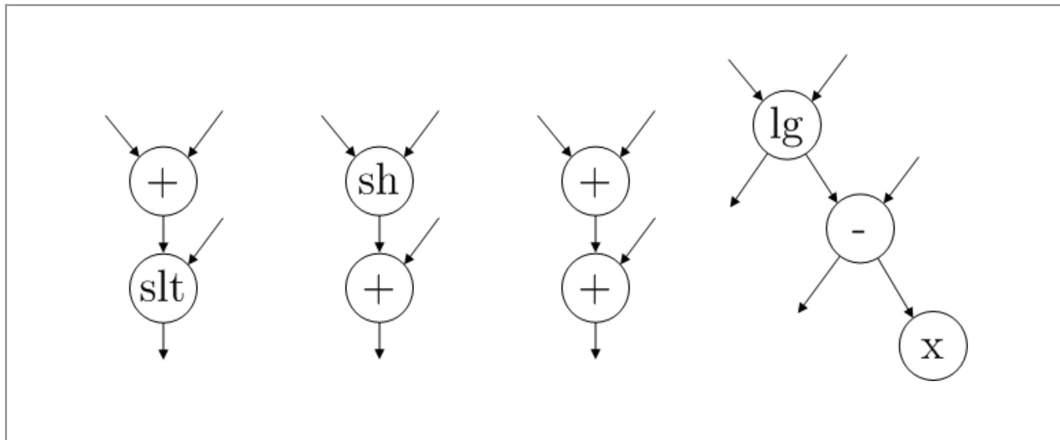


Figure 44: Four most popular patterns for Motion Compensation

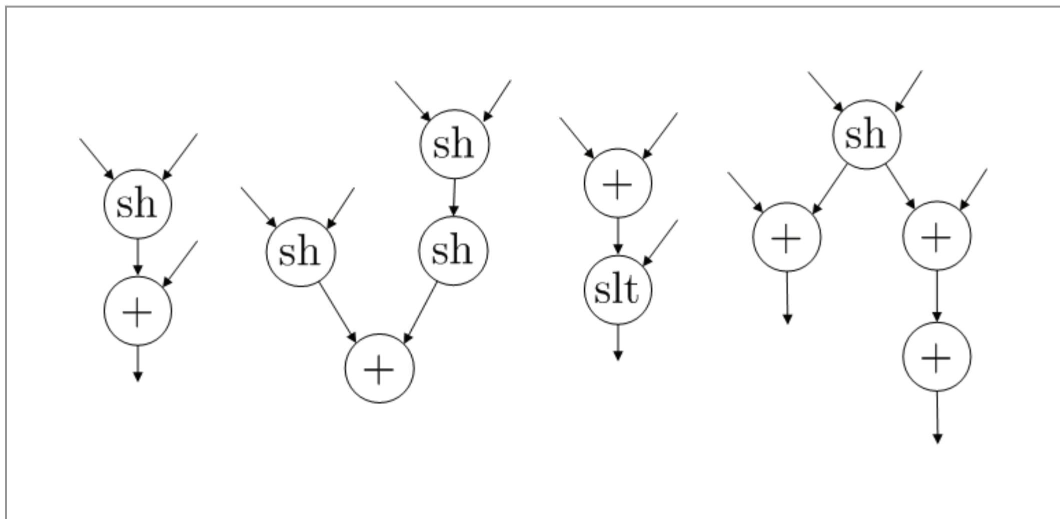


Figure 45: Four most popular patterns for Debloking Filter

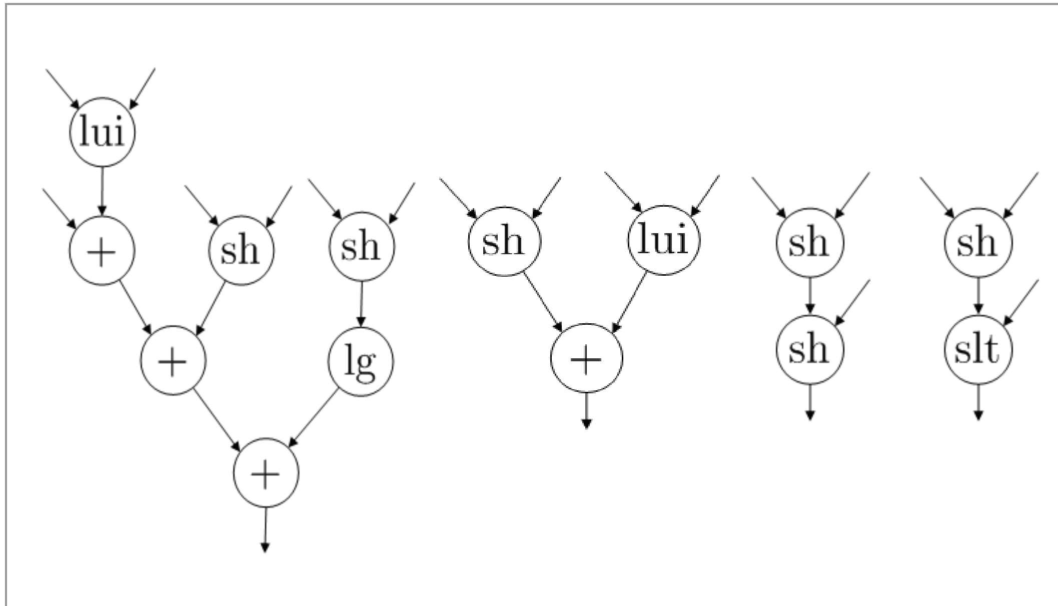


Figure46: Four most popular patterns for Arithmetic Coding (cabac)

Finally, we applied our system on the entire H.264/AVC software (i.e. including miscellaneous functions such as initializing coding parameters, writing bit streams, etc, and the overall speedup is measured to be 16.6%.

6.8 Conclusion

In this chapter, experiments are setup to show our system's ability in identifying efficient custom instructions. We first performed a series of experiments on selected benchmarks from MiBench. We analyzed the effects of input-output constraint, custom instruction set size, and cross-application mapping. Interested users may refer to [4] and [30] for further verification. Finally, we performed an case study on H.264/AVC reference software and showed the identified custom patterns. However, it should be noted that the H.264/AVC reference software is

none-optimized; hence the identified patterns given here may not be practically the best solution. Due to the time limits and the unavailability of highly optimized video codec, we have not applied our system to a more practical H.264/AVC implementation. It is suggested that it should be done in the future work.

Chapter 7: Conclusion

More and more embedded applications have stringent requirements in terms of high-performance and low-power. Examples are handheld devices and 3G hand-phones for which integrated camera and video coding now become basic requirements. While traditional DSP processors are difficult to meet these stringent demands, application specific instruction set processors are shown to be effective in meeting the performance and power demands. However, designing these custom instructions is traditionally done by experts and intensive manual work is necessary. Recently, researchers have been interested in designing automated methods that free people from such workloads. Standard approach is to explore the application's data flow graph and discover pattern candidates that can potentially improve performance if implemented in hardware. However, due to the NP-hardness of pattern enumeration problem, most proposed systems use heuristic methods to grow patterns from random seed nodes in the DFG. Moreover, the problem of application to custom instruction set mapping is avoided in most previous works. In this work, we propose an automated system that generates all valid patterns and performs the optimal application mapping.

In chapter 2 we introduced the trace collection and DFG construction methods. In chapter 3, Pan's improved full pattern enumeration method and its limitations are discussed. In chapter 4, we presented the pattern representation format and

canonical labeling using the Nauty graph library. A greedy algorithm is proposed to select the final set of custom instructions from a large candidate pool. The core of the greedy algorithm is the maximal speedup potential calculation of each pattern. We use the maximal speedup potential as a priority function that guides the greedy algorithm. In chapter 5, application mapping is discussed. We proposed a modified version of Ullmann's graph isomorphism algorithm to perform application matching. Finally, optimal code generation is achieved using a branch-and-bound algorithm that minimizes the total execution cycles. In chapter 6, experiment results are presented. We use MiBench to study the effects of input-output constraints, custom instruction set size and cross-application compilation. In addition, a case study on real applications, i.e. H.264/AVC is performed and results are presented. Experiments show that our system is able to identify the critical patterns and almost all applications can benefit from custom instruction and the speedup ratios are in the range 15%-70%.

We note the limitations of the current systems as follows:

- The pattern selection phase is sub-optimal. However, we believe no practical solution exists for exact optimal pattern selection.
- For difficult DFGs (please refer to chapter 2), the runtime of pattern enumeration phase may be impractical. We propose two possible improvements to get around this problem in future works:
    - Implement an efficient but not necessarily optimal method that identifies

isomorphic nodes in a pattern. As mentioned before, difficult DFGs usually posses high degree of regularity. If we can partition the nodes into equivalent groups, the complexity of enumeration can be greatly reduced.

  - A heuristic pattern generation algorithm, similar to those in [Nathan][Sun], should be built into the system, in parallel with the current full enumeration method. The system should be intelligent enough to switch between these two modes depending on the difficulty of the DFGs.

- Due to the time constraint, the application mapping algorithm has not been ported to real compilers. Instead we performed all simulation within the SimpleScalar framework. Nevertheless, the validity of the experiment results presented in chapter 6 is not affected.

Bibliography

[1] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," In *DAC*, 2003.

[2] D. Burger and T. M. Austin, "The SimpleScaler Tool Set", Version 2.0. Computer Architecture News, page 13-15, June, 1997

[3] H. Choi et al. "Synthesis of application specific instructions for embedded DSP software," *IEEE Transactions on Computers*, 48(6):603-614, June 1999.

[4] N. Clark, H. Zhong, and S. Mahlke, "Processor Acceleration Through Automated Instruction Set Customization," *International Symposium on Microarchitecture (MICRO-36)*, pp. 129-140, December 2003.

[5] J. Cong , Y. Fan , G. Han , and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," *Proceeding of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, February 22-24, 2004, Monterey, California, USA.

[6] L. Cordella et al, "Performance evaluation of the VF graph matching algorithm," *in Proceedings of the 10th ICIAP*, vol. 2, pp.1038-1041, IEEE Computer Society Press, 1999.

[7] D. G. Corneil and C. C. Gotlieb, "An efficient algorithm for graph isomorphism," *Journal of the ACM*,17:51-64, 1970.

[8] O. Coudert and J-C. Madre, "New Ideas for Solving Covering Problems," In *Proceedings of the 32nd Design Automation Conference*, pages 641-646, June 1995.

[9] J. Fu, "Pattern Matching in Directed Graphs," In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching*, *volume* 937 of Lecture Notes in Computer Science, pages 64–77. Springer-Verlag, 1995.

[10] M. R. Garey and D. S. Johnson, Computers and intractability: A guide to NP-completeness, Freeman, 1979.

[11] R. E. Gonzalez, "Xtensa: A Configurable and Extensible Processor," *IEEE Micro*, vol. 20(2), pp. 60-70, Mar. 2000.

[12] A. Gupta and N. Nishimura, “Characterizing the complexity of subgraph isomorphism for graphs of bounded path-width,” *In Proc. 15th Annual Symposium on Theoretical Aspects of Computer Science, volume 1046 of Lecture Notes in Computer Science*, pages 453-464. Springer-Verlag, 1997.

[13] M. R. Guthausch et al, “Mibench: A free, commercially representative embedded benchmark suite,” In *IEEE 4th Annual Workshop on Workload Characterization*, 2001. http://www.eecs.umich.edu/mibench/.

[14] L. J. Huang, and A. M. Despain, “Synthesis of Instruction Sets for Pipelined Microprocessors,” *31st Design Automation Conference*, pp.5-11, 1994.

[15] “Information technology - Coding of audio-visual objects - Part 10: Advanced video coding,” *Final Draft International Standard,* ISO/IEC FDIS 14496-10, Dec. 2003.

[16] J. Katzenelson, S. S. Pinter, and E. Schenfeld, “Type matching, type-graphs, and the Schanuel conjecture,” *ACM Transactions on Programming Languages and Systems*, 14(4):574-588, Oct. 1992.

[17] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang, “Instruction Selection Using Binate Covering for Code Size Optimization,” In *Proceedings of International Conference on Computer Aided Design*, pp. 393-399, Nov. 1995.

[18] A. Lingas, “Subgraph isomorphism for biconnected outerplanar graphs in cubic time,” *Theoretical Computer Science*, 63:295-302, 1989.

[19] A. Lingas and M. M. Syslo, “A polynomial-time algorithm for subgraph isomorphism of two-connected series-parallel graphs,” In *Proc. 15th Int. Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, pages 394{409. Springer-Verlag, 1988.

[20] B. D. McKay, “Practical Graph Isomorphism,” *Congressus Numerantium*, vol 30, pp. 45-87, 1981

[21] K. Mehlhorn, S. Näher, “LEDA: a platform for combinatorial and geometric computing,” *Communications of the ACM*, v.38 n.1, p.96-102, Jan. 1995

[22] B. T. Messmer, H. Bunke, “A decision tree approach to graph and subgraph isomorphism detection,” *Pattern Recognition*, vol. 32, pp. 1979-1998, 1999.

[23] MIPS Technologies, “MIPS R10000 Microprocessor User’s Manual, Version 2.0,” MIPS Technologies, 1996.

[24] Katta G. Murty, Operations research: deterministic optimization models, Prentice-Hall, Inc., Upper Saddle River, NJ, 1994

[25] R. L. Rudell, “Logic Synthesis for VLSI Design,” Ph.D. Thesis, U.C.Berkeley , ERL Memo 89/49, 1989.

[26] F. Sun , S. Ravi , A. Raghunathan , and N. K. Jha, “Synthesis of custom processors based on extensible platforms,” In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, p.641-648, November 10-14, 2002, San Jose, California.

[27] J.R. Ullmann, “An algorithm for subgraph isomorphism,” *Journal of the ACM*, 23(1):31-42, 1976.

[28] J. Valdes, R. E. Tarjan, E. L. Lawler, “The Recognition of Series Parallel Digraphs,” *SIAM J. Comput*. 11(2): 298-313, 1982

[29] P. Yu , T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," *Proc. of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, September 22-25, 2004, Washington DC, USA

[30] P. Yu , T. Mitra, "Characterization of Embedded Applications for Instruction-Set Extensible Processors," *41st ACM/IEEE Design Automation Conference (DAC)*, June 2004.

[31] Nauty Package, http://cs.anu.edu.au/people/bdm/nauty.

Appendix

Appendix A

** Text symbols sorted by address:
sym `WriteAnnexbNALU': text seg, init-y, pub-y, local-n, addr=0x004001f0, size=1024
sym `OpenAnnexbFile': text seg, init-y, pub-y, local-n, addr=0x004005f0, size=144
sym `CloseAnnexbFile': text seg, init-y, pub-y, local-n, addr=0x00400680, size=112
sym `arienco_create_encoding_environment': text seg, init-y, pub-y, local-n, addr=0x004006f0, size=128
sym `arienco_delete_encoding_environment': text seg, init-y, pub-y, local-n, addr=0x00400770, size=160
sym `arienco_start_encoding': text seg, init-y, pub-y, local-n, addr=0x00400810, size=112
sym `arienco_bits_written': text seg, init-y, pub-y, local-n, addr=0x00400880, size=72
sym `arienco_done_encoding': text seg, init-y, pub-y, local-n, addr=0x004008c8, size=1608
sym `biari_encode_symbol': text seg, init-y, pub-y, local-n, addr=0x00400f10, size=1528
sym `biari_encode_symbol_eq_prob': text seg, init-y, pub-y, local-n, addr=0x00401508, size=1208
sym `biari_encode_symbol_final': text seg, init-y, pub-y, local-n, addr=0x004019c0, size=1240
sym `biari_init_context': text seg, init-y, pub-y, local-n, addr=0x00401e98, size=248
sym `intrapred_luma': text seg, init-y, pub-y, local-n, addr=0x00401f90, size=7224
sym `intrapred_luma_16x16': text seg, init-y, pub-y, local-n, addr=0x00403bc8, size=2664
sym `dct_luma_16x16': text seg, init-y, pub-y, local-n, addr=0x00404630, size=7696
sym `dct_luma': text seg, init-y, pub-y, local-n, addr=0x00406440, size=4248
sym `dct_chroma': text seg, init-y, pub-y, local-n, addr=0x004074d8, size=6648
sym `dct_luma_sp': text seg, init-y, pub-y, local-n, addr=0x00408ed0, size=6984
sym `dct_chroma_sp': text seg, init-y, pub-y, local-n, addr=0x0040aa18, size=11760
sym `copyblock_sp': text seg, init-y, pub-y, local-n, addr=0x0040d808, size=3176
sym `cabac_new_slice': text seg, init-y, pub-y, local-n, addr=0x0040e470, size=16
sym `CheckAvailabilityOfNeighborsCABAC': text seg, init-y, pub-y, local-n, addr=0x0040e480, size=472
sym `create_contexts_MotionInfo': text seg, init-y, pub-y, local-n, addr=0x0040e658, size=128
sym `create_contexts_TextureInfo': text seg, init-y, pub-y, local-n, addr=0x0040e6d8, size=128
sym `delete_contexts_MotionInfo': text seg, init-y, pub-y, local-n, addr=0x0040e758, size=56
sym `delete_contexts_TextureInfo': text seg, init-y, pub-y, local-n, addr=0x0040e790, size=56
sym `writeSyntaxElement_CABAC': text seg, init-y, pub-y, local-n, addr=0x0040e7c8, size=256
sym `writeFieldModeInfo_CABAC': text seg, init-y, pub-y, local-n, addr=0x0040e8c8, size=488
sym `writeMB_skip_flagInfo_CABAC': text seg, init-y, pub-y, local-n, addr=0x0040eab0, size=728
sym `writeMB_typeInfo_CABAC': text seg, init-y, pub-y, local-n, addr=0x0040ed88, size=3816
sym `writeB8_typeInfo_CABAC': text seg, init-y, pub-y, local-n, addr=0x0040fc70, size=1352
sym `writeIntraPredMode_CABAC': text seg, init-y, pub-y, local-n, addr=0x004101b8, size=360
sym `writeRefFrame_CABAC': text seg, init-y, pub-y, local-n, addr=0x00410320, size=1960
sym `writeDquant_CABAC': text seg, init-y, pub-y, local-n, addr=0x00410ac8, size=440

sym `writeMVD_CABAC': text seg, init-y, pub-y, local-n, addr=0x00410c80, size=1720
sym `writeCIPredMode_CABAC': text seg, init-y, pub-y, local-n, addr=0x00411338, size=464
sym `writeCBP_BIT_CABAC': text seg, init-y, pub-y, local-n, addr=0x00411508, size=648
sym `writeCBP_CABAC': text seg, init-y, pub-y, local-n, addr=0x00411790, size=984
sym `write_and_store_CBP_block_bit': text seg, init-y, pub-y, local-n, addr=0x00411b68, size=2504
sym `write_significance_map': text seg, init-y, pub-y, local-n, addr=0x00412530, size=1224
sym `write_significant_coefficients': text seg, init-y, pub-y, local-n, addr=0x004129f8, size=800
sym `writeRunLevel_CABAC': text seg, init-y, pub-y, local-n, addr=0x00412d18, size=704
sym `unary_bin_encode': text seg, init-y, pub-y, local-n, addr=0x00412fd8, size=304
sym `unary_bin_max_encode': text seg, init-y, pub-y, local-n, addr=0x00413108, size=360
sym `exp_golomb_encode_eq_prob': text seg, init-y, pub-y, local-n, addr=0x00413270, size=328
sym `unary_exp_golomb_level_encode': text seg, init-y, pub-y, local-n, addr=0x004133b8, size=416
sym `unary_exp_golomb_mv_encode': text seg, init-y, pub-y, local-n, addr=0x00413558, size=536
sym `JMHelpExit': text seg, init-y, pub-y, local-n, addr=0x00413770, size=88
sym `Configure': text seg, init-y, pub-y, local-n, addr=0x004137c8, size=4376
sym `CeilLog2': text seg, init-y, pub-y, local-n, addr=0x004148e0, size=7576
sym `PatchInputNoFrames': text seg, init-y, pub-y, local-n, addr=0x00416678, size=1720
sym `create_context_memory': text seg, init-y, pub-y, local-n, addr=0x00416d30, size=1424
sym `free_context_memory': text seg, init-y, pub-y, local-n, addr=0x004172c0, size=384
sym `SetCtxModelNumber': text seg, init-y, pub-y, local-n, addr=0x00417440, size=512
sym `init_contexts': text seg, init-y, pub-y, local-n, addr=0x00417640, size=5560
sym `XRate': text seg, init-y, pub-y, local-n, addr=0x00418bf8, size=640
sym `GetCtxModelNumber': text seg, init-y, pub-y, local-n, addr=0x00418e78, size=5288
sym `store_contexts': text seg, init-y, pub-y, local-n, addr=0x0041a320, size=368
sym `update_field_frame_contexts': text seg, init-y, pub-y, local-n, addr=0x0041a490, size=640
sym `decode_one_b8block': text seg, init-y, pub-y, local-n, addr=0x0041a710, size=3312
sym `decode_one_mb': text seg, init-y, pub-y, local-n, addr=0x0041b400, size=8
sym `Get_Reference_Block': text seg, init-y, pub-y, local-n, addr=0x0041b408, size=376
sym `Get_Reference_Pixel': text seg, init-y, pub-y, local-n, addr=0x0041b580, size=5008
sym `UpdateDecoders': text seg, init-y, pub-y, local-n, addr=0x0041c910, size=312
sym `DecOneForthPix': text seg, init-y, pub-y, local-n, addr=0x0041ca48, size=376
sym `compute_residue_b8block': text seg, init-y, pub-y, local-n, addr=0x0041cbc0, size=752
sym `compute_residue_mb': text seg, init-y, pub-y, local-n, addr=0x0041ceb0, size=160
sym `Build_Status_Map': text seg, init-y, pub-y, local-n, addr=0x0041cf50, size=976
sym `Error_Concealment': text seg, init-y, pub-y, local-n, addr=0x0041d320, size=464
sym `Conceal_Error': text seg, init-y, pub-y, local-n, addr=0x0041d4f0, size=4160
sym `DefineThreshold': text seg, init-y, pub-y, local-n, addr=0x0041e530, size=352
sym `DefineThresholdMB': text seg, init-y, pub-y, local-n, addr=0x0041e690, size=752
sym `get_mem_mincost': text seg, init-y, pub-y, local-n, addr=0x0041e980, size=1288
sym `get_mem_bwmincost': text seg, init-y, pub-y, local-n, addr=0x0041ee88, size=1288
sym `get_mem_FME': text seg, init-y, pub-y, local-n, addr=0x0041f390, size=208
sym `free_mem_mincost': text seg, init-y, pub-y, local-n, addr=0x0041f460, size=736

sym `free_mem_bwmincost': text seg, init-y, pub-y, local-n, addr=0x0041f740, size=736
sym `free_mem_FME': text seg, init-y, pub-y, local-n, addr=0x0041fa20, size=104
sym `PartCalMad': text seg, init-y, pub-y, local-n, addr=0x0041fa88, size=896
sym `FastIntegerPelBlockMotionSearch': text seg, init-y, pub-y, local-n, addr=0x0041fe08, size=14056
sym `AddUpSADQuarter': text seg, init-y, pub-y, local-n, addr=0x004234f0, size=2576
sym `FastSubPelBlockMotionSearch': text seg, init-y, pub-y, local-n, addr=0x00423f00, size=3104
sym `decide_intrabk_SAD': text seg, init-y, pub-y, local-n, addr=0x00424b20, size=336
sym `skip_intrabk_SAD': text seg, init-y, pub-y, local-n, addr=0x00424c70, size=464
sym `error': text seg, init-y, pub-y, local-n, addr=0x00424e40, size=120
sym `start_sequence': text seg, init-y, pub-y, local-n, addr=0x00424eb8, size=488
sym `terminate_sequence': text seg, init-y, pub-y, local-n, addr=0x004250a0, size=1760
sym `FmoInit': text seg, init-y, pub-y, local-n, addr=0x00425780, size=200
sym `FmoUninit': text seg, init-y, pub-y, local-n, addr=0x00425848, size=2840
sym `FmoStartPicture': text seg, init-y, pub-y, local-n, addr=0x00426360, size=240
sym `FmoEndPicture': text seg, init-y, pub-y, local-n, addr=0x00426450, size=16
sym `FmoMB2SliceGroup': text seg, init-y, pub-y, local-n, addr=0x00426460, size=288
sym `FmoGetNextMBNr': text seg, init-y, pub-y, local-n, addr=0x00426580, size=272
sym `FmoGetPreviousMBNr': text seg, init-y, pub-y, local-n, addr=0x00426690, size=184
sym `FmoGetFirstMBOfSliceGroup': text seg, init-y, pub-y, local-n, addr=0x00426748, size=288
sym `FmoGetLastCodedMBOfSliceGroup': text seg, init-y, pub-y, local-n, addr=0x00426868, size=264
sym `FmoSetLastMacroblockInSlice': text seg, init-y, pub-y, local-n, addr=0x00426970, size=208
sym `FmoGetFirstMacroblockInSlice': text seg, init-y, pub-y, local-n, addr=0x00426a40, size=40
sym `FmoSliceGroupCompletelyCoded': text seg, init-y, pub-y, local-n, addr=0x00426a68, size=88
sym `SliceHeader': text seg, init-y, pub-y, local-n, addr=0x00426ac0, size=7952
sym `get_picture_type': text seg, init-y, pub-y, local-n, addr=0x004289d0, size=248
sym `Partition_BC_Header': text seg, init-y, pub-y, local-n, addr=0x00428ac8, size=600
sym `MbAffPostProc': text seg, init-y, pub-y, local-n, addr=0x00428d20, size=1320
sym `code_a_picture': text seg, init-y, pub-y, local-n, addr=0x00429248, size=1808
sym `encode_one_frame': text seg, init-y, pub-y, local-n, addr=0x00429958, size=4160
sym `frame_picture': text seg, init-y, pub-y, local-n, addr=0x0042a998, size=624
sym `field_picture': text seg, init-y, pub-y, local-n, addr=0x0042ac08, size=8344
sym `UnifiedOneForthPix': text seg, init-y, pub-y, local-n, addr=0x0042cca0, size=11816
sym `dummy_slice_too_big': text seg, init-y, pub-y, local-n, addr=0x0042fac8, size=16
sym `copy_rdopt_data': text seg, init-y, pub-y, local-n, addr=0x0042fad8, size=16968
sym `RandomIntraInit': text seg, init-y, pub-y, local-n, addr=0x00433d20, size=624
sym `RandomIntra': text seg, init-y, pub-y, local-n, addr=0x00433f90, size=136
sym `RandomIntraNewPicture': text seg, init-y, pub-y, local-n, addr=0x00434018, size=256
sym `RandomIntraUninit': text seg, init-y, pub-y, local-n, addr=0x00434118, size=72
sym `get_LeakyBucketRate': text seg, init-y, pub-y, local-n, addr=0x00434160, size=416
sym `PutBigDoubleWord': text seg, init-y, pub-y, local-n, addr=0x00434300, size=192
sym `write_buffer': text seg, init-y, pub-y, local-n, addr=0x004343c0, size=624

sym `Sort': text seg, init-y, pub-y, local-n, addr=0x00434630, size=224
sym `calc_buffer': text seg, init-y, pub-y, local-n, addr=0x00434710, size=1968
sym `main': text seg, init-y, pub-y, local-n, addr=0x00434ec0, size=4792
sym `report_stats_on_error': text seg, init-y, pub-y, local-n, addr=0x00436178, size=344
sym `init_poc': text seg, init-y, pub-y, local-n, addr=0x004362d0, size=440
sym `CAVLC_init': text seg, init-y, pub-y, local-n, addr=0x00436488, size=272
sym `init_img': text seg, init-y, pub-y, local-n, addr=0x00436598, size=2512
sym `free_img': text seg, init-y, pub-y, local-n, addr=0x00436f68, size=224
sym `malloc_picture': text seg, init-y, pub-y, local-n, addr=0x00437048, size=128
sym `free_picture': text seg, init-y, pub-y, local-n, addr=0x004370c8, size=96
sym `report': text seg, init-y, pub-y, local-n, addr=0x00437128, size=15640
sym `information_init': text seg, init-y, pub-y, local-n, addr=0x0043ae40, size=328
sym `init_global_buffers': text seg, init-y, pub-y, local-n, addr=0x0043af88, size=2224
sym `free_global_buffers': text seg, init-y, pub-y, local-n, addr=0x0043b838, size=1704
sym `get_mem_mv': text seg, init-y, pub-y, local-n, addr=0x0043bee0, size=1264
sym `free_mem_mv': text seg, init-y, pub-y, local-n, addr=0x0043c3d0, size=792
sym `get_mem_ACcoeff': text seg, init-y, pub-y, local-n, addr=0x0043c6e8, size=688
sym `get_mem_DCcoeff': text seg, init-y, pub-y, local-n, addr=0x0043c998, size=472
sym `free_mem_ACcoeff': text seg, init-y, pub-y, local-n, addr=0x0043cb70, size=408
sym `free_mem_DCcoeff': text seg, init-y, pub-y, local-n, addr=0x0043cd08, size=256
sym `combine_field': text seg, init-y, pub-y, local-n, addr=0x0043ce08, size=888
sym `decide_fld_frame': text seg, init-y, pub-y, local-n, addr=0x0043d180, size=208
sym `process_2nd_IGOP': text seg, init-y, pub-y, local-n, addr=0x0043d250, size=248
sym `SetImgType': text seg, init-y, pub-y, local-n, addr=0x0043d348, size=344
sym `DeblockFrame': text seg, init-y, pub-y, local-n, addr=0x0043d4a0, size=264
sym `DeblockMb': text seg, init-y, pub-y, local-n, addr=0x0043d5a8, size=1968
sym `GetStrength': text seg, init-y, pub-y, local-n, addr=0x0043dd58, size=3896
sym `EdgeLoop': text seg, init-y, pub-y, local-n, addr=0x0043ec90, size=3600
sym `set_MB_parameters': text seg, init-y, pub-y, local-n, addr=0x0043faa0, size=664
sym `clip1a': text seg, init-y, pub-y, local-n, addr=0x0043fd38, size=64
sym `proceed2nextMacroblock': text seg, init-y, pub-y, local-n, addr=0x0043fd78, size=656
sym `start_macroblock': text seg, init-y, pub-y, local-n, addr=0x00440008, size=4600
sym `terminate_macroblock': text seg, init-y, pub-y, local-n, addr=0x00441200, size=3048
sym `slice_too_big': text seg, init-y, pub-y, local-n, addr=0x00441de8, size=512
sym `OneComponentLumaPrediction4x4': text seg, init-y, pub-y, local-n, addr=0x00441fe8, size=1520
sym `copyblock4x4': text seg, init-y, pub-y, local-n, addr=0x004425d8, size=376
sym `LumaPrediction4x4': text seg, init-y, pub-y, local-n, addr=0x00442750, size=3032
sym `LumaResidualCoding8x8': text seg, init-y, pub-y, local-n, addr=0x00443328, size=1944
sym `SetModesAndRefframe': text seg, init-y, pub-y, local-n, addr=0x00443ac0, size=896
sym `LumaResidualCoding': text seg, init-y, pub-y, local-n, addr=0x00443e40, size=904
sym `OneComponentChromaPrediction4x4': text seg, init-y, pub-y, local-n, addr=0x004441c8, size=1648
sym `IntraChromaPrediction4x4': text seg, init-y, pub-y, local-n, addr=0x00444838, size=360

sym `ChromaPrediction4x4': text seg, init-y, pub-y, local-n, addr=0x004449a0, size=2688
sym `ChromaResidualCoding': text seg, init-y, pub-y, local-n, addr=0x00445420, size=1928
sym `IntraChromaPrediction8x8': text seg, init-y, pub-y, local-n, addr=0x00445ba8, size=5456
sym `ZeroRef': text seg, init-y, pub-y, local-n, addr=0x004470f8, size=216
sym `MBType2Value': text seg, init-y, pub-y, local-n, addr=0x004471d0, size=720
sym `writeIntra4x4Modes': text seg, init-y, pub-y, local-n, addr=0x004474a0, size=1168
sym `B8Mode2Value': text seg, init-y, pub-y, local-n, addr=0x00447930, size=136
sym `writeMBHeader': text seg, init-y, pub-y, local-n, addr=0x004479b8, size=3984
sym `write_terminating_bit': text seg, init-y, pub-y, local-n, addr=0x00448948, size=216
sym `writeChromaIntraPredMode': text seg, init-y, pub-y, local-n, addr=0x00448a20, size=560
sym `set_last_dquant': text seg, init-y, pub-y, local-n, addr=0x00448c50, size=192
sym `write_one_macroblock': text seg, init-y, pub-y, local-n, addr=0x00448d10, size=1272
sym `BType2CtxRef': text seg, init-y, pub-y, local-n, addr=0x00449208, size=40
sym `writeReferenceFrame': text seg, init-y, pub-y, local-n, addr=0x00449230, size=1048
sym `writeMotionVector8x8': text seg, init-y, pub-y, local-n, addr=0x00449648, size=1720
sym `writeMotionInfo2NAL': text seg, init-y, pub-y, local-n, addr=0x00449d00, size=1960
sym `writeChromaCoeff': text seg, init-y, pub-y, local-n, addr=0x0044a4a8, size=2608
sym `writeLumaCoeff4x4_CABAC': text seg, init-y, pub-y, local-n, addr=0x0044aed8, size=1168
sym `writeLumaCoeff8x8': text seg, init-y, pub-y, local-n, addr=0x0044b368, size=280
sym `writeCBPandLumaCoeff': text seg, init-y, pub-y, local-n, addr=0x0044b480, size=3544
sym `predict_nnz': text seg, init-y, pub-y, local-n, addr=0x0044c258, size=696
sym `predict_nnz_chroma': text seg, init-y, pub-y, local-n, addr=0x0044c510, size=760
sym `writeCoeff4x4_CAVLC': text seg, init-y, pub-y, local-n, addr=0x0044c808, size=4544
sym `find_sad_16x16': text seg, init-y, pub-y, local-n, addr=0x0044d9c8, size=3208
sym `mb_is_available': text seg, init-y, pub-y, local-n, addr=0x0044e650, size=272
sym `CheckAvailabilityOfNeighbors': text seg, init-y, pub-y, local-n, addr=0x0044e760, size=1560
sym `get_mb_block_pos': text seg, init-y, pub-y, local-n, addr=0x0044ed78, size=336
sym `get_mb_pos': text seg, init-y, pub-y, local-n, addr=0x0044eec8, size=144
sym `getNonAffNeighbour': text seg, init-y, pub-y, local-n, addr=0x0044ef58, size=960
sym `getAffNeighbour': text seg, init-y, pub-y, local-n, addr=0x0044f318, size=3016
sym `getNeighbour': text seg, init-y, pub-y, local-n, addr=0x0044fee0, size=344
sym `getLuma4x4Neighbour': text seg, init-y, pub-y, local-n, addr=0x00450038, size=304
sym `getChroma4x4Neighbour': text seg, init-y, pub-y, local-n, addr=0x00450168, size=312
sym `dump_dpb': text seg, init-y, pub-y, local-n, addr=0x004502a0, size=24
sym `getDpbSize': text seg, init-y, pub-y, local-n, addr=0x004502b8, size=640
sym `init_dpb': text seg, init-y, pub-y, local-n, addr=0x00450538, size=1056
sym `free_dpb': text seg, init-y, pub-y, local-n, addr=0x00450958, size=448
sym `alloc_frame_store': text seg, init-y, pub-y, local-n, addr=0x00450b18, size=192
sym `alloc_storable_picture': text seg, init-y, pub-y, local-n, addr=0x00450bd8, size=920
sym `free_frame_store': text seg, init-y, pub-y, local-n, addr=0x00450f70, size=184
sym `free_storable_picture': text seg, init-y, pub-y, local-n, addr=0x00451028, size=1888
sym `is_short_ref': text seg, init-y, pub-y, local-n, addr=0x00451788, size=64
sym `is_long_ref': text seg, init-y, pub-y, local-n, addr=0x004517c8, size=1312

sym `init_lists': text seg, init-y, pub-y, local-n, addr=0x00451ce8, size=7136
sym `init_mbaff_lists': text seg, init-y, pub-y, local-n, addr=0x004538c8, size=2680
sym `reorder_ref_pic_list': text seg, init-y, pub-y, local-n, addr=0x00454340, size=792
sym `update_ref_list': text seg, init-y, pub-y, local-n, addr=0x00454658, size=440
sym `update_ltref_list': text seg, init-y, pub-y, local-n, addr=0x00454810, size=6504
sym `mm_update_max_long_term_frame_idx': text seg, init-y, pub-y, local-n, addr=0x00456178, size=1776
sym `store_picture_in_dpb': text seg, init-y, pub-y, local-n, addr=0x00456868, size=1528
sym `replace_top_pic_with_frame': text seg, init-y, pub-y, local-n, addr=0x00456e60, size=3992
sym `flush_dpb': text seg, init-y, pub-y, local-n, addr=0x00457df8, size=272
sym `gen_field_ref_ids': text seg, init-y, pub-y, local-n, addr=0x00457f08, size=552
sym `dpb_split_field': text seg, init-y, pub-y, local-n, addr=0x00458130, size=10608
sym `dpb_combine_field': text seg, init-y, pub-y, local-n, addr=0x0045aaa0, size=5712
sym `alloc_ref_pic_list_reordering_buffer': text seg, init-y, pub-y, local-n, addr=0x0045c0f0, size=672
sym `free_ref_pic_list_reordering_buffer': text seg, init-y, pub-y, local-n, addr=0x0045c390, size=256
sym `fill_frame_num_gap': text seg, init-y, pub-y, local-n, addr=0x0045c490, size=592
sym `alloc_colocated': text seg, init-y, pub-y, local-n, addr=0x0045c6e0, size=864
sym `free_collocated': text seg, init-y, pub-y, local-n, addr=0x0045ca40, size=544
sym `compute_collocated': text seg, init-y, pub-y, local-n, addr=0x0045cc60, size=17456
sym `get_mem2D': text seg, init-y, pub-y, local-n, addr=0x00461090, size=368
sym `get_mem2Dint': text seg, init-y, pub-y, local-n, addr=0x00461200, size=384
sym `get_mem2Dint64': text seg, init-y, pub-y, local-n, addr=0x00461380, size=384
sym `get_mem3D': text seg, init-y, pub-y, local-n, addr=0x00461500, size=336
sym `get_mem3Dint': text seg, init-y, pub-y, local-n, addr=0x00461650, size=344
sym `get_mem3Dint64': text seg, init-y, pub-y, local-n, addr=0x004617a8, size=344
sym `get_mem4Dint': text seg, init-y, pub-y, local-n, addr=0x00461900, size=392
sym `free_mem2D': text seg, init-y, pub-y, local-n, addr=0x00461a88, size=192
sym `free_mem2Dint': text seg, init-y, pub-y, local-n, addr=0x00461b48, size=192
sym `free_mem2Dint64': text seg, init-y, pub-y, local-n, addr=0x00461c08, size=192
sym `free_mem3D': text seg, init-y, pub-y, local-n, addr=0x00461cc8, size=256
sym `free_mem3Dint': text seg, init-y, pub-y, local-n, addr=0x00461dc8, size=256
sym `free_mem3Dint64': text seg, init-y, pub-y, local-n, addr=0x00461ec8, size=256
sym `free_mem4Dint': text seg, init-y, pub-y, local-n, addr=0x00461fc8, size=288
sym `no_mem_exit': text seg, init-y, pub-y, local-n, addr=0x004620e8, size=152
sym `InitializeFastFullIntegerSearch': text seg, init-y, pub-y, local-n, addr=0x00462180, size=2272
sym `ClearFastFullIntegerSearch': text seg, init-y, pub-y, local-n, addr=0x00462a60, size=968
sym `ResetFastFullIntegerSearch': text seg, init-y, pub-y, local-n, addr=0x00462e28, size=160
sym `SetupLargerBlocks': text seg, init-y, pub-y, local-n, addr=0x00462ec8, size=4424
sym `SetupFastFullPelSearch': text seg, init-y, pub-y, local-n, addr=0x00464010, size=4128
sym `SetMotionVectorPredictor': text seg, init-y, pub-y, local-n, addr=0x00465030, size=6392
sym `Init_Motion_Search_Module': text seg, init-y, pub-y, local-n, addr=0x00466928, size=2064
sym `Clear_Motion_Search_Module': text seg, init-y, pub-y, local-n, addr=0x00467138, size=248

sym `FullPelBlockMotionSearch': text seg, init-y, pub-y, local-n, addr=0x00467230, size=2176
sym `FastFullPelBlockMotionSearch': text seg, init-y, pub-y, local-n, addr=0x00467ab0, size=1320
sym `SATD': text seg, init-y, pub-y, local-n, addr=0x00467fd8, size=2280
sym `SubPelBlockMotionSearch': text seg, init-y, pub-y, local-n, addr=0x004688c0, size=6896
sym `BlockMotionSearch': text seg, init-y, pub-y, local-n, addr=0x0046a3b0, size=13056
sym `BIDPartitionCost': text seg, init-y, pub-y, local-n, addr=0x0046d6b0, size=2136
sym `GetSkipCostMB': text seg, init-y, pub-y, local-n, addr=0x0046df08, size=632
sym `FindSkipModeMotionVector': text seg, init-y, pub-y, local-n, addr=0x0046e180, size=2064
sym `Get_Direct_Cost8x8': text seg, init-y, pub-y, local-n, addr=0x0046e990, size=968
sym `Get_Direct_CostMB': text seg, init-y, pub-y, local-n, addr=0x0046ed58, size=208
sym `PartitionMotionSearch': text seg, init-y, pub-y, local-n, addr=0x0046ee28, size=2672
sym `Get_Direct_Motion_Vectors': text seg, init-y, pub-y, local-n, addr=0x0046f898, size=10240
sym `sign': text seg, init-y, pub-y, local-n, addr=0x00472098, size=56
sym `SODBtoRBSP': text seg, init-y, pub-y, local-n, addr=0x004720d0, size=168
sym `RBSPtoEBSP': text seg, init-y, pub-y, local-n, addr=0x00472178, size=544
sym `AllocNalPayloadBuffer': text seg, init-y, pub-y, local-n, addr=0x00472398, size=216
sym `FreeNalPayloadBuffer': text seg, init-y, pub-y, local-n, addr=0x00472470, size=80
sym `RBSPtoNALU': text seg, init-y, pub-y, local-n, addr=0x004724c0, size=752
sym `AllocNALU': text seg, init-y, pub-y, local-n, addr=0x004727b0, size=224
sym `FreeNALU': text seg, init-y, pub-y, local-n, addr=0x00472890, size=128
sym `write_picture': text seg, init-y, pub-y, local-n, addr=0x00472910, size=704
sym `init_out_buffer': text seg, init-y, pub-y, local-n, addr=0x00472bd0, size=56
sym `uninit_out_buffer': text seg, init-y, pub-y, local-n, addr=0x00472c08, size=64
sym `clear_picture': text seg, init-y, pub-y, local-n, addr=0x00472c48, size=432
sym `write_unpaired_field': text seg, init-y, pub-y, local-n, addr=0x00472df8, size=496
sym `flush_direct_output': text seg, init-y, pub-y, local-n, addr=0x00472fe8, size=176
sym `write_stored_frame': text seg, init-y, pub-y, local-n, addr=0x00473098, size=200
sym `direct_output': text seg, init-y, pub-y, local-n, addr=0x00473160, size=576
sym `GenerateParameterSets': text seg, init-y, pub-y, local-n, addr=0x004733a0, size=144
sym `FreeParameterSets': text seg, init-y, pub-y, local-n, addr=0x00473430, size=72
sym `GenerateSeq_parameter_set_NALU': text seg, init-y, pub-y, local-n, addr=0x00473478, size=240
sym `GeneratePic_parameter_set_NALU': text seg, init-y, pub-y, local-n, addr=0x00473568, size=240
sym `FillParameterSetStructures': text seg, init-y, pub-y, local-n, addr=0x00473658, size=2136
sym `GenerateSeq_parameter_set_rbsp': text seg, init-y, pub-y, local-n, addr=0x00473eb0, size=1736
sym `GeneratePic_parameter_set_rbsp': text seg, init-y, pub-y, local-n, addr=0x00474578, size=2072
sym `AllocPPS': text seg, init-y, pub-y, local-n, addr=0x00474d90, size=200
sym `AllocSPS': text seg, init-y, pub-y, local-n, addr=0x00474e58, size=128
sym `FreePPS': text seg, init-y, pub-y, local-n, addr=0x00474ed8, size=168
sym `FreeSPS': text seg, init-y, pub-y, local-n, addr=0x00474f80, size=144

sym `rc_init_seq': text seg, init-y, pub-y, local-n, addr=0x00475010, size=1512
sym `rc_init_GOP': text seg, init-y, pub-y, local-n, addr=0x004755f8, size=1176
sym `rc_init_pict': text seg, init-y, pub-y, local-n, addr=0x00475a90, size=3952
sym `calc_MAD': text seg, init-y, pub-y, local-n, addr=0x00476a00, size=216
sym `rc_update_pict': text seg, init-y, pub-y, local-n, addr=0x00476ad8, size=224
sym `rc_update_pict_frame': text seg, init-y, pub-y, local-n, addr=0x00476bb8, size=816
sym `setbitscount': text seg, init-y, pub-y, local-n, addr=0x00476ee8, size=16
sym `updateQuantizationParameter': text seg, init-y, pub-y, local-n, addr=0x00476ef8, size=9352
sym `updateRCModel': text seg, init-y, pub-y, local-n, addr=0x00479380, size=2616
sym `RCModelEstimator': text seg, init-y, pub-y, local-n, addr=0x00479db8, size=1232
sym `ComputeFrameMAD': text seg, init-y, pub-y, local-n, addr=0x0047a288, size=296
sym `updateMADModel': text seg, init-y, pub-y, local-n, addr=0x0047a3b0, size=1640
sym `MADModelEstimator': text seg, init-y, pub-y, local-n, addr=0x0047aa18, size=1208
sym `QP2Qstep': text seg, init-y, pub-y, local-n, addr=0x0047aed0, size=248
sym `Qstep2QP': text seg, init-y, pub-y, local-n, addr=0x0047afc8, size=584
sym `clear_rdopt': text seg, init-y, pub-y, local-n, addr=0x0047b210, size=216
sym `init_rdopt': text seg, init-y, pub-y, local-n, addr=0x0047b2e8, size=248
sym `UpdatePixelMap': text seg, init-y, pub-y, local-n, addr=0x0047b3e0, size=864
sym `CheckReliabilityOfRef': text seg, init-y, pub-y, local-n, addr=0x0047b740, size=3520
sym `RDCost_for_4x4IntraBlocks': text seg, init-y, pub-y, local-n, addr=0x0047c500, size=1472
sym `Mode_Decision_for_4x4IntraBlocks': text seg, init-y, pub-y, local-n, addr=0x0047cac0, size=4224
sym `Mode_Decision_for_8x8IntraBlocks': text seg, init-y, pub-y, local-n, addr=0x0047db40, size=360
sym `Mode_Decision_for_Intra4x4Macroblock': text seg, init-y, pub-y, local-n, addr=0x0047dca8, size=304
sym `RDCost_for_8x8blocks': text seg, init-y, pub-y, local-n, addr=0x0047ddd8, size=3728
sym `I16Offset': text seg, init-y, pub-y, local-n, addr=0x0047ec68, size=72
sym `SetModesAndRefframeForBlocks': text seg, init-y, pub-y, local-n, addr=0x0047ecb0, size=4072
sym `Intra16x16_Mode_Decision': text seg, init-y, pub-y, local-n, addr=0x0047fc98, size=136
sym `SetCoeffAndReconstruction8x8': text seg, init-y, pub-y, local-n, addr=0x0047fd20, size=1200
sym `SetMotionVectorsMB': text seg, init-y, pub-y, local-n, addr=0x004801d0, size=3288
sym `RDCost_for_macroblocks': text seg, init-y, pub-y, local-n, addr=0x00480ea8, size=3600
sym `store_macroblock_parameters': text seg, init-y, pub-y, local-n, addr=0x00481cb8, size=1808
sym `set_stored_macroblock_parameters': text seg, init-y, pub-y, local-n, addr=0x004823c8, size=7392
sym `SetRefAndMotionVectors': text seg, init-y, pub-y, local-n, addr=0x004840a8, size=5160
sym `field_flag_inference': text seg, init-y, pub-y, local-n, addr=0x004854d0, size=208
sym `encode_one_macroblock': text seg, init-y, pub-y, local-n, addr=0x004855a0, size=29712
sym `set_mbaff_parameters': text seg, init-y, pub-y, local-n, addr=0x0048c9b0, size=2240
sym `delete_coding_state': text seg, init-y, pub-y, local-n, addr=0x0048d270, size=168
sym `create_coding_state': text seg, init-y, pub-y, local-n, addr=0x0048d318, size=432

sym `store_coding_state': text seg, init-y, pub-y, local-n, addr=0x0048d4c8, size=1560
sym `reset_coding_state': text seg, init-y, pub-y, local-n, addr=0x0048dae0, size=1552
sym `PutPel_14': text seg, init-y, pub-y, local-n, addr=0x0048e0f0, size=48
sym `PutPel_11': text seg, init-y, pub-y, local-n, addr=0x0048e120, size=56
sym `FastLine16Y_11': text seg, init-y, pub-y, local-n, addr=0x0048e158, size=48
sym `UMVLine16Y_11': text seg, init-y, pub-y, local-n, addr=0x0048e188, size=656
sym `FastLineX': text seg, init-y, pub-y, local-n, addr=0x0048e418, size=48
sym `UMVLineX': text seg, init-y, pub-y, local-n, addr=0x0048e448, size=664
sym `UMVPelY_14': text seg, init-y, pub-y, local-n, addr=0x0048e6e0, size=536
sym `FastPelY_14': text seg, init-y, pub-y, local-n, addr=0x0048e8f8, size=56
sym `ComposeRTPPacket': text seg, init-y, pub-y, local-n, addr=0x0048e930, size=1272
sym `WriteRTPPacket': text seg, init-y, pub-y, local-n, addr=0x0048ee28, size=488
sym `WriteRTPNALU': text seg, init-y, pub-y, local-n, addr=0x0048f010, size=984
sym `RTPUpdateTimestamp': text seg, init-y, pub-y, local-n, addr=0x0048f3e8, size=160
sym `OpenRTPFile': text seg, init-y, pub-y, local-n, addr=0x0048f488, size=144
sym `CloseRTPFile': text seg, init-y, pub-y, local-n, addr=0x0048f518, size=56
sym `InitSEIMessages': text seg, init-y, pub-y, local-n, addr=0x0048f550, size=400
sym `CloseSEIMessages': text seg, init-y, pub-y, local-n, addr=0x0048f6e0, size=248
sym `HaveAggregationSEI': text seg, init-y, pub-y, local-n, addr=0x0048f7d8, size=248
sym `write_sei_message': text seg, init-y, pub-y, local-n, addr=0x0048f8d0, size=664
sym `finalize_sei_message': text seg, init-y, pub-y, local-n, addr=0x0048fb68, size=176
sym `clear_sei_message': text seg, init-y, pub-y, local-n, addr=0x0048fc18, size=160
sym `AppendTmpbits2Buf': text seg, init-y, pub-y, local-n, addr=0x0048fcb8, size=632
sym `InitSparePicture': text seg, init-y, pub-y, local-n, addr=0x0048ff30, size=488
sym `CloseSparePicture': text seg, init-y, pub-y, local-n, addr=0x00490118, size=200
sym `CalculateSparePicture': text seg, init-y, pub-y, local-n, addr=0x004901e0, size=8
sym `ComposeSparePictureMessage': text seg, init-y, pub-y, local-n, addr=0x004901e8, size=240
sym `CompressSpareMBMap': text seg, init-y, pub-y, local-n, addr=0x004902d8, size=1976
sym `FinalizeSpareMBMap': text seg, init-y, pub-y, local-n, addr=0x00490a90, size=872
sym `InitSubseqInfo': text seg, init-y, pub-y, local-n, addr=0x00490df8, size=504
sym `UpdateSubseqInfo': text seg, init-y, pub-y, local-n, addr=0x00490ff0, size=712
sym `FinalizeSubseqInfo': text seg, init-y, pub-y, local-n, addr=0x004912b8, size=656
sym `ClearSubseqInfoPayload': text seg, init-y, pub-y, local-n, addr=0x00491548, size=264
sym `CloseSubseqInfo': text seg, init-y, pub-y, local-n, addr=0x00491650, size=208
sym `InitSubseqLayerInfo': text seg, init-y, pub-y, local-n, addr=0x00491720, size=160
sym `CloseSubseqLayerInfo': text seg, init-y, pub-y, local-n, addr=0x004917c0, size=8
sym `FinalizeSubseqLayerInfo': text seg, init-y, pub-y, local-n, addr=0x004917c8, size=264
sym `InitSubseqChar': text seg, init-y, pub-y, local-n, addr=0x004918d0, size=456
sym `ClearSubseqCharPayload': text seg, init-y, pub-y, local-n, addr=0x00491a98, size=192
sym `UpdateSubseqChar': text seg, init-y, pub-y, local-n, addr=0x00491b58, size=328
sym `FinalizeSubseqChar': text seg, init-y, pub-y, local-n, addr=0x00491ca0, size=1040
sym `CloseSubseqChar': text seg, init-y, pub-y, local-n, addr=0x004920b0, size=144
sym `InitSceneInformation': text seg, init-y, pub-y, local-n, addr=0x00492140, size=312
sym `CloseSceneInformation': text seg, init-y, pub-y, local-n, addr=0x00492278, size=144

sym `FinalizeSceneInformation': text seg, init-y, pub-y, local-n, addr=0x00492308, size=528
sym `UpdateSceneInformation': text seg, init-y, pub-y, local-n, addr=0x00492518, size=424
sym `InitPanScanRectInfo': text seg, init-y, pub-y, local-n, addr=0x004926c0, size=352
sym `ClearPanScanRectInfoPayload': text seg, init-y, pub-y, local-n, addr=0x00492820, size=200
sym `UpdatePanScanRectInfo': text seg, init-y, pub-y, local-n, addr=0x004928e8, size=144
sym `FinalizePanScanRectInfo': text seg, init-y, pub-y, local-n, addr=0x00492978, size=552
sym `ClosePanScanRectInfo': text seg, init-y, pub-y, local-n, addr=0x00492ba0, size=144
sym `InitUser_data_unregistered': text seg, init-y, pub-y, local-n, addr=0x00492c30, size=392
sym `ClearUser_data_unregistered': text seg, init-y, pub-y, local-n, addr=0x00492db8, size=256
sym `UpdateUser_data_unregistered': text seg, init-y, pub-y, local-n, addr=0x00492eb8, size=192
sym `FinalizeUser_data_unregistered': text seg, init-y, pub-y, local-n, addr=0x00492f78, size=496
sym `CloseUser_data_unregistered': text seg, init-y, pub-y, local-n, addr=0x00493168, size=176
sym `InitUser_data_registered_itu_t_t35': text seg, init-y, pub-y, local-n, addr=0x00493218, size=392
sym `ClearUser_data_registered_itu_t_t35': text seg, init-y, pub-y, local-n, addr=0x004933a0, size=288
sym `UpdateUser_data_registered_itu_t_t35': text seg, init-y, pub-y, local-n, addr=0x004934c0, size=216
sym `FinalizeUser_data_registered_itu_t_t35': text seg, init-y, pub-y, local-n, addr=0x00493598, size=648
sym `CloseUser_data_registered_itu_t_t35': text seg, init-y, pub-y, local-n, addr=0x00493820, size=176
sym `InitRandomAccess': text seg, init-y, pub-y, local-n, addr=0x004938d0, size=288
sym `ClearRandomAccess': text seg, init-y, pub-y, local-n, addr=0x004939f0, size=240
sym `UpdateRandomAccess': text seg, init-y, pub-y, local-n, addr=0x00493ae0, size=128
sym `FinalizeRandomAccess': text seg, init-y, pub-y, local-n, addr=0x00493b60, size=496
sym `CloseRandomAccess': text seg, init-y, pub-y, local-n, addr=0x00493d50, size=144
sym `init_ref_pic_list_reordering': text seg, init-y, pub-y, local-n, addr=0x00493de0, size=40
sym `start_slice': text seg, init-y, pub-y, local-n, addr=0x00493e08, size=704
sym `terminate_slice': text seg, init-y, pub-y, local-n, addr=0x004940c8, size=576
sym `encode_one_slice': text seg, init-y, pub-y, local-n, addr=0x00494308, size=4264
sym `free_slice_list': text seg, init-y, pub-y, local-n, addr=0x004953b0, size=568
sym `modify_redundant_pic_cnt': text seg, init-y, pub-y, local-n, addr=0x004955e8, size=1512
sym `ue_v': text seg, init-y, pub-y, local-n, addr=0x00495bd0, size=240
sym `se_v': text seg, init-y, pub-y, local-n, addr=0x00495cc0, size=240
sym `u_1': text seg, init-y, pub-y, local-n, addr=0x00495db0, size=240
sym `u_v': text seg, init-y, pub-y, local-n, addr=0x00495ea0, size=232
sym `ue_linfo': text seg, init-y, pub-y, local-n, addr=0x00495f88, size=280
sym `se_linfo': text seg, init-y, pub-y, local-n, addr=0x004960a0, size=320
sym `cbp_linfo_intra': text seg, init-y, pub-y, local-n, addr=0x004961e0, size=80
sym `cbp_linfo_inter': text seg, init-y, pub-y, local-n, addr=0x00496230, size=80
sym `levrun_linfo_c2x2': text seg, init-y, pub-y, local-n, addr=0x00496280, size=664
sym `levrun_linfo_inter': text seg, init-y, pub-y, local-n, addr=0x00496518, size=1032
sym `levrun_linfo_intra': text seg, init-y, pub-y, local-n, addr=0x00496920, size=1008

sym `symbol2uvlc': text seg, init-y, pub-y, local-n, addr=0x00496d10, size=104
sym `writeSyntaxElement_UVLC': text seg, init-y, pub-y, local-n, addr=0x00496d78, size=224
sym `writeSyntaxElement_fixed': text seg, init-y, pub-y, local-n, addr=0x00496e58, size=152
sym `writeSyntaxElement_Intra4x4PredictionMode': text seg, init-y, pub-y, local-n, addr=0x00496ef0, size=264
sym `writeSyntaxElement2Buf_UVLC': text seg, init-y, pub-y, local-n, addr=0x00496ff8, size=184
sym `writeUVLC2buffer': text seg, init-y, pub-y, local-n, addr=0x004970b0, size=304
sym `writeSyntaxElement2Buf_Fixed': text seg, init-y, pub-y, local-n, addr=0x004971e0, size=80
sym `symbol2vlc': text seg, init-y, pub-y, local-n, addr=0x00497230, size=120
sym `writeSyntaxElement_VLC': text seg, init-y, pub-y, local-n, addr=0x004972a8, size=160
sym `writeSyntaxElement_NumCoeffTrailingOnes': text seg, init-y, pub-y, local-n, addr=0x00497348, size=768
sym `writeSyntaxElement_NumCoeffTrailingOnesChromaDC': text seg, init-y, pub-y, local-n, addr=0x00497648, size=616
sym `writeSyntaxElement_TotalZeros': text seg, init-y, pub-y, local-n, addr=0x004978b0, size=552
sym `writeSyntaxElement_TotalZerosChromaDC': text seg, init-y, pub-y, local-n, addr=0x00497ad8, size=552
sym `writeSyntaxElement_Run': text seg, init-y, pub-y, local-n, addr=0x00497d00, size=552
sym `writeSyntaxElement_Level_VLC1': text seg, init-y, pub-y, local-n, addr=0x00497f28, size=360
sym `writeSyntaxElement_Level_VLCN': text seg, init-y, pub-y, local-n, addr=0x00498090, size=392
sym `writeVlcByteAlign': text seg, init-y, pub-y, local-n, addr=0x00498218, size=248
sym `estimate_weighting_factor_P_slice': text seg, init-y, pub-y, local-n, addr=0x00498310, size=2872
sym `estimate_weighting_factor_B_slice': text seg, init-y, pub-y, local-n, addr=0x00498e48, size=6712
sym `__do_global_dtors': text seg, init-y, pub-y, local-n, addr=0x0049a880, size=128
sym `__do_global_ctors': text seg, init-y, pub-y, local-n, addr=0x0049a900, size=296
sym `__main': text seg, init-y, pub-y, local-n, addr=0x0049aa28, size=88
sym `__divdi3': text seg, init-y, pub-y, local-n, addr=0x0049aa80, size=2720
sym `__libc_init': text seg, init-y, pub-y, local-n, addr=0x0049b520, size=48
sym `exit': text seg, init-y, pub-y, local-n, addr=0x0049b550, size=416
sym `_cleanup': text seg, init-y, pub-y, local-n, addr=0x0049b6f0, size=64
sym `__assert_fail': text seg, init-y, pub-y, local-n, addr=0x0049b730, size=256
sym `__stdio_check_funcs': text seg, init-y, pub-y, local-n, addr=0x0049b830, size=264
sym `__stdio_check_offset': text seg, init-y, pub-y, local-n, addr=0x0049b938, size=3896
sym `__flshfp': text seg, init-y, pub-y, local-n, addr=0x0049c870, size=1336
sym `__fillbf': text seg, init-y, pub-y, local-n, addr=0x0049cda8, size=1424
sym `__invalidate': text seg, init-y, pub-y, local-n, addr=0x0049d338, size=120
sym `fwrite': text seg, init-y, pub-y, local-n, addr=0x0049d3b0, size=1760
sym `printf': text seg, init-y, pub-y, local-n, addr=0x0049da90, size=112

sym `fflush': text seg, init-y, pub-y, local-n, addr=0x0049db00, size=512
sym `__getmode': text seg, init-y, pub-y, local-n, addr=0x0049dd00, size=568
sym `fopen': text seg, init-y, pub-y, local-n, addr=0x0049df38, size=296
sym `fclose': text seg, init-y, pub-y, local-n, addr=0x0049e060, size=640
sym `calloc': text seg, init-y, pub-y, local-n, addr=0x0049e2e0, size=160
sym `snprintf': text seg, init-y, pub-y, local-n, addr=0x0049e380, size=80
sym `_free_internal': text seg, init-y, pub-y, local-n, addr=0x0049e3d0, size=2224
sym `free': text seg, init-y, pub-y, local-n, addr=0x0049ec80, size=160
sym `pow': text seg, init-y, pub-y, local-n, addr=0x0049ed20, size=1680
sym `fprintf': text seg, init-y, pub-y, local-n, addr=0x0049f3b0, size=80
sym `memset': text seg, init-y, pub-y, local-n, addr=0x0049f400, size=368
sym `strncmp': text seg, init-y, pub-y, local-n, addr=0x0049f570, size=416
sym `strlen': text seg, init-y, pub-y, local-n, addr=0x0049f710, size=1784
sym `malloc': text seg, init-y, pub-y, local-n, addr=0x0049fe08, size=2440
sym `fseek': text seg, init-y, pub-y, local-n, addr=0x004a0790, size=1024
sym `ftell': text seg, init-y, pub-y, local-n, addr=0x004a0b90, size=336
sym `fread': text seg, init-y, pub-y, local-n, addr=0x004a0ce0, size=1216
sym `strcmp': text seg, init-y, pub-y, local-n, addr=0x004a11a0, size=96
sym `sscanf': text seg, init-y, pub-y, local-n, addr=0x004a1200, size=80
sym `strcpy': text seg, init-y, pub-y, local-n, addr=0x004a1250, size=80
sym `fscanf': text seg, init-y, pub-y, local-n, addr=0x004a12a0, size=80
sym `log10': text seg, init-y, pub-y, local-n, addr=0x004a12f0, size=80
sym `memcpy': text seg, init-y, pub-y, local-n, addr=0x004a1340, size=448
sym `rand': text seg, init-y, pub-y, local-n, addr=0x004a1500, size=48
sym `log': text seg, init-y, pub-y, local-n, addr=0x004a1530, size=1264
sym `__log__D': text seg, init-y, pub-y, local-n, addr=0x004a1a20, size=1376
sym `ceil': text seg, init-y, pub-y, local-n, addr=0x004a1f80, size=240
sym `ftime': text seg, init-y, pub-y, local-n, addr=0x004a2070, size=272
sym `time': text seg, init-y, pub-y, local-n, addr=0x004a2180, size=144
sym `srand': text seg, init-y, pub-y, local-n, addr=0x004a2210, size=48
sym `fputc': text seg, init-y, pub-y, local-n, addr=0x004a2240, size=304
sym `localtime': text seg, init-y, pub-y, local-n, addr=0x004a2370, size=432
sym `strftime': text seg, init-y, pub-y, local-n, addr=0x004a2520, size=3632
sym `sprintf': text seg, init-y, pub-y, local-n, addr=0x004a3350, size=800
sym `qsort': text seg, init-y, pub-y, local-n, addr=0x004a3670, size=1456
sym `floor': text seg, init-y, pub-y, local-n, addr=0x004a3c20, size=240
sym `sqrt': text seg, init-y, pub-y, local-n, addr=0x004a3d10, size=1088
sym `atexit': text seg, init-y, pub-y, local-n, addr=0x004a4150, size=136
sym `__new_exitfn': text seg, init-y, pub-y, local-n, addr=0x004a41d8, size=392
sym `__init_misc': text seg, init-y, pub-y, local-n, addr=0x004a4360, size=192
sym `abort': text seg, init-y, pub-y, local-n, addr=0x004a4420, size=16
sym `__stdio_read': text seg, init-y, pub-y, local-n, addr=0x004a4430, size=48
sym `__stdio_write': text seg, init-y, pub-y, local-n, addr=0x004a4460, size=248
sym `__stdio_seek': text seg, init-y, pub-y, local-n, addr=0x004a4558, size=120

sym `__stdio_close': text seg, init-y, pub-y, local-n, addr=0x004a45d0, size=48
sym `__stdio_fileno': text seg, init-y, pub-y, local-n, addr=0x004a4600, size=16
sym `__stdio_open': text seg, init-y, pub-y, local-n, addr=0x004a4610, size=296
sym `__stdio_reopen': text seg, init-y, pub-y, local-n, addr=0x004a4738, size=824
sym `__stdio_init_stream': text seg, init-y, pub-y, local-n, addr=0x004a4a70, size=320
sym `memchr': text seg, init-y, pub-y, local-n, addr=0x004a4bb0, size=496
sym `vfprintf': text seg, init-y, pub-y, local-n, addr=0x004a4da0, size=11632
sym `__newstream': text seg, init-y, pub-y, local-n, addr=0x004a7b10, size=320
sym `vsnprintf': text seg, init-y, pub-y, local-n, addr=0x004a7c50, size=400
sym `__finite': text seg, init-y, pub-y, local-n, addr=0x004a7de0, size=160
sym `__copysign': text seg, init-y, pub-y, local-n, addr=0x004a7e80, size=112
sym `__drem': text seg, init-y, pub-y, local-n, addr=0x004a7ef0, size=1408
sym `exp': text seg, init-y, pub-y, local-n, addr=0x004a8470, size=728
sym `__exp__D': text seg, init-y, pub-y, local-n, addr=0x004a8748, size=808
sym `__default_morecore': text seg, init-y, pub-y, local-n, addr=0x004a8a70, size=80
sym `__vsscanf': text seg, init-y, pub-y, local-n, addr=0x004a8ac0, size=368
sym `__vfscanf': text seg, init-y, pub-y, local-n, addr=0x004a8c30, size=8560
sym `_wordcopy_fwd_aligned': text seg, init-y, pub-y, local-n, addr=0x004aada0, size=528
sym `_wordcopy_fwd_dest_aligned': text seg, init-y, pub-y, local-n, addr=0x004aafb0, size=512
sym `_wordcopy_bwd_aligned': text seg, init-y, pub-y, local-n, addr=0x004ab1b0, size=544
sym `_wordcopy_bwd_dest_aligned': text seg, init-y, pub-y, local-n, addr=0x004ab3d0, size=544
sym `__srandom': text seg, init-y, pub-y, local-n, addr=0x004ab5f0, size=384
sym `__initstate': text seg, init-y, pub-y, local-n, addr=0x004ab770, size=976
sym `__setstate': text seg, init-y, pub-y, local-n, addr=0x004abb40, size=608
sym `__random': text seg, init-y, pub-y, local-n, addr=0x004abda0, size=352
sym `__logb': text seg, init-y, pub-y, local-n, addr=0x004abf00, size=384
sym `ldexp': text seg, init-y, pub-y, local-n, addr=0x004ac080, size=1024
sym `__tzset': text seg, init-y, pub-y, local-n, addr=0x004ac480, size=4552
sym `__tzname_max': text seg, init-y, pub-y, local-n, addr=0x004ad648, size=2512
sym `__tz_compute': text seg, init-y, pub-y, local-n, addr=0x004ae018, size=520
sym `__tzfile_read': text seg, init-y, pub-y, local-n, addr=0x004ae220, size=3064
sym `__tzfile_default': text seg, init-y, pub-y, local-n, addr=0x004aee18, size=632
sym `__tzfile_compute': text seg, init-y, pub-y, local-n, addr=0x004af090, size=1136
sym `gmtime': text seg, init-y, pub-y, local-n, addr=0x004af500, size=80
sym `__offtime': text seg, init-y, pub-y, local-n, addr=0x004af550, size=1856
sym `mbtowc': text seg, init-y, pub-y, local-n, addr=0x004afc90, size=560
sym `vsprintf': text seg, init-y, pub-y, local-n, addr=0x004afec0, size=336
sym `_quicksort': text seg, init-y, pub-y, local-n, addr=0x004b0010, size=1936
sym `__isnan': text seg, init-y, pub-y, local-n, addr=0x004b07a0, size=128
sym `__isinf': text seg, init-y, pub-y, local-n, addr=0x004b0820, size=112
sym `strrchr': text seg, init-y, pub-y, local-n, addr=0x004b0890, size=656
sym `__isatty': text seg, init-y, pub-y, local-n, addr=0x004b0b20, size=112
sym `register_printf_function': text seg, init-y, pub-y, local-n, addr=0x004b0b90, size=144
sym `strchr': text seg, init-y, pub-y, local-n, addr=0x004b0c20, size=416

sym `_itoa': text seg, init-y, pub-y, local-n, addr=0x004b0dc0, size=416
sym `__printf_fp': text seg, init-y, pub-y, local-n, addr=0x004b0f60, size=19056
sym `__sbrk': text seg, init-y, pub-y, local-n, addr=0x004b59d0, size=144
sym `realloc': text seg, init-y, pub-y, local-n, addr=0x004b5a60, size=1184
sym `strtol': text seg, init-y, pub-y, local-n, addr=0x004b5f00, size=1072
sym `strtoul': text seg, init-y, pub-y, local-n, addr=0x004b6330, size=992
sym `strtod': text seg, init-y, pub-y, local-n, addr=0x004b6710, size=1520
sym `ungetc': text seg, init-y, pub-y, local-n, addr=0x004b6d00, size=1752
sym `do_normalization': text seg, init-y, pub-y, local-n, addr=0x004b73d8, size=4568
sym `_mktime_internal': text seg, init-y, pub-y, local-n, addr=0x004b85b0, size=1096
sym `mktime': text seg, init-y, pub-y, local-n, addr=0x004b89f8, size=72
sym `getenv': text seg, init-y, pub-y, local-n, addr=0x004b8a40, size=384
sym `__mpn_extract_double': text seg, init-y, pub-y, local-n, addr=0x004b8bc0, size=256
sym `__mpn_lshift': text seg, init-y, pub-y, local-n, addr=0x004b8cc0, size=240
sym `__mpn_cmp': text seg, init-y, pub-y, local-n, addr=0x004b8db0, size=160
sym `__mpn_divmod_1': text seg, init-y, pub-y, local-n, addr=0x004b8e50, size=3712
sym `__mpn_mul_1': text seg, init-y, pub-y, local-n, addr=0x004b9cd0, size=144
sym `__mpn_add_n': text seg, init-y, pub-y, local-n, addr=0x004b9d60, size=160
sym `__mpn_divmod': text seg, init-y, pub-y, local-n, addr=0x004b9e00, size=2880
sym `__mpn_rshift': text seg, init-y, pub-y, local-n, addr=0x004ba940, size=224
sym `__mpn_sub_n': text seg, init-y, pub-y, local-n, addr=0x004baa20, size=256
sym `memmove': text seg, init-y, pub-y, local-n, addr=0x004bab20, size=784
sym `__mpn_submul_1': text seg, init-y, pub-y, local-n, addr=0x004bae30, size=192
sym `__umoddi3': text seg, init-y, pub-y, local-n, addr=0x004baef0, size=2272
sym `__udivdi3': text seg, init-y, pub-y, local-n, addr=0x004bb7d0, size=2656