AN ATTRIBUTE GRAMMAR BASED FRAMEWORK FOR MACHINE-DEPENDENT COMPUTATIONAL OPTIMIZATION OF MEDIA PROCESSING ALGORITHMS

Gene Cheung, Steven McCanne

Department of EECS, University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

Media processing algorithms are typically computationally intensive, and in complexity constrained environments, finding the most computationally efficient algorithm is critical. In this paper, we present an attribute grammar based framework which captures the computational complexity of an algorithm in a machine-dependent manner. Using this formalism, a media processing algorithm can be optimally and automatically tuned to a particular machine by a problem specific optimizer. Moreover, the tradeoff between performance and execution time on a specific machine can be controlled and thus exploited to optimize overall performance. To illustrate the viability of our approach, we applied it to the variable-length code (VLC) decoding problem and show that the optimal VLC decoding algorithm can be found using the framework. Tradeoff between coding efficiency and decoding speed of Huffman code can be exploited by employing length-limited code.

1. INTRODUCTION

The vast speedups in general purpose processors over recent years enables many complex media processing algorithms such as JPEG and MPEG1 to be implemented in software and executed on modern computers. Modern processors, however, have unique architectural features that may dramatically affect an algorithm's execution speed — stringent complexity constraints of a hand-held device processor, or the sizes and speeds of different hierarchical memories of a general-purpose processor. The existence of such features, together with the computation intensive nature of media processing algorithms, means that traditional compiler-level code optimizations may not be sufficient to fully realize all potential speedups. Instead, an algorithm can be computationally optimized for a specific machine at the algorithmic level. Formally, one may ask how to optimally tune a particular algorithm given the specification of a machine?

From a theoretical viewpoint, computational complexity has recently garnered interests in the study of optimal tradeoffs between complexity and performance (e.g. distortion) for specific transform-based image codecs [1], [2], [3]. These tradeoffs are typically performed in a machine-independent manner, using a generic metric such as number of multiplication operations. We believe, however, that a better approach is to adopt a machine-dependent metric, the *execution time*, and ask a related machine-dependent question: what is the optimal tradeoff between execution time and performance for a particular algorithm? This has important implications in practice; for example, a user may be willing to tolerate the increase in distortion of a video stream if the video encoding speed is increased.

In this paper, we present an attribute grammar based framework that addresses the two previously mentioned questions. For a given algorithm, a set of context-free grammar rules (CFG) defines a search space within the algorithm. Each terminal symbol in a rule represents a locally optimized patch of hand-written code, and has associated attributes that reflect the properties of the patch, such as code size and execution speed at a particular machine. An instance of an algorithm is thus represented by an ordered sequence of terminal symbols. By representing execution time as an attribute, we can optimally tune an algorithm by finding an ordered sequence of symbols that minimizes the execution time attribute. In so doing, we can formalize the notion of computational complexity given an algorithm and a specific machine. Moreover, the computationally optimal algorithm can be easily synthesized by performing code-stitching.

Our framework differs from existing works in three regards. First, by coupling the notion of algorithm computational complexity to a particular machine, we can more precisely estimate the actual computational cost of the executable when tuning an algorithm. Second, the framework provides an extensible and general foundation to study a general class of media processing optimization problems with computational complexity as objective or constraint. Finally, by performing machine-specific optimizations, we are able to optimally tune and automatically synthesize an implementation of an algorithm tailored for the chosen platform.

We first describe the proposed attribute grammar based framework and its theoretical implications in section 2. An important example of algorithm tuning using this framework, the variable-length code (VLC) decode problem, is discussed in section 3. Finally, we provide some concluding remarks and future work in section 4.

2. PROPOSED FRAMEWORK

2.1. Attribute Grammar based Framework

Our proposed framework, illustrated in Figure 1, is based on an attribute grammar (AG) formalism [4]. An AG is a set of context free grammar rules (CFG) decorated with attributes that are evaluated by attribute evaluation rules.

0-7803-5467-2/99/ $ 10.00 © 1999 IEEE

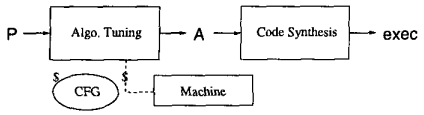


Figure 1: Attribute Grammar based Framework

A CFG, commonly used for formal language specifications, is a set of reduction rules, each of which specifies a set of right-hand side symbols that can be reduced to a left-hand side symbol. A sequence of reductions from an ordered sequence of terminal symbols to the start symbol S is called a *derivation* [1]. The role of CFG in our framework is to define a search space within an algorithm (*tunable space*), in which an instance of an algorithm can be uniquely specified.

Each CFG rule has a set of associated attributes, which reflects the properties (estimated execution time, data memory usage, etc) of the patch of locally optimized hand-written code that implements this rule. Attributes are evaluated using a set of *synthesized* attribute evaluation rules, which shows how the attribute values of the left-side symbol is evaluated from the attribute values of the right-side symbols. A set of CFG rules, attributes, and attribute evaluation rules, are collectively called an attribute grammar.

Given an attribute grammar, we can define the notion of *computational optimality* within our framework: Let B be the set of algorithm instances, whose syntax is described by CFG of a defined framework that solves a media processing problem P. Let the computational cost of an algorithm $b \in B$, evaluated using attribute evaluation rules, be $H(b)$. A *computationally optimal instance* to a media processing problem P is:

$$\arg\min_{b \in B} H(b) \tag{1}$$

If there are additional implementation constraints such as memory, then the computationally optimal instance is one that minimizes $H(b)$ while satisfying the constraints. Optimal tradeoffs between performance and complexity can be found by solving the constrained optimization problem multiple times, each time with a different constraint value. For example, if $D(b)$ is the distortion of an algorithm instance, we can find the optimal tradeoff between complexity and distortion by iteratively solving the following for different values of D_1:

$$\min_{b \in B} H(b) \qquad \text{s.t.} \quad D(b) \leq D_1 \tag{2}$$

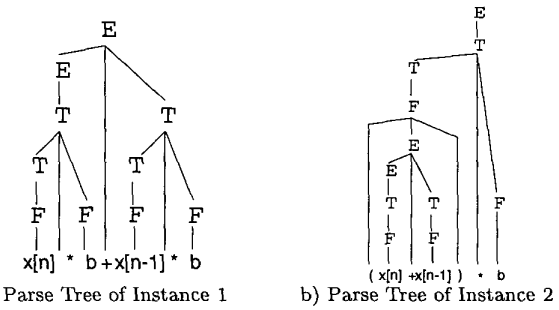
Even though the expressiveness of CFG restricts the tunable space of algorithm instances for a given problem, this restriction is not necessarily problematic in practice. During compilation, for example, code patches that correspond to various CFG terminal symbols need only be stitched up to constitute the algorithm, and so the machine-dependent algorithm tuning process can be easily automated by a problem specific optimizer-code synthesizer pair, as shown in Figure 1.

[1] If two derivations are possible from the same sequence of terminal symbols, the grammar is called *ambiguous*. For our purpose, we will assume all grammars are unambiguous.

attributes: **syn** cost **with** S, E, T, F **domain** {int}

```
1.   S -> E              S.cost = E.cost
2.   E -> E '+' T        E0.cost = E1.cost + T.cost + 1
3.   E -> T              E.cost = T.cost
4.   T -> T '*' F        T0.cost = T1.cost + F.cost + 2
5.   T -> F              T.cost = F.cost
6.   F -> '(' E ')'      F.cost = E.cost
7.   F -> num            F.cost = 0
```

Figure 2: AG for Expressions



a) Parse Tree of Instance 1 b) Parse Tree of Instance 2

Figure 3: Parse Trees for 2-tap Filter Algorithm

2.2. Filtering Example

As an example usage of the framework, we consider the following filtering problem. Suppose we want to find the fastest algorithm instance that implements a two-tap moving average filter. Consider the grammar rules of expression in Figure 2, where `cost` is the attribute that reflects execution time. A straight forward algorithm performs the following calculation at each discrete time n:

$$y[n] = x[n] * b + x[n-1] * b \tag{3}$$

After parsing this algorithm instance's description into a derivation tree (parse tree) shown in Figure 3a, the cost of this instance is calculated by evaluating the synthesized attribute `cost` using the attribute rules. In this case, the implementation cost is 5.

To see how the algorithm and its subsequent cost can be optimized, we can use associativity and rewrite the expression. The new algorithm instance is:

$$y[n] = (x[n] + x[n-1]) * b \tag{4}$$

The parse tree of this expression is shown in Figure 3b. The cost of this instance is 3. Therefore, we can conclude that this instance has a lower computational complexity than the previous one. In fact, we can make the following stronger statement: given the attribute grammar, the latter algorithm instance is in fact a computationally *optimal* instance that implements the two-tap filter. [2]

2.3. Implications of AG Framework

Using our framework to pose computational optimization problems has important theoretical implications. For a

[2] We conclude that the instance is computationally optimal by observing semantically that the filter cannot be implemented with fewer than one add and one multiply for the given AG.

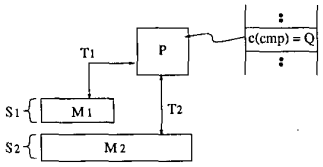
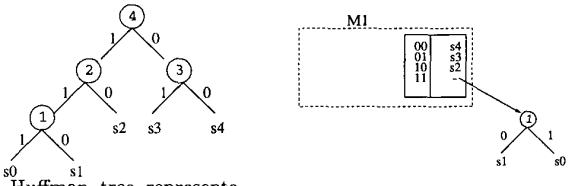


Figure 4: Hierarchical Memory Machine Model



a) Huffman tree representation

b) Hybrid of table and logic

Figure 5: Example of VLC Decoding

given AG, let B be the corresponding tunable space that solves a problem P. The computational complexity of an algorithm $b \in B$ is modeled by a series of synthesized attribute evaluations. If we now assume: i) all algorithm instances $b \in B$ are of polynomial lengths of terminal symbols, ii) an instance can be checked if it solves P in polynomial time (feasibility), and iii) each attribute evaluation can be evaluated in polynomial time, then the decision problem that corresponds to (1) — does that exist an algorithm instance that solve P with computation cost less than C? — must necessarily be non-deterministic polynomial (NP) [4]. The reason is that if the decision problem is true, then there exists a verifiable certificate algorithm instance b^* of polynomial length (assumption i), which can be checked against feasibility in polynomial time (assumption ii), and whose complexity can be computed in polynomial time (by a sequence of attribute evaluations, each of which takes polynomial time (assumption iii)). The theoretical implication is the following: because a great deal is known about the class of optimization problems NP, optimization problems posed using the framework can potentially be solved using traditional optimization techniques. In the next section, we will see such an example.

3. VLC DECODE PROBLEM

Many signal processing applications rely upon VLC to reduce distortion and/or encoding bit rate. For example, the Huffman code maps a sequence of recurring, statistically independent symbols into a minimally described bit sequence. Likewise, a pruned tree structured vector quantizer (PTSVQ) maps an input vector into a codeword in a finite-size codebook via multi-stage approximation. In both cases, the encoder maps each input symbol to a VLC; VLCs are then concatenated to form a bit stream.

The implementation of a VLC decoder often involves a tradeoff between computation and memory usage. Efficient data structures representing a set of VLCs often mean a slow decoding process, while fast VLC decoding algorithms usually require large memory space. Depending on cost criteria, numerous approaches have been offered in the literature. In contrast, our goal is to solve the VLC decode problem in a *machine-dependent* manner using our framework, where the objective is to find an algorithm instance that minimizes the average decoding time per codeword. We assume the target machine is a general purpose processor with hierarchical memories. To model the memory hierarchy, we constructed a machine model shown in Figure 4, with two levels of memories M_1 and M_2, with different sizes and memory access speeds. Because of the intricate interplay between decoding time and memory usage in VLC decoding, we design attributes and attribute evaluation rules in this problem to capture their mutual dependence in the next sections.

3.1. AG for Static Memory Model

Two common techniques for VLC decoding are table lookup and logical comparison. Consider the set of VLCs in Figure 5a, represented as a Huffman tree. A particular decoding algorithm is shown in Figure 5b, where a height 2-bit lookup table is first indexed, followed possibly by a logical comparison. The execution time of a given table lookup operation will depend on what type of memory the table is residing in; if the table resides in type 1 (type 2) memory, then the lookup operation will take T_1 (T_2) amount of time. The total size of tables assigned to type 1 memory by a decoding algorithm must not exceed the available type 1 memory of the machine, S_1. We assume that the size of level 2 memory is infinite, and that we can explicitly assign lookup tables to levels of hierarchical memory statically. An AG capable of expressing a decoding algorithm that uses a set of memory assigned lookup tables, with attribute rules that find the average lookup cost and memory usage, is shown in Figure 6. To expand the grammar to permit logical comparison as well, we add a small set of AG rules shown in Figure 7. The execution time of a logical comparison is Q.

Given such a grammar, we can express a VLC decoding algorithm instance that uses a mixture of lookup tables and programmed logics and evaluate its execution cost. For example, the algorithm instance in Figure 5b can be expressed as:

$$1.1.[(0, s_4, p_4), (1, s_3, p_3), (2, s_2, p_2), (3, 0.[(0, s_1, p_1), (1, s_0, p_0)])] \quad (5)$$

where p_j is the probability of symbol s_j. The cost of the instance, evaluating using the attribute rules, is: $p_0(Q + T_1) + p_1(Q + T_1) + p_2T_1 + p_3T_1 + p_4T_1$. More generally, we can compute the average decoding time per symbol of an instance b, denoted $H(b)$, as:

$$H(b) = \sum_j p_j(a_jT_1 + b_jT_2 + c_jQ) \quad (6)$$

where a_j (b_j) is the number of type 1 (type 2) memory access needed to decode symbol j, and c_j is the number of logical comparison needed. In other words, a_j (b_j) is the number of lookup tables residing in type 1 (type 2) memory used in the decoding process of symbol j.

Armed with this AG, we can pose a well-formed optimization problem:

$$\min_{b \in B} H(b) \quad \text{s.t.} \quad R(b) \leq S_1 \quad (7)$$

```
attributes:
  syn prob w/ T, M, m domain {real}    /* prob. of node */
  syn cost w/ S, T, M, m domain {real} /* lookup cost of node */
  syn size w/ T, M, m domain {int}     /* size of type 1 tables */

1.    S -> T                          S.cost = T.cost
2.    T -> h '.' Y '.' '[' M ']'      T.prob = M.prob
                  /* define lookup table */
                  if (Y.value == 1) {
                      T.cost = M.cost + M.prob * T_1
                      T.size = 2^h + M.size
                  }
                  else {    /* Y.value == 2 */
                       T.cost = M.cost + M.prob * T_2
                       T.size = M.size
                  }
                  if (T.size > S_1) ERROR;
3.    M -> M ',' m                    M0.prob = M1.prob + m.prob
                  M0.cost = M1.cost + m.cost
                  M0.size = M1.size + m.size
                  if (M0.size > S_1) ERROR;
4.    M -> m                          M.prob = m.prob
                  M.cost = m.cost
                  M.size = m.size
5.    m -> '(' int ',' id ',' real ')'
                                      m.prob = real
                   /* define table entry */
                   m.cost = 0
                   /* (index, symbol, entry probability) */
                   m.size = 0
6.    m -> '(' int ',' T ')'          m.prob = T.prob
                  m.cost = T.cost
                  m.size = T.size
7.    h -> int                        /* define table width */
8.    Y -> int                        Y.value = int
                  /* define memory type */
```

Figure 6: AG for Tables: Static Memory Model

```
1'.   S -> L                          S.cost = L.cost
2'.   L -> '0' '.' '[' m ',' m ']'
                 L.prob = m1.prob + m2.prob
                 /* define logic comparison */
                 L.cost = L.prob * Q + m1.cost + m2.cost
6'.   m -> '(' int ',' L ')'          m.prob = T.prob
                                      m.cost = T.cost
```

Figure 7: AG for Programmed Logic

where B is the set of feasible algorithm instances using lookup tables and logics, and $R(b)$ is the total size of lookup tables assigned to type 1 memory. In other words, given a set of VLCs and their associated probabilities, what is the optimal algorithm instance that minimizes the decoding time? Note that the optimal instance must not assign tables to type 1 memory in such a way that it exceeds the memory capacity of the machine model.

We have shown in [5] that (7) is an NP-hard problem and presented a branch-and-bound technique based on Lagrange multipliers that finds an approximate solution. We will later briefly discuss some results from [5].

3.2. AG for Dynamic Memory Model

In many processor architectures, programmers do not have explicit control on which level of hierarchical memory data structures reside in; movements are typically determined by the architecture's cache replacement strategies. In such cases, we adopt a dynamic memory model, which can be reflected by a slight change in the AG.

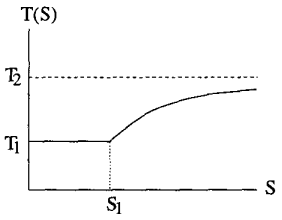


Figure 8: Memory Access Cost Function

```
1.    S -> T          S.cost = Time(T.size)*T.cost
2.    T -> h '.[' M ']'   T.prob = M.prob
                  /* define lookup table */
                      T.cost = M.cost + M.prob
                      T.size = 2^h + M.size
3-6.
```

Figure 9: AG for Tables: Dynamic Memory Model

Suppose the size of the data structures of an algorithm in memory, S, is less than or equal to S_1. Then the memory access time of a desired datum, $T(S)$, is always T_1, since all the data structures can be loaded into level 1 memory. If $S > S_1$, then the desired datum may not be in level 1 memory. We estimate the memory access time in this case as follow: a size S_1 portion of the S memory will be in level 1 memory at any given time. If all pieces of data are equally likely to be in the level 1 memory, then with probability $\frac{S_1}{S}$ we will find the data in level 1 memory, and with probability $\frac{S-S_1}{S}$ we will find the data in level 2 memory. Using this approximation, the memory access cost as a function of the size of the data structures in memory is:

$$T(S) = \begin{cases} T_1 & \text{if } S \le S_1 \\ (\frac{S_1}{S})T_1 + (\frac{S-S_1}{S})T_2 & \text{otherwise} \end{cases} \quad (8)$$

This function is shown in Figure 8. To evaluate the complexity of an algorithm instance $b \in B$, we need first to find the size of the data structures of the instance, $R(b)$. We then find the corresponding memory access cost $T(R(b))$. The execution time of an instance b will be a function of both b and $T(R(b))$. The difference in execution time evaluation is reflected by a change in attribute evaluation rules shown in Figure 9. The new optimization problem is:

$$\min_{b \in B} H(b,\ T(R(b)))\ ) \quad (9)$$

In [6], it is found that (9) is an NP-hard problem as well. It discusses a similar branch-and-bound technique that finds an approximate solution. We see that the new machine-dependent architectural detail can be captured by a slight change in the AG.

3.3. Results

We now discuss the performance of the VLC decoding algorithm instance generated by the branch-and-bound optimization technique in [5], based on AG in Figure 6, 7. For input, we use two different sets of VLCs. The first set

	Logic only	Lagrange	Full Table	M & T
H.263 VLC	3.87	4.76	4.32	3.82
PTSVQ	2.67	4.16	3.70	-

Figure 10: Results VLC Decode, in mil. lookups/s

is the motion vector VLC table of the H.263 video compression standard, with a 13-bit longest codeword. Using the technique in [5], we found a near-optimal algorithm instance and automatically generated a VLC decoder for the machine parameters: $S_1 = 16kB$, $T_1 = 1$, $T_2 = 3$, and $Q = 0.5$. They were estimates for our testbed, a 266MHz Pentium processor.

To create a test bit stream, we generated 10 million random codewords using the available codeword probabilities. Using our testbed machine, we executed our VLC decoder 20 times on the bit stream to obtain an average lookup cost. In Figure 10, we first compare the performance of our decoder to two other decoders: i) full table lookup algorithm that uses a single lookup table of 2^h elements, where h is the height of the Huffman tree; and, ii) logical comparison only algorithm that decodes one bit at a time using `if` statements. Note that full table lookup algorithm is the optimal VLC decoder if one uses number of operations as the metric, and logic-only algorithm is optimal if one uses number of memory access as the metric — both of which are machine-independent metrics. The results show that our decoder is 10.2% faster than the full table algorithm, and is 23.0% faster than the logic-only algorithm. Since the set of VLCs is a canonical code, we are able to compare our algorithm to algorithm ONE-SHIFT in [7]; Figure 10 shows that our algorithm has a 24.6% improvement over algorithm ONE-SHIFT.

The second set of VLCs is the codebook of a pruned tree-structured vector quantizer. The training set used for the construction of the codebook consists of three 512x512 grey-scale images: the well-known `lena`, `tiffany` and `baboon` images. Using the training set as input to the program, we obtained the PTSVQ codebook and the codeword probabilities for vector dimension 4 and rate 5. The longest codeword was length 15. In Figure 10, we see that we have 12.4% improvement over the full table algorithm and 55% improvement over the logic-only algorithm. We see that by using the proposed framework, we are able to capture the machine-dependent computational complexity when optimizing an algorithm, and as a result, our optimizations outperform those based on machine-independent techniques.

We can also trade off between encoding rate of a lossless variable-length code and the decoding time. VLC decoding using lookup tables is difficult when the length of the longest codeword is very long compared to the average codeword length, i.e. the Huffman tree is very skewed. Suppose we employ a length-limited code [8] instead, where we systematically reduced the length of the longest codeword. For each new set of length-limited VLCs, we find a near-optimal algorithm instance using technique in [5]. In Figure 11, we see this tradeoff between encoding rate and decoding time can be exploited by modifying the motion vector VLC table as mentioned above.

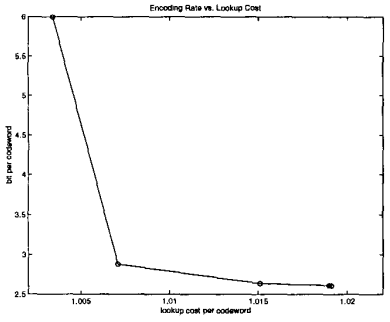


Figure 11: Tradeoff between Bit Rate and Decoding Time

4. CONCLUSION & FUTURE WORK

In this paper, we presented an attribute grammar based framework for the computational optimization of media processing algorithms. Using our framework, machine dependent computational optimality can be formally defined. After finding the optimal algorithm instance using a problem specific optimizer, an implementation of the algorithm can be automatically generated. In particular, we show how the VLC decoding problem can be optimized using the framework. In the future, we plan to investigate optimizations of other computationally intensive algorithms using the framework, such as the unconstrained VQ encoding problem.

5. REFERENCES

[1] V. Goyal, M. Vetterli, "Computation-Distortion Characteristics of Block Transform Coding," *ICIP 97*, pp.2729-2732, 1997.

[2] M.Gormish, J.Gill, "Computation-rate-distortion in transform coders for image compression," *SPIE vol.1903 Image and Video Processing*, pp.146-152, 1993.

[3] K.Lengwehasatit, A.Ortega, "Distortion/Decoding Time Tradeoffs in software DCT-based Image Coding," *ICASSP 97*, 1997.

[4] M.Sipser, *Introduction to the Theory of Computation*, December, 1996.

[5] G.Cheung, S.McCanne, C.Papadimitriou, "Software Synthesis of Variable-length Code Decoder using a Mixture of Programmed Logic and Table Lookups," *DCC 99*, Jan, 1999.

[6] G.Cheung, S.McCanne, "Dynamic Memory Model based Optimization and Code Synthesis for IP Address Lookup," Submitted for publication in *ICNP 99*, May, 1999.

[7] A.Moffat, A.Turpin, "On the Implementation of Minimum Redundancy Prefix Codes," *IEEE Trans. Comm.*, vol.45, No.10, pp.1200-1207, October 1997.

[8] J.Katajainen, A.Moffat, A.Turpin, "A Fast and Space-Economical Algorithm for Length-Limited Coding," *Proc. Int. Symp. Algorithms and Computation*, pp.12-21, Dec, 1995.