Automated Synthesis of Efficient Binary Decoders for Retargetable Software Toolkits

Wei Qin
wqin@ee.princeton.edu

Sharad Malik
sharad@ee.princeton.edu

Department of Electrical Engineering
Princeton University
Princeton, NJ 08544

ABSTRACT

A binary decoder is a common component of software development tools such as instruction set simulators, disassemblers and debuggers. The efficiency of the decoder can have a significant impact on the efficiency of these software tools. Automated synthesis of efficient binary decoders is therefore necessary for retargetable software tool development frameworks targeting the rapidly growing field of application-specific processor design. This paper describes a decoder synthesis algorithm that translates a simple instruction pattern specification into efficient binary decoders in C under given memory constraints. The algorithm constructs a decision tree with carefully chosen decoding primitives and cost models. As demonstrated through two case studies, the synthesized decoders achieve efficiency comparable to hand-coded decoders with ensured correctness. The algorithm has no limitation on the input instruction patterns and it requires only the least amount of knowledge about the instruction encoding. Therefore it can be used with any machine description scheme containing instruction encoding information.

Categories and Subject Descriptors

F2.2 [**Nonnumerical Algorithms and Problems**]: Sorting and Searching; I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis

General Terms

Algorithms, Performance

Keywords

binary decoder, decoding tree, decision tree, instruction set simulator

DAC 2003, June 2–6, 2003, Anaheim, California, USA.

1. INTRODUCTION

A growing number of application-specific processors are being designed and deployed in modern electronic systems. The development of these processors requires not only hardware development tools for synthesis and verification at both the logic and physical level, but also software development tools that fully exploit the programmability of the processors. These software tools include synthesis tools such as compilers, assemblers and linkers, and verification tools such as disassemblers, debuggers and simulators at various abstraction levels.

Compared with general-purpose microprocessors, application-specific instruction sets have a shorter life time and smaller volumes. So vendors are often reluctant to invest limited resources on a complete software tool chain. Thus it is desirable that these tools be generated automatically from high level processor specifications. As a result, we have seen an increasing number of processor description language driven retargetable software tool synthesis frameworks [4, 6, 8, 14] in both academia and industry.

Many of the software development tools share one common component – the binary decoder that translates a stream of binary words into an instruction stream. Unlike hardware-based decoding where multiple logic expressions can be evaluated concurrently, software decoding is sequential and control flow intensive. Therefore, the speed of a binary decoder can be very slow and may become a major performance bottleneck for speed-critical software tools such as the instruction set simulators (ISSs). According to our experience and the results reported by other researchers [13], a slow decoder can affect the simulation speed of the ISSs by a factor of 2 to 4. An efficient binary decoder is thus highly desirable.

This paper addresses the problem of automatic synthesis of efficient binary decoders for arbitrary instruction set architectures. The synthesized decoders can be used in the software development tools mentioned above. Because of their high efficiency, they may also be used in operating systems or as micro-code to interpret unimplemented instructions. Here we focus on the problem of decoding opcode fields. Decoding of operand fields is straightforward once the instruction opcode is decoded.

The paper is organized as follows. Section 2 describes related work in the field, Section 3 formulates the problem, and Section 4 presents our solution. We describe experimental results for two case studies in Section 5 and present some conclusions in Section 6.

2. RELATED WORK

Efficient hand-coded binary decoders for general purpose processors can be found in popular tools such as the GNU debugger [5]. A typical decoding scheme is to extract the main opcode field of the instruction and then perform a multi-way branch based on its value. After the main opcode is decoded, the sub-opcode fields can be handled in a similar way. The hand-coded decoders require human intelligence that is not available in automatic decoder synthesis. For complex and irregular instruction sets, it is an error-prone task for humans to find a good solution.

A simple binary decoder synthesis scheme is described in [7]. The generated decoder sequentially matches the input bit string with all possible instruction patterns that were extracted from a description in the machine description language ISDL [6]. The execution time of such a decoding scheme is linear in the number of instructions of the instruction set. Since a typical instruction set contains more than 100 instructions, the sequential decoding scheme can be very slow.

In [18], a decision-tree based decoding scheme is described. Each internal node of the decision tree tests a few bits of the input bit string and makes a multi-way branch to the matching child node. The process iterates until a leaf node is reached where the instruction can be unambiguously classified. The algorithm is deterministic in that the synthesized decoding tree is completely determined by the input instruction set specification. The algorithm generates decoders with efficiency comparable to that of hand-coded decoders in the reported cases. A known problem of the algorithm is that it will fail on certain instruction pattern combinations, which the author assumes will not appear. The assumption is not fully justified since such a situation may occur in application-specific processor designs where irregular encoding is preferable due to code size considerations.

The New Jersey Machine-Code Toolkit [15] is capable of synthesizing binary decoders from machine specifications of a special format. In order to generate an efficient binary decoder, the instruction patterns must be cleanly factored and grouped into tables in the specification. In a retargetable software tool development environment where a more general machine description language is used, it is a non-trivial task to derive such a well-organized specification, especially when the instruction set is irregular.

[13] addresses the efficiency of binary decoding in ISS from a different angle. The proposed technique exploits the locality of the program under simulation by caching the decoding results for reuse. When the cache hit rate is high, decoder efficiency becomes less of a problem. However, caching the decoding results consumes a large portion of the precious data cache and therefore negatively impacts the simulation speed. Furthermore, the decoding cache hit ratio is subject to the locality characteristics of the program being simulated and therefore the performance cannot be guaranteed.

The problem of decoder construction is closely related to the long studied multi-discipline field of decision tree construction from data [11, 12]. Specifically, the problem of binary decoding is very similar to the problem of identification key construction [16], which has been studied in the fields of systematic biology, pattern recognition, fault diagnostics, etc. However, due to the different context, objective and problem scale, no existing solution can be directly borrowed from these fields. To our knowledge, binary decoder construction has not been discussed in the decision tree construction field.

```
(000--, 11, 0.25)
(001--, 12, 0.25)
(01---, 13, 0.25)
(1----, 14, 0.25)
```

(a) Well-formed

```
(001--, 11, 0.25)
(000--, 12, 0.25)
(-1---, 13, 0.25)
(1----, 14, 0.25)
```

(b) Not well-formed

Figure 1: Example pattern sets

3. PROBLEM FORMULATION

3.1 Definitions

The task of a binary decoder is to find the matching instruction for a bit string. To clearly model the search process and to formalize the discussion in the paper, we define several terms below.

We define a bit pattern $p \in \{0, 1, -\}^n$, where 0 and 1 are binary values and "$-$" stands for a don't-care value. A bit string $s \in B^n$ ($B = \{0, 1\}$) matches a pattern p if and only if $\forall 0 \leq i \leq n$, either $s[i] = p[i]$ or $p[i] = -$. We write $s \in p$ if s matches p.

Each instruction is represented by a decoding entry in the form of a triple (p, l, λ), where p is a bit pattern of length n, $l \in L$ a classification label and $\lambda \in R$ the probability that a bit string will match p. An instruction set can be represented by a set of such decoding entries. As pointed out in [18], for a variable length instruction set, we can pad the short patterns with "$-$"s so that all patterns are of the same length.

Given a set of decoding entries E, the task of a binary decoder $d : B^n \mapsto L$ is to map a bit string s to the classification label of decoding entry $(p, l, \lambda) \in E$ so that $s \in p$. We define the *capacity* of E as the total number of unique bit strings that can match a pattern in E. The set E is said to be *well-formed* if there exists no bit string that matches more than one entry. Figure 1(a) shows a well-formed pattern set example, while Figure 1(b) shows a not-well-formed pattern set, in which bit string "11000" matches both the third and the fourth entry.

In logic synthesis terminology, a pattern specifies a product term in the space of B^n. Well-formedness can be verified by checking that the products of all pattern pairs are zero.

3.2 Decoding Tree

Decoding is a search process. Common searching algorithms, including hashing, can all be represented by search trees [3]. The problem of decoder construction is therefore equivalent to the construction of a min-cost search tree.

We define a decoding tree (V, E) similarly to the decision tree in [18]. The node set $V = D \cup N$, where D is the set of terminal nodes and N the set of inner nodes. Each terminal node is labeled with either a decoding entry or a null entry. Each inner node $v \in N$ is labeled with a decision function $f_v : B^n \mapsto Z$, where Z is the set of integers. Each possible evaluation result of f_v corresponds to an outgoing edge of v, which is labeled with the result value.

The decoding process starts from the root node. It iter-

atively evaluates the decision function of the current node with the input bit string s as the argument, and descends along the edge labeled with the evaluation result. The process repeats until a terminal node is reached. If the node is a decoding entry, then its classification label is the decoding result; if the node is a null entry, then a decoding error is reported. A decoding error occurs when s matches no given pattern. Given a decoding tree, the decoding height $H(s)$ is defined as the number of edges from the root node to the matching terminal node. Figure 2 shows a possible decoding tree for the decoding entry set of Figure 1(a). The decoding path for bit string "01101" has a height of 2.

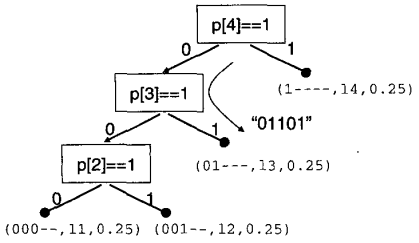


Figure 2: Example decoding tree

The construction of a decoder involves selecting the tree structure as well as the set of decision functions for the inner nodes.

3.3 Cost Modeling

Since this paper addresses efficient decoder synthesis, we take average execution time of the decision function as the decoding cost. However, we cannot know the actual cost until the entire decoder is constructed, compiled and tested, which is impractical if the construction process involves the evaluation of a huge number of candidate decoders. Therefore, it is necessary that we model the execution time at a higher level.

Assuming that the execution time of each decision function in the decoding tree is constant, we can take the average decoding height as a measure of decoding time, which is defined as

$$H_{avg} = \frac{1}{K}\sum_{i=1}^{K} H(s_i) = \sum_i \lambda_i \cdot D(e_i), \qquad (1)$$

where K is the total number of decoded bit strings, and $D(e_i)$ is the path length from the root to the terminal node.

Note that the execution time of a modern microprocessor is affected not only by the length of the execution trace but also by its memory usage. So in order for the above estimation to be reasonably accurate, we must ensure that the synthesized decoder uses only a limited amount of memory. If memory usage is unlimited, the smallest decoding height can always be achieved by a lookup table of size 2^n.

In summary, the decoder construction problem can be stated as below: from a well-formed decoding entry set E, construct a decoding tree d_{min} so that d_{min} has the minimum average decoding height under given memory usage constraints. The input of such a problem requires the least amount of knowledge about instruction encoding formats and can be easily obtained from any form of machine description containing encoding information.

4. DECODER CONSTRUCTION

4.1 Decision function

In general, the decision functions can be constructed from arbitrary arithmetic or logic operators and their combinations. So there exist an infinite number of possible candidates for decision functions. To simplify the selection of decision functions, we allow for only two classes of simple decision functions as shown below.

1. Pattern decoding

   A pattern decoding function tries to match the bit string s with a pre-specified pattern. The function returns 1 if the two match and 0 otherwise. Since the function has only two possible results, a node with a pattern decoding function always has two children.

2. Table decoding

   A table decoding function extracts m contiguous bits from the bit string as its result. Such a function has 2^m possible outcomes. Therefore a node with a table decoding function has 2^m children.

The two classes are chosen since they are commonly used in hand-coded decoders. They can be implemented efficiently as single C statements if the bit string of size n can fit into a built-in data type of C. Such efficiency is desirable since it keeps the decoder small and fast, and the similar costs of the functions validate our execution time assumption for Equation (1).

The contiguity constraint that we impose on table decoding keeps the decoding function simple. It also helps to limit the number of table decoding functions to $n(n+1)/2$. On the other hand, if non-contiguous bits are allowed, there would be $2^n - 1$ total functions, which is usually too large to handle. The constraint is not a serious problem in practice since table decoding is most useful for decoding the opcode fields of an instruction set, which are often contiguous.

4.2 Division of Decoding Entry Set

At the start of the decoding process, we view all decoding entries as possible decoding outcomes since there exists a path from the root node to any leaf node. Once the decision function of the root node is evaluated, we can descend along the edge corresponding to the evaluation result to one child-node v. At this point, the possible decoding outcomes contain only the leaf nodes of the sub-tree under v, which constitute a subset of the entire decoding entry set. In other words, the evaluation of a decision function f divides a decoding entry set into a set of smaller ones by "revealing" information from the bit string. Such division provides for a means to divide and conquer the decoding problem.

To understand how a decision function f divides a decoding entry set E into $\{E_i\}$, we consider two cases for each entry $(p, l, \lambda) \in E$,

1. If $\forall s \in p$, $f(s) = i$, then (p, l, λ) is added to set E_i.

2. If bit strings matching p evaluate to several results, then we split the entry to a smallest set $\{(p_i, l, \lambda_i)\}$ so that $\forall s \in p_i$, $f(s)$ is a constant c_i, and $\cup p_i = p$, $\sum \lambda_i = \lambda$ with λ_i's linearly proportional to the probabilities that s matches p_i's. We add entry (p_i, l, λ_i) to E_{c_i}.

The second case above involves splitting decoding entries, which is one major difference between our algorithm and the one in [18]. Figure 3(a) shows a division example of a table decoding function extracting the left-most two bits. Figure 3(b) shows one possible decoding tree constructed from the division. As we can see, the average height of the tree is 1.5. In comparison, for the algorithm in [18] in which splitting is not allowed, Figure 2 is the only valid decoding tree with an average decoding height of 2. Clearly, splitting enables the trade-off of tree width with tree height and allows for faster decoding.

We call a decision function useless if one $E_i = E$ and the rest are all empty. For example, a pattern decoding function with the pattern "11---" is useless to the sub-tree in Figure 3(b). In such a case, the decision function does not reveal any new information.

4.3 Evaluation of Decision Function

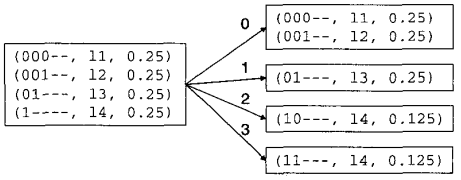
In order to find the best decoding tree with the two classes of decision functions, the simplistic solution is to exhaustively search for the best decoding tree. The procedure *find_tree*(E) below takes a decoding entry set and returns the decoding tree with the minimum decoding cost.

1. If $|E| = 1$, return a terminal node labeled with the entry in E.
2. Initialize set F with all decision functions, which are possible for our search space. Set H_{min} to ∞. Pick the current decision function f_c from F and remove f_c from F.
3. Divide E with f_c into $\{E_i\}$. If f_c is useless, go to Step 5. Otherwise, for each $E_i \neq \phi$, recursively call *find_tree*(E_i) and obtain its best decoding tree d_i. Then calculate the decoding cost of f_c as
$$H_c = 1 + \sum_i \Lambda_i \cdot H_i,$$
where H_i is the decoding cost of tree d_i and Λ_i is the total probability of E_i.
4. If $H_c < H_{min}$, let $H_{min} = H_c$ and $d_{min} = (f_c, \{d_i\})$.
5. If F is not empty, then pick a new f_c, remove it from F and go back to Step 3. Otherwise, return d_{min} as the min-cost decoding tree.

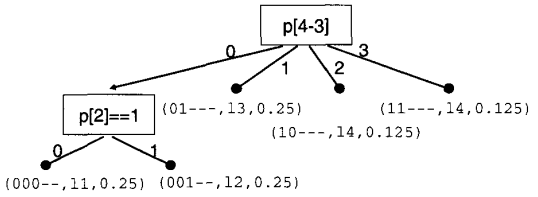
The *find_tree* procedure is guaranteed to terminate since the capacity of each E_i in Step 3 is smaller than that of E. However, the space that it explores is extremely large. Recall that a pattern $p \in \{0, 1, -\}^n$. So there exist $3^n - 1$ pattern decision functions to evaluate. The maximum recursion depth is related to the capacity of E and can be as large as 2^n.

In order to find a practical solution, instead of recursively calculating the best decoding cost for each subset in Step 3, we try to estimate the decoding cost of the subsets.

A common cost estimation heuristic used for decision tree construction [11] is Shannon's entropy [17], which is used as a measure of the randomness. In coding theory, Shannon's entropy is known to be the theoretical lower bound of the average length of binary codes [2]. A closely related but tighter bound is the height of the Huffman tree [9]. Intuitively, the average code length and the decoding tree height



(a) Table division



(b) Resulting decoding tree

Figure 3: Example of division

are both related to the randomness of the data and hence are correlated. Therefore, we adopt the height of the Huffman tree as a measure of decoding difficulty, or a measure of decoding cost.

Recall that the decision functions may split decoding entries and increase the size of the decoding tree. To avoid excessive splitting, we need to model the memory efficiency of a decision function quantitatively.

A decoding tree may consume memory in two ways: for decision functions and for decoding tables. To ensure that the memory usage of the decoding tree is reasonable, we adopt a simplified memory model by assuming that a decision function or a decoding table entry consumes one unit of memory. Therefore, the pattern decoding function consumes one unit of memory, while the table decoding function consumes $1 + 2^m$ units of memory.

We use a pattern decoding tree without splitting as the baseline of memory usage. Since such a decoder is a binary tree, we have $|D| = |E|$ and $|N| = |E| - 1$. So it consumes $|E| - 1$ units of memory.

We define the memory efficiency ratio of a decision function as the ratio of the estimated memory usage after and before the division, as is shown below:

$$m_r = S/(|E| - 1)$$

and

$$S = \begin{cases} |E_0| + |E_1| - 1, & \text{for pattern decoding,} \\ \sum_{E_i \neq \phi}(|E_i| - 1) + 1 + 2^m, & \text{for table decoding.} \end{cases}$$

For a pattern decoding function involving no splitting, m_r is 1. For splitting pattern decoding or table decoding, $m_r > 1$.

In our decoding cost estimation, we combine the Huffman tree height and a memory usage penalty term as below.

$$H_c = 1 + \sum_i (H_i \cdot \Lambda_i) + \gamma \cdot \log_2 m_r,$$

where H_i is the Huffman tree height of E_i and $\gamma \geq 0$ is the penalty factor. When γ is large, the penalty term will increase the cost significantly for the splitting cases and for table decoding. In our experimental implementation, to avoid excessive splitting when γ is 0, we filter those decoding functions with m_r below 0.1.

4.4 Further Pruning of Search Space

Although we have only two classes of decoding functions, the size of F is still quite large since there exist $3^n - 1$ pattern decoding functions. To prune the search space, we use a pattern growing heuristic. First, we find the best single-bit pattern, i.e. a pattern with only one bit as 0 or 1 and the rest as "–", by enumerating the n possibilities. Then we grow the best single-bit pattern to a 2-bit pattern by finding another bit which yields the minimum cost when combined with the single-bit pattern. We iteratively grow the pattern until an additional bit in the pattern no longer reduces the cost. We take the resulting pattern as the best pattern decoding function.

The space of the best table decoding function is much smaller. However, it is still impractical to evaluate all these functions because the division complexity is proportional to 2^m for table decoding. To simplify the task, we start by evaluating all $(n-1)$ 2-bit tables. (1-bit table decoding is the same as the single-bit pattern decoding case.) After finding the best m-bit table decoding function, we try to find the best $(m+1)$-bit function from all $(n-m)$ candidates. The process stops if one more bit does not reduce the cost any further. The best function found is taken as the best table decoding function.

We then compare the best pattern decoding function and the best table decoding function and pick the better of the two as the decision function for the current node.

5. EXPERIMENTAL RESULTS

To evaluate the effectiveness of the decoding algorithm, We programmed a decoder synthesizer in C++. The input to the synthesizer is a decoding entry set. The output is a decoder in C.

We then performed an experimental study on two popular instruction sets: the ARM instruction set [1] and the PowerPC instruction set [10]. Our experiments were all performed on a P-III 800MHz Linux-based workstation with 2GB main memory.

We described the ARM instruction set as 137 decoding entries, each containing a label as the instruction name and a probability obtained through profiling over a set of SPECInt benchmarks. To catch illegal bit strings, we computed the unused opcode space as the complement of the union of all patterns. After logic minimization, the unused space is expressed as 50 decoding entries, each containing a label as the name of an error handler, and a tiny probability so that it is not ignored by the synthesizer. Similarly, the PowerPC instruction set description contains 148 decoding entries. The unused opcode space contains 130 entries.

By varying the penalty factor γ, our synthesizer generated a series of decoders. The run time of the synthesizer is less than 10 seconds for each case. Figure 4 shows the average decoding tree heights of the decoders for different γ's, and Figure 5 shows the memory usage. Generally when γ decreases, the decoding tree height decreases as more decoding tables are used.

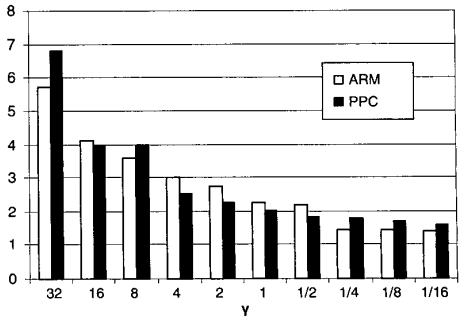


Figure 4: Decoding tree heights

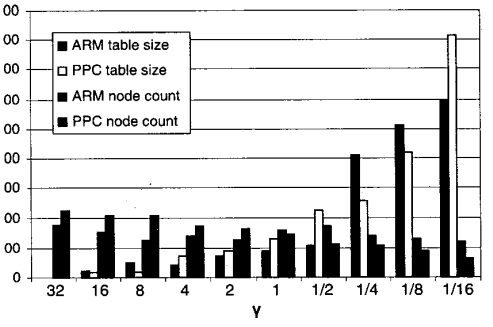


Figure 5: Memory usage of the decoders

The results show that the synthesizer can generate efficient decoders with an average height of less than 2 and with less than 1,000 table entries. Such small decoders can easily fit into the cache of the host machine and have little impact on the performance of the applications using them. Such decoding cost is no higher than hand-coded decoders or the decoding cache in [13].

To evaluate the run time efficiency of the synthesized decoders, we linked the decoder with ISSs and tested the average execution speed of the decoder over a set of SPECInt benchmarks. We used GCC to compile the ISS and the decoder with optimization switches "-O3 -fomit-frame-pointer".

As γ varies from 32 to 1/16, the simulation speed of the ARM ISS varies from 8.38MIPS (million instructions per second) to 9.56MIPS, and the speed of the PowerPC ISS varies from 7.10MIPS to 8.18MIPS. When γ is under 1/2, the speed variation becomes negligible as the average decoding height changes very little. In comparison, the ARM ISS with hand-coded decoder runs at 8.88MIPS and the PowerPC one at 8.15MIPS.

Profiling for instruction frequencies is not always desirable since it is time consuming. So we studied the cases when such profiling results are not available. In such cases, we assigned homogeneous probability for each instruction pattern. We found that the resulting decoders are similar in both speed and memory usage for both the instruction sets, especially when γ is below 1. This is because table decoding can resolve multiple patterns simultaneously and is independent of the pattern probabilities.

In Table 1 we compared the synthesized decoders with

	ARM		PowerPC	
	H_{avg}	MIPS	H_{avg}	MIPS
Sequential	48.4	4.77	58.5	4.24
Trained Seq.	6.76	7.84	8.88	7.11
[18]	2.47	7.72	1.59	8.16
$\gamma = 1/16$	1.41	9.56	1.58	8.18

Table 1: Decoder result comparison

$\gamma = 1/16$ against three other decoding schemes: a sequential decoder with instructions simply sorted by their names, another sequential decoder sorted by decreasing order of instruction frequency, and a decoder based on the algorithm in [18]. For fast decoding speed, we implemented the tree nodes of [18] as direct-addressed tables.

We noticed a significant difference between the non-trained sequential decoding and the trained one, which is due to the fact that compilers tend to use a small set of instructions more frequently than others. In the trained cases, the decoders try to match with the most frequent instruction patterns first and get better results on the average. However, for benchmarks differing greatly in instruction frequency from the training set, the result can be much worse. For instance, the PowerPC decoder trained by the SPECInt benchmarks yields an average decoding height of 29.9 for SPECFp benchmarks and an average simulation speed of 5.27MIPS.

The algorithm in [18] generates very good decoding trees. For the PowerPC instruction set, which is regularly laid out as two levels of opcodes, the resulting decoding tree is almost of the same height as ours. However, since the tree nodes of the algorithm involve testing non-contiguous bits of the instruction word, the resulting decoders are slower than ours. Moreover, we found that for both instruction sets it is possible to add new instruction patterns that cause the algorithm of [18] to fail.

The evaluation results here were all based on the ISSs we used. The ISSs have a slow down factor of around 100, which means that on average it takes about 100 native instructions to interpret a target instruction. For faster ISS implementations, the efficiency benefit of the decoders will be more significant.

6. CONCLUSIONS

The paper addresses the problem of fast decoder synthesis from simple instruction pattern specifications. We model the problem as decision tree construction. By carefully choosing decoding functions and cost models, we have designed effective heuristics that guide the tree construction process. The resulting synthesizer can generate fast binary decoders with quality comparable to hand-coded ones as well as guaranteed correctness. The decoder synthesizer has no limitation on the input instruction patterns and can be used as part of a retargetable software tool development framework.

7. ACKNOWLEDGMENTS

This work is part of the MESCAL project of the Gigascale Silicon Research Center sponsored by DARPA/MARCO. We thank the anonymous reviewers for their invaluable comments.

8. REFERENCES

[1] Advanced RISC Machines Ltd. *Arm Architecture Reference Manual*, 1996.

[2] T. M. Cover and J. A. Thomas. *Elements of information theory*. Wiley, New York, 1991.

[3] D.E.Knuth. *The Art of Computer Programming, Vol.3:Searching and Sorting*. Addison-Wesley, Reading, MA, 1973.

[4] A. Fauth, J. V. Praet, and M. Freericks. Describing instructions set processors using nML. In *Proceedings of Conference on Design Automation and Test in Europe*, pages 503–507, Paris, France, 1995.

[5] Free Software Foundation, Inc. `http://www.gnu.org/software/gdb/gdb.html`, Dec 2002.

[6] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proceedings of Design Automation Conference*, pages 299–302, June 1997.

[7] G. Hadjiyiannis, P. Russo, and S. Devadas. A methodology for accurate performance evaluation in architecture exploration. In *Design Automation Conference*, pages 927–932, 1999.

[8] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of Conference on Design Automation and Test in Europe*, pages 485–490, 1999.

[9] D. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40:1098–1101, 1952.

[10] International Business Machines Corporation. *PowerPC Microprocessor Family: The Programming Environments for 32-bit Microprocessors*, 2000.

[11] B. M. E. Moret. Decision trees and diagrams. *ACM Computing Surveys*, 14(4):593–623, 1982.

[12] S. K. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data Mining and Knowledge Discovery*, 2(4):345–389, 1998.

[13] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of Design Automation Conference*, pages 22–27, 2002.

[14] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA – machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings of Design Automation Conference*, pages 933–938, 1999.

[15] N. Ramsey and M. F. Fernandez. The New Jersey Machine-Code Toolkit. In *USENIX Technical Conference*, pages 289–302, 1995.

[16] R.W.Payne and D.A.Preece. Identification keys and diagnostic tables: a review. *Journal of the Royal Statistics Society, Series A*, 143(3):253–292, 1980.

[17] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, July, October 1948.

[18] H. Theiling. Generating decision trees for decoding binaries. *ACM SIGPLAN Notices*, 36(8):112–120, 2001.