Efficient Software Decoder Design

Rajeev Krishna, Todd Austin
Advanced Computer Architecture Laboratory
University of Michigan
{rkrishna, taustin} @ eecs.umich.edu

Abstract:

In this paper, we evaluate several techniques for generating and optimizing high speed software decoders. We begin by presenting the early stages of a new instruction set description language named 'Rosetta'. We use specifications written in this language to automatically generate a number of different software decoders. We explore heuristics for generating decoder trees, particularly with regard to enumerating "don't care" bit positions during evaluation in order to reduce decode tree depth and thus increase performance. We also investigate the application of cache-conscious data placement techniques, decoder structure, and the effects of non-contiguous bit sequences on decoder performance. By applying these techniques to decoders produced for the ARM and IA32 (x86) instruction sets, we are able to produce highly flexible decoders that are comparable in size and performance to carefully hand-coded, hand-optimized decoders with substantially less programmer time and effort.

Section 1: Introduction

Until fairly recently, efficient decoding of an instruction set in software has not been of significant interest beyond the research community, specifically those researchers involved in simulation and testing of new hardware architectures. With the introduction of languages such as Sun's Java [1] and virtual machine based hardware infrastructures such as Transmeta's Crusoe processor line [2], the problem of generating efficient software decoders has grown in importance. The fundamental issue in both of these domains is that of high speed dynamic binary translation. The overall performance of Java's virtual machine interface and Crusoe's software translation layer is dependent on the ability to quickly translate from 'non-native' instruction formats to instructions executable on the host architecture.

One of the major components of any such binary translation infrastructure is the software decoder. The ability to quickly identify the non-native instruction is clearly a factor in overall translation speed. In this paper we attempt to analyze several approaches to automatic generation of such high speed decoders. We believe that the key to decoder performance lies in the ability to divide groups of instructions by evaluating large sequences of contiguous bits, minimizing the cost of traversing each stage of the decode tree as well as the average tree depth. While previous work in this area has focused on contiguous or non-contiguous sequences of bits with fixed values for all instructions at a particular step in the decode process (as opposed to bit positions representing "don't care" values in some or all instructions, termed 'unbound bits' from here on), we believe that dynamically expanding (fully enumerating) don't care positions as needed to allow set division on longer useful contiguous bit sequences may provide a substantial performance benefit by decreasing decoder tree depth. Such an optimization can only realistically be performed through automated techniques, as the usefulness of unbound ("don't care") bit positions would be rather difficult for a programmer to evaluate.

In order to facilitate automatic decoder generation, we begin by presenting the early stages of a new ISA specification language called Rosetta. This language is inspired by regular expression format, and attempts to use a similarly concise form to describe instruction syntax. Our intent is to expand the language to capture semantic information. At this stage, however, Rosetta provides a very straighforward way to describe instruction syntax. This specification is translated into a flattened internal representation, providing an abstraction from the aesthetic qualities of the specification (useful for human understanding), and allowing decoder generation to focus on aspects of the specification that are directly relevant to decoder structure.

We begin our analysis by considering the usefulness of allowing unbound bit expansion during generation of the decoder through a fairly straightforward bit selection heuristic that requires specification of a maximum number of unbound bits. We also consider data placement techniques, reordering decoder state to maximize memory locality of frequently traversed decode nodes. Finally, we evaluate decoder structure, testing data-centered table based decoders against instruction-centered switch statement decoders. We evaluate the performance of resulting decoders it terms of raw decode performance (determined by number of processor cycles required for an average decode) and cache performance (through the Cheetah cache simulation library). From the information gathered in these tests, we attempt to refine out technique. We re-tool the bit selection heuristic to better account for the nature of unbound bits, resulting in better automation and the elimination of the "maximum unbound bits" constraint. We also change the focus of the evaluation to the more efficient switch statement based decoders, and consider the effect of non-contiguous bit sequences.

We begin this evaluation by considering related work in Section 2. Section 3 presents a brief overview of the Rosetta specification language syntax, as well as a discussion of the internal representation produced from the specification. The remainder of the paper discusses bit selection and tree generation techniques. Since the primary focus of this work is efficient bit selection and decoder generation, we will present some generic techniques on tree compaction and decoder generation in Sections 4 and 5. These techniques are common to all of the decoder generation heuristics evaluated and are thus presented independently. Section 6 covers the primary aspects of the work, decoder performance evaluation. Finally, Section 7 summarizes and provides concluding remarks.

Section 2: Related Work

Several groups have done work in the area of instruction set specification and decoder / simulator generation. Vengroff [3] presents a tool called 'decgen' that translates a simple specification format into ISA decoders. The specification language is very straightforward, but does not appear to provide sufficient expressivity to easily capture instructions with numerous syntactic forms or optional and variable length fields. For example, in order to express the multiple syntactic forms of a single instruction in the IA32 (x86) ISA, it appears necessary to explicitly describe each valid syntax as a separate instruction description. The decoders produced by decgen are table based, but appear to function by traversing multiple linked tables, making memory location optimizations difficult. The decoders also consider only contiguous sections of globally bound bits, a constrain which we believe can be relaxed, resulting in a increase in decoder performance.

The New Jersey Machine Code Toolkit [4] uses the SLED [5] specification language to generate assemblers and disassemblers. SLED provides a class based description format for specifying field locations and names, with separate pattern statements specifying constraints on bit positions. We believe a description format based on regular expressions can capture both field and pattern information in a more readable format. The toolkit generates decoders by building a decision tree based on token sequences provided by the specification writer. We attempt instead to create an internal abstraction that is completely independent of the nuances of the specification in order to better isolate the syntax information that is relevant to the decode process.

Architecture Description Language (ADL) is a specification language used by the UPFAST simulator generator [6]. It provides a format not only to describe instruction set syntax, but instruction semantics and architectural semantics as well, allowing the automatic creation of complete microarchitecture simulators as well as the obvious assemblers and disassemblers. The syntax specification format used in ADL, however, seems to fall victim to the same potential problems as decgen, and it is somewhat unclear how useful ADL would be in specifying variable length instruction sets or instruction sets with optional fields.

The SimpleScalar Toolset [7] provides syntax and semantic descriptions through definitions written as C macro commands. These macro definitions can be targeted to the requirements of the individual using the toolset. The definitions are translated directly into simulator code at compile time by the C preprocessor. Though this abstraction provides a clean way of describing decoder semantics, it requires that the entire decode tree be specified manually in the definition file, placing the entire burden of building the decoder upon the programmer.

It should be noted that none of the specification tools described above currently make provisions for decoder structure optimization based on usage statistics. Though this is obviously a second order affect, we wish to explore potential benefits of such optimizations in operation environments with large working sets. Work in the area of cache conscious data placement and structure layout [8,9] suggests that making such considerations in data placement can improve program performance by reducing cache miss rates. Building off of the basic idea of these works, we attempt to exploit knowledge of the problem domain and usage information acquired through profiling to arrange decoder state in a cache conscious manner.

The Rosetta toolset produces tree based decoders emitted as finite state machines. This allows the toolset to optimize decoders using techniques from both tree processing and DFA optimization. In particular, as part of their work on efficient path profiling [10], Ball and Larus present a numbering technique using edge increments that produces a unique numeric value for each unique path through an arbitrary tree. This work provided the basis for the annotation method used in the Rosetta tree compaction algorithm. This compaction algorithm attempts to minimize the number of true states used by the DFA through a method related to standard DFA optimization employed by compilers for regular expression DFA generation [11]. The standard technique minimizes states through systematic partitioning of the state space by distinguishing input sequences. The algorithm used by the Rosetta toolset achieves the same result by condensing all identical subtrees of the complete decoder graph, using the Ball/Larus path numbering technique to help generate self-similar subtrees.

Section 3: Rosetta Specification Language

The Rosetta ISA specification language is intended to provide a format for completely describing the syntax and semantics of an instruction set. As a first step toward this goal, we have developed a concise method for defining valid instructions in an ISA. This format was originally inspired by regular expression description format, though the final form has undergone significant alterations to better accommodate the domain. Among these modifications is the inclusion of field names and replacement parameters allowing the designer to easily label specific subsequences for later use or to re-use common sub expressions. Consider a simple example shown in Figure 3.1 from the ARM7 ISA. This example serves to demonstrate the basic structure of Rosetta's expression format. An instruction group is entered with a 'definst' statement. This block contains minimally a 'match' section within which valid instruction formats are described, and optionally a 'bind' section which allows for parameterization based on fields named in the match section. Within the match section, a 'mainseq' identifier is used to specify the expression that describes valid instruction. This main sequence may be preceded by any number of subsequences (denoted 'subseq' in later examples) which represent what are effectively macro definitions that may be referenced in the future. In this example, both the match and bind sections are fairly straightforward. The match section contains a single sequence description identifying various fields such as registers and

Figure 3.1: ARM7 Multiply Instruction (Rosetta Specification)

```
definst("Multiply")
{
 match {
 mainseq =  {
        Cond(----).000000.A(-).S(-).>Rd(----).>Rn(----).
                                           >Rs(----).1001.>Rm(----)
    };
 }

 bind {
    switch(A) {
        case 0: { OP = "mul"; }
        case 1: { OP = "mla"; }
    }
 }
}
```

instruction flags. These are described simply as a series of bit positions, many of which are just unbound don't care terms (denoted by the '-'). The sequence also specifies certain bit positions that are bound to specific binary values for all instances of the instruction. The bind section does not affect decoding, serving only to parameterize on OP field for later use in semantic descriptions.

Though this example serves as a good introduction to the specification format used by Rosetta, it does not really demonstrate the power of the language as applied to instruction set specification. To further explore the features provided by this specification technique we turn to the slightly more challenging example of describing the data processing instructions from ARM7. The match section for this definition is shown in Figure 3.2. Here we see the first use of subsequences to aid in the clarity of the definition. The data processing instructions in ARM7 have two forms, one to utilize a register value as an operand, and one to specify an immediate value. The subsequence format allows the intricacies of these forms to be described in isolation, then introduced into the main matching sequence at their appropriate locations. In this example, the condition field and register specifiers have also been replaced by macros. These macros, rather than being defined with the instruction block, were previously defined in a global section that has the same semantics as a 'definst' block but is used only to hold globally useful parameters or commands and does not resolve to any actual instructions. This example clearly demonstrates how regular expression type matching can be used to concisely capture specific constraints in the set of valid instructions. Note, for example, the use of the exclusion ('^') operator to create specific constrains in the OPMATCH subsequence, and the square bracket shorthand notation for repeating a particular sequence (*e.g.*; '[-.5]' means five don't care bits).

The other important feature to note is the '>' symbol which appears before many of the field names in the example. This identifier is used to indicate to the Rosetta toolset that the marked field does not actually affect the semantics of the instruction. For example, while the 'OP' field is vital to determining the behavior of an instruction (in this case, selecting between addition, subtraction, etc.) and must be enumerated in the evaluation process in order to correctly identify an instruction, the register specifier 'Rd' carries no such semantic value. Rather, it serves as a parameter to the instruction and need not be evaluated until an actual executing instruction is available. This information is used as a hint by the toolset while flattening the regular expression sequences representing an instruction (described later).

The distinction between semantically relevant fields and parameter fields leads us to question what actually constitutes an explicit instruction. In the strictest sense, there is no reason for the OP field to be given any more weight than the Rd field during the decode process. One could simply choose to leave the OP field unexpanded, resulting in a decoder that is only able to distinguish groups of instructions. The job of identifying precisely which instruction is represented by a bit sequence could easily be left to the semantic section, for example by scanning the OP field at run-time. We believe, however, that every attempt should be made to push semantically relevant fields into the decoder. This allows the specification to take advantage of decoder optimization techniques discussed later in this paper, and moves towards isolating instruction selection semantics from instruction execution semantics. We believe this will make the resulting specifications far more concise and far easier to understand.

The opposite end of the argument, expanding all fields during the decode process, also has far less merit than a balanced approach. The claim that parameter fields (such as the Rd field) do not carry the same semantic value as non-parameter fields comes from the observation that the values of such fields are used directly and do not affect the control flow of the instruction. Thus, while there is some benefit in distinguishing between an ADD and a SUBTRACT instruction at the decoder level, there is no benefit from distinguishing between and ADD to R1 and an ADD to R2. Throughout this work, we adopt the philosophy that fields representing semantic information should be expanded, while fields representing instruction parameters or arguments should not.

As a final example, let us consider specification of the Intel IA-32 instruction set, specifically the MOD/RM byte, which presents some rather challenging syntax. The IA-32 specification for Rosetta begins by describing the entire syntax of the MOD/RM, SIB, and displacement bytes as a global subsequence. Since the architecture allows for a 16 bit and 32 bit operating mode, separate specifications are provided for each. The specification of the MOD/RM bytes for the 32 bit

Figure 3.2: ARM7 Data Processing Instructions (Rosetta Specification Match Section)

```
 match {
        subseq OPMATCH() = { OP(^10--).S(-) | OP(10--).S(1) };
        subseq REGMODE() = { { >ShAmt([-.5]).>ShType(--).0
                                        | >ShReg($REGISTER()).0.>ShType(--).1 }. >Rm($REGISTER())};
        subseq IMMMODE() = { >RotAmt([-.4]).>Imm([-.8]) };
        mainseq = { $COND_MATCH().00.{ 0.$OPMATCH().>Rn($REGISTER()).>Rd($REGISTER()).$REGMODE()
                                              | 1.$OPMATCH().>Rn($REGISTER()).>Rd($REGISTER()).$IMMMODE()}
        };
```

Figure 3.3: 32-bit MOD/RM decoding sub-sequence for IA-32 Instruction Set

```
subseq MOD_REGOP_RM32() = {  >MOD(00).>REGOP(---).{ >RM(^100 & ^101)
                                                    | >RM(100).>SCALE(--).>IDX(---).{ >BASE(^101)
                                                                                    | >BASE(101).>DISP([-.32]) }
                                                    | >RM(101).>DISP([-.32]) }
                           | >MOD(01).>REGOP(---).{ >RM(^100).>DISP([-.8])
                                                    | >RM(100).>SCALE(--).>IDX(---).>BASE(---).>DISP([-.8])   }
                           | >MOD(10).>REGOP(---).{ >RM(^100).>DISP([-.8])
                                                    | >RM(100).>SCALE(--).>IDX(---).>BASE(---).>DISP([-.32])  }
                           | >MOD(11).>REGOP(---).>RM(---)
                           };
```

operating mode is shown in Figure 3.3. The MOD/RM specifications can then be used to concisely specify rather complicated instruction syntax, as shown in the specification of the general purpose arithmetic instructions in Figure 3.4. In order to properly capture information on the current machine state, our specification assumes that prefixes modifying the current operand, address, and segment size are parsed out of the instruction before decode, and a one byte sequence representing the result of these prefixes applied to the current machine state is provided as the first 'byte' of the instruction. These are represented by the 'O', 'A', and 'S' values in the specification. the 'PREFILL' parameter is simply a set of pad bits filling up the remaining bits in the first byte. The MOD/RM replacement sequences are used extensively in this specification to provide a concise description. The 'MOD_REGOP_RM' sequence (in its 16 bit and 32 bit forms) is utilized in the first section of the specification. The 32 bit version of this sequence was described in Figure 3.3. The 'MOD_OPS_RM' subsequence is identical to the 'MOD_REGOP_RM' sequence, except that the reg/op field of the MOD/RM byte now constitutes an opcode extension. The 'MOD_OPS_RM' sequences thus expand this field, rather than suppress it with the '>' identifier.

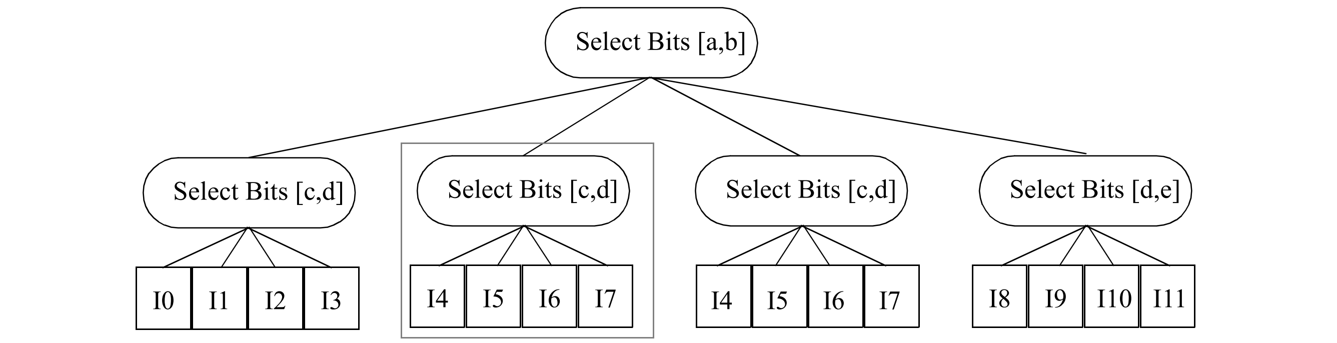
As these examples demonstrate, regular expression syntax provides a powerful way to concisely describe the syntax of an instruction set. In order to make use of these descriptions, however, the Rosetta toolset must extract valid instructions and annotate them with appropriate binding (and, in the future, semantic) information. This is done by the previously mentioned process of regular expression flattening. In the first stage of processing, the description of a set of instructions is completely flattened to enumerate all valid instruction sequences specified. During the flattening process, field names and locations are attached to each individual instruction sequence. It is at this point that the non-semantic field specifier (the '>' symbol) becomes vital to managing the size of the resulting set of instructions. Marked fields are expanded only if they place a constraint on valid instruction sequences, and may later be compacted to reduce the total size of the flattened instruction set. The vital point here is that this identifier is the only hint required by the toolset, and it's use is fairly straightforward. Use of parameters and field-names to group bits together have no effect at all on generated decoders.

Once the regular expression description of an instruction block has been completely flattened, binding information is applied to each instruction in the set, and the resulting construct is added to a running list of all valid instructions (represented by other instruction blocks). The final list of valid instruction sequences generated by this process forms the basis for all further processing on the instruction specification. It is important to note that conflicts between instructions are not completely evaluated at this stage of processing, as "don't care" terms are left unexpanded and may create conflicts. Any conflicts remaining in the instruction set will, however, be identified during decoder generation described in the later sections.

Figure 3.4: Specification of arithmetic operations for IA-32 instruction set (match section)

```
match {
    subseq PRIOP() = {00.OPCODE(---)};
    mainseq = { $PREFILL().>O(-). { A(0).>S(-)..$PRIOP().0.d(-).w(-).$MOD_REGOP_RM16()
                                  | A(1).>S(-)..$PRIOP().0.d(-).w(-).$MOD_REGOP_RM32()
                                  }
              | $PREFILL().>O(-). { >A(-).>S(-)..$PRIOP().10.w(0).$IMM8()
                                  | A(0).>S(-)..1000.00.sextend(-).w(0).$MOD_OPS_RM16().$IMM8()
                                  | A(1).>S(-)..1000.00.sextend(-).w(0).$MOD_OPS_RM32().$IMM8()
                                  }
              | $PREFILL().O(0). { A(-).>S(-)..$PRIOP().10.w(1).$IMM16()
                                 | A(0).>S(-)..1000.00.sextend(-).w(1).$MOD_OPS_RM16().$IMM16()
                                 | A(1).>S(-)..1000.00.sextend(-).w(1).$MOD_OPS_RM32().$IMM16()
                                 }
              | $PREFILL().O(1). { A(-).>S(-)..$PRIOP().10.w(1).$IMM32()
                                 | A(0).>S(-)..1000.00.sextend(-).w(1).$MOD_OPS_RM16().$IMM32()
                                 | A(1).>S(-)..1000.00.sextend(-).w(1).$MOD_OPS_RM32().$IMM32()
                                 }
              };
}
```

Figure 4.1: Initial Decode Tree for Compaction



Section 4: Tree Compaction

The process of generating a decoder from the instruction set specification begins with the flattened set of semantically distinct instructions described in the previous section. We then recursively divide this set of instructions into subsets based on common bit-slices. The result of this set division is a tree of bit selection information (*e.g.*, evaluate the numeric value represented by bits three through six and select the next node appropriately) which, if followed, terminates in a leaf node representing the decoded instruction. The structure of the decoder tree itself is fairly straightforward. As such, the decoder generation discussion in Section 6 will focus on the heuristic used to select which bits to evaluate at each stage of the decode tree. While we believe the application of our heuristics will produce reasonably efficient decoder trees, the resulting tree may still be quite large due to self-similar regions in the instruction set (for example, the address mode decoding for the IA32 instruction set creates large groups of identical subtrees along paths leading to distinct instructions). We wish to be able to compact common subtrees in order to minimize the overall memory size of the decoder tree, however in order to do so we must convert the leaf nodes of the tree (the instructions) into a more generalizable abstraction. Otherwise, subtrees that are identical in all respects except the final set of instructions could not be combined without introducing potentially complicated book keeping that would have to be performed at runtime, reducing the speed of the decoder. Since our tree compaction technique is common to all heuristics presented, we describe it briefly before moving on to the actual heuristic evaluation.

The problem of common subtree compaction comes down to one of path identification. If two distinct subtrees can be combined into one without loosing the ability to distinguish between them during the decode process, then the combination technique can be successful. Intuitively, this requires some annotation at the incoming edges of the common subtree to identify which of the logical subtrees is actually being traversed, differentiating the subtree based on the path used to reach it. An annotation technique that solves this precise problem is found in work by Ball and Larus on efficient profiling techniques [10]. In this technique, edges of a graph are annotated with counter incrementation values. The sum of these increments for any traversal of the graph produces a number in the range [0,n] where n is the number of possible paths. This number uniquely identifies the path taken through the graph. Since the process of compacting common subtrees turns the decoder tree into a directed acyclic graph, this technique is directly applicable. It turns out, however, to be too precise for our needs. We are not actually interested in identifying the specific path taken through the tree. Rather, we are only concerned with the instruction at the end of that path. Due to replication caused by the evaluation of unbound bits during tree generation, these goals are not necessarily identical. Furthermore, though instructions can easily be identified by the final path increment value, instruction replication effectively increases the number of leaf nodes (and thus number of paths) in the decode tree to a value greater than the actual number of instructions decoded, potentially requiring a mapping between the path identifier and the instruction. For our purposes, we would actually prefer to be able to select explicitly the (potentially non-unique) numeric value generated by each path, and annotate the tree such that this value is generated. We therefore make a slight modification to the approach used by Ball and Larus. Consider the simple decode tree in Figure 4.1. Note that two of the subtrees are completely identical, and three are identical except for the instructions. We will demonstrate how our modified path annotation technique allows all three of these subtrees to be combined, maintaining necessary distinctions between trees without introducing unnecessary ones.

The procedure begins at the leaf nodes. The numeric identifier for each instruction is annotated to the incoming edge as an increment. The result of this procedure on the marked subtree from figure 4.1 is shown in Figure 4.2a. The path increments are then normalized to the smallest increment, resulting in a base increment for the subtree and a set of edge increments, as shown in Figure 4.2b. Once this procedure has been completed for each leaf node, the procedure is repeated for each of the intermediate nodes, using the annotated base increment values to the same effect that the instruction enumerations had in the first iteration. The annotation procedure is completed when the head of the tree has been reached. By ensuring that some instruction has an enumeration of zero, we ensure that the base increment of the head of the decoder tree is also zero, thus reducing the results to a set of path incrementation values associated with the edges of the tree. The completed tree is shown in Figure 4.3.

Figure 4.2a

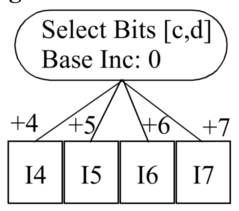


Figure 4.2b

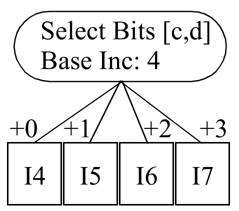


Figure 4.3: Annotated Decode Tree for Compaction

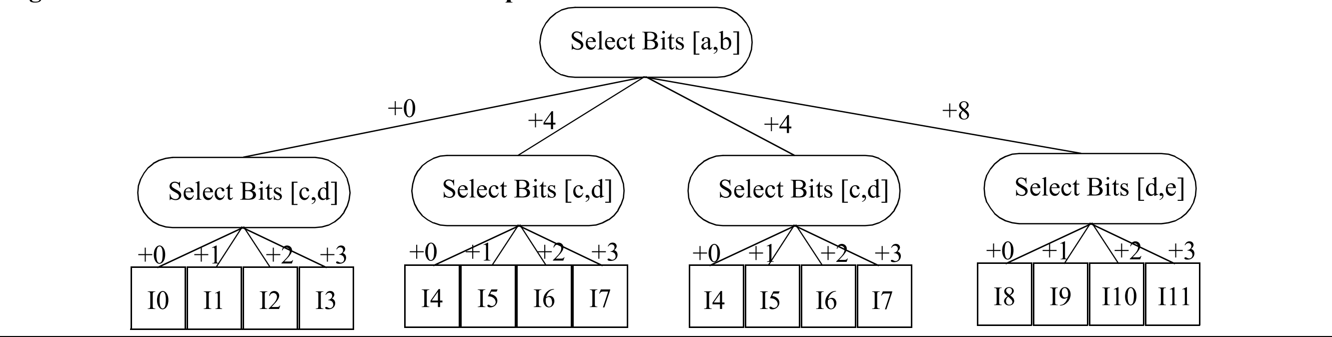
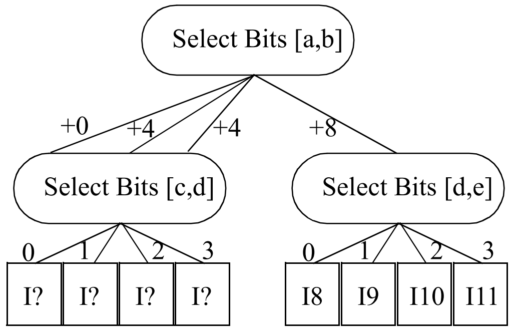


Figure 4.4: Compacted Decode Tree



With the path annotations in place, it is now possible to compact all identical subtrees resulting in Figure 4.4. It is clear that all of the relevant information from the original tree has been preserved in the compacted structure. It is also important to note with this technique that identical subtrees need not terminate in the same instructions. Rather, the burden has been shifted to the relative order of instruction enumerations. In order to expose the greatest similarity in subtrees, we begin the annotation procedure by performing a depth first traversal of the decode tree, assigning leaf enumerations in order when possible, and re-mapping instruction enumerations to this new order. Thus, the relative order of instructions within a subtree should be identical (in order) whenever the set of instructions identified by the subtree permits.

Section 5: Decoder Generation

The most direct approach to decoding a set of instructions is to select a single bit position at each stage that is fixed in all instructions in the set. This divides the set of instructions into two subsets which may then be further divided. Vengroff [3] observes that the availability of a bit position that is bound in all instructions is not a strict requirement for decodability. Rather, so long as all instructions are pairwise distinguishable, the instruction set is decodable. The artificial restriction of bound bits in all instruction, however, does not present a problem in practice.

Though single bit selection is conceptually simple, it has the potential to generate rather deep decode trees resulting in slow decoder performance. Ideally, we wish to minimize the depth of the resulting decode trees while also minimizing the amount of work that must be performed at each level of the decode tree. The single bit solutions meets the second criteria, but not the first. The first step to resolving this problem is to allow selection based on several bits at once. We explore two related heuristics for selecting bits as part of our evaluation in Section 6.

Either of the bit selection heuristics considered can be applied to a set of instructions to produce a decode tree. The generic tree compaction technique discussed in the previous section can be applied to this decode tree to reduce its overall state. This results in a second abstract internal representation (a compacted decode tree). From this representation, we are able to explore various approaches to generating a final decoder in C code. Specifically, we emit the decoder state in the form of a single table (array) of integer values operated upon by a table walking function, or we emit the state as a series of nested switch statements, representing the decoder state directly in code.

The choice of output method affects one other stage of our evaluation. For either decoder type, we attempt to re-order physical state based on node usage profiles in order to minimize working set size. For the table based decoders, this optimization takes the form or physical re-ordering of nodes in the decoder array to maximize locality of accesses to the decoder table. For the switch statement based decoders, this optimization amounts to re-ordering of case-blocks to maximize access locality in the instruction stream.

Section 6: Decoder Evaluation

To perform evaluations, decoders generated from ARM7 and IA32 Rosetta specifications were tested against decoders from the SimpleScalar toolset [7]. As described previously, SimpleScalar uses a macro based description language in concert with the C pre-processor to describe a complete tree based decoder in table form at compile-time. This represents a carefully hand-coded, hand-optimized decoder structure, providing a very competitive baseline for comparison. We will consider the average number of cycles needed to decode instructions, and the cache performance of each decoder.

The evaluation software consists of a tight loop, issuing instructions to a single decoder from instruction tracefiles of common benchmark programs (SPEC 2000). This method was chosen in order to better isolate decoder performance from the effects of other unrelated code. This was particularly useful for cache performance evaluations which were acquired using SimCheetah, the SimpleScalar front end to the ‘cheetah’ cache simulation library [12]. This simulation environment was configured to not perform cache flush on context switches, effectively eliminating the effects of tracefile read calls and allowing nearly complete isolation of the cache behavior of the decoder under test.

All decoder speed tests were performed on a Pentium-III system, allowing access to the architecture's cycle counter register for reasonably fine-grain raw performance measurements. In order to ensure that cycle count values represented a valid metric, relative values were compared against relative wall clock time measurements. While time measurements were unable to consistently resolve the actual time spent in the decoder, the cycle count values did correspond reasonably with measurements of total execution time, indicating that the cycle count metric does correlate well with decoder performance and thus constitutes a valid measurement tool.

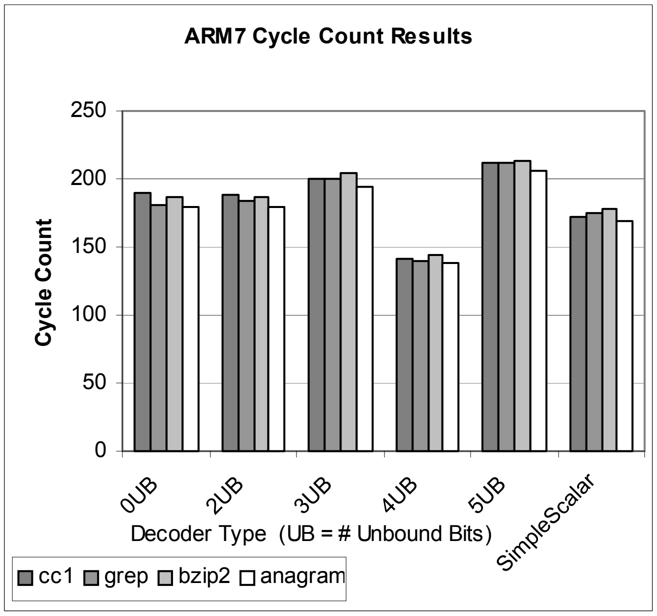
6.1 - Usefulness of Unbound Bit Expansion

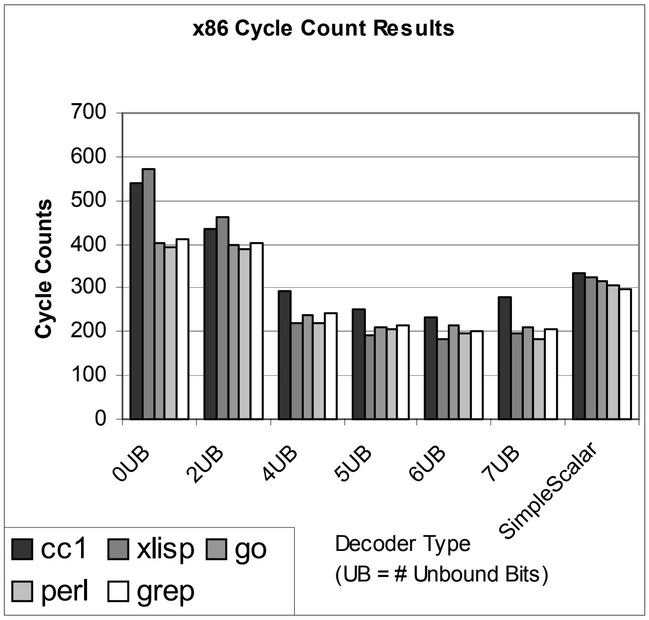
While the ideal situation would be to isolate a large, contiguous group of bound bits (recall that these are bit positions that have fixed values in ALL instructions in a set), it may not be desirable to introduce extra levels into the decode tree if a group of bound bits is divided by a small number of unbound bits. We therefore expand our selection criteria to allow a limited number of unbound bits to be selected. These unbound bits are assigned both of their possible values, and the associated instruction is replicated in the appropriate subsets. This implies that an instruction with N unbound bits for some particular selection range will be replicated 2^N times in resulting subsets. This represents, therefore, a potential trade-off between decoder depth and decoder size. While size increases due to replication can be mitigated somewhat by the tree compression algorithm mentioned previously, we would still expect a greater number of unbound bits to be associated with more overall state in the final decoder.

To motivate the use of unbound bits in decoder generation, we begin with a simple heuristic to evaluate the usefulness of such bit expansions. We begin developing our bit selection heuristic with the observation that the depth of decode tree required by any set of instructions is likely to be related to the number of instructions in the set and that decode tree depth relates directly to decoder performance. Obviously there are many other factors that affect decoder depth and performance (such as the number of bound bits), but as a first order approximation this relationship makes intuitive sense. Thus, for any given set of instructions, we evaluate all valid bit sequences and select the sequence that minimizes the size of the largest resulting subset. This approach intuitively favors bit sequences that spread the instructions out among the result subsets, minimizing the expected depth of the decoder at each stage. We evaluate the usefulness of expanding unbound bits by setting restrictions on the number of such unbound bit positions that are allowed in any valid bit sequence. Figure 6.1.1 shows results for the cycle count metric for a number of benchmark programs across a range of maximum allowed unbound bits. Relaxing the unbound bit constraint beyond five bits for ARM and seven bits for IA32 had no effect on generated decoders, indicating the effectiveness of the bit selection heuristic is eliminating arbitrarily poor decoder structures. The figures show a clear performance benefit to expansion of unbound bit locations during decoder tree generation. While the IA32 results show a steady progression of performance, the ARM results show a sudden performance increase at 4 unbound bits. Though the precise reason for this 'sweet spot' is as yet undetermined, we observe that in both decoders the best performance was gained when the unbound bit constraint approximately matched the size of the primary opcode field for the instruction set. In the ARM instruction set, the primary opcode for most instructions, as well as the register specifier size, is four bits. In the IA32 instruction set, the primary opcode is generally considered to be 8 bits, however in most instructions one or two of these bits actually represent modifiers on instruction semantics, rather than separate instruction groups.

It is important to note that decoders produced by Rosetta were able to achieve performance comparable to or better than decoders from SimpleScalar. This clearly shows that the Rosetta toolset is able to construct decoders that perform as well as hand-coded decoders with much less effort on the part of the programmer. It also demonstrates that unbound bit

Figure 6.1.1: Cycle Count Results





expansion is an effective method for increasing decoder performance.

6.2 - Node Ordering and Cache Performance

We next use the decode trees generated with our bit selection heuristic to evaluate the usefulness of re-ordering the memory state of decoders to improve cache performance. The table based decoder emitted by the toolset allow memory locations of various decoder nodes to be arbitrarily re-arranged (though the internal state of an individual node cannot be similarly re-arranged). We recognize that for ordering to be useful, nodes must be accessed frequently during program execution. A preliminary profiling run indicated, however, that the actual working set of these decoders was very small. In fact, the second most frequently accessed node in the generated ARM decoders (out of over 400 nodes) was referenced on fewer than 10% of all instructions. The IA32 decoders showed similar access profiles, though the working set was slightly larger (third most frequently accessed node referenced on less than 15% of instructions). These working set profiles indicate that node reordering would actually not be very useful in this domain, as very few nodes are accessed to decode most instructions. A brief set of evaluations using the Cheetah cache simulation library showed that node ordering is indeed ineffective at increasing performance. The small working set demonstrated in the profiling statistics, however, does serve as further validation that the bit selection heuristic is quite effective for generating good decoder trees that resolve instructions at minimal depth.

6.3 - Decoder Type

As a final evaluation of our basic node selection heuristic, we wish to consider the effect of decoder type on performance. Specifically, we compare the performance of the table based decoders discussed above, and switch statement based decoders commonly constructed by programmers. By working from the common internal representation provided by the toolset, it was possible to construct these decoders from identical decode trees, thus isolating the performance differences between the two forms.

It was expected that latency due to indirect branches and poor cache performance would seriously hamper the performance of switch statement based decoders. The result of our trials, however, show that switch statement based decoders *consistently outperform* table based decoders (by approximately 100 cycles / decode). A more detailed evaluation of decoder structure provides an explanation for the performance gap. Firstly, the profiling information from the previous section indicates that very few indirect branches are executed by the decoder before a result is available, mitigating their effect on performance. We believe, however, that the major performance win comes from optimizations that can easily be incorporated into the switch statement format and cannot be incorporated into a table based format. For example, all information regarding bit selection (first bit position, selection mask, etc.) involves a memory access for table based decoders. In the switch statement format, much of this information can be incorporated directly into the code as immediate values, thus pushing most of the work of bit selection on the instruction fetch unit and eliminating the data accesses altogether. In a similar fashion, several other memory accesses and conditional tests were unnecessary in the switch statement format. Due to the otherwise isomorphic structures of the evaluated decoders, it is reasonable to conclude that these optimizations lead directly to the observed performance. We must note, however, that despite the better performance, switch statement based decoders tended to be 20-30% larger than identical table based decoders. Thus, the performance increase comes at the cost of total memory footprint.

6.4 - Bit Selection Heuristic Revisited

Given these results, we would like to evaluate the use of unbound bits in more detail. Our previous heuristic considered a bit position unbound if it resolved to a “don’t care” position in any instruction in the set. This is a fairly coarse metric, as unbound bits may actually be bound in most of the instructions in the set. In fact, a more detailed evaluation of bit selection shows that this is by far the common case near the head of the decode tree. Instruction sets tend to group together semantically relevant bits (e.g., opcode fields), leading to naturally occurring contiguous blocks of mostly bound bit positions with occasional unbound bits occurring in those few instructions with odd syntax. Thus, we wish to adopt a cost function that does not place hard limits on ‘unbound’ bit positions from a global perspective, but rather considers the effect of unbound bit positions on replication in the resulting decoder tree. The results from section 6.1 seem to indicate that minimizing the largest resulting subset is an effective way of selecting bit ranges. We thus wish to incorporate this metric with a better evaluation of unbound bit usage. The final selection heuristic is effectively a weighted sum of the largest subset size and the total number of instructions in all subsets. The first value (largest subset size) is obviously in the range [1, #instructions]. This value is normalized to the total number of instructions, resulting in a ‘worst case’ value of 1.0. The second value is a number in the range [#inst, N*#inst], where N is the number of subsets. Thus, if no unbound bits are used, the sum of instructions in all subsets is equivalent to the number of instructions that must be differentiated. If only completely unbound bits are used (a worst case scenario) all instructions are replicated in all subsets, thus the sum is N times the number of instructions. This value is normalized to the number of instructions times the number of subsets, resulting once again in a worst case value of 1.0. Thus, if both portions of the metric are unweighted, the final cost metric ranges from a minimum dependent on the number of instructions up to a maximum (worst case) value of 2.0. By once again evaluating all reasonable groups of contiguous bits, we acquire a list of potential selection ranges, each with an associated cost. With this approach, and selecting the lowest cost bit range, we consider the effect of unbound bits by measuring the replication they cause, rather than placing an arbitrary constraint upon them.

This new evaluation also allows us to favor portions of the cost metric by weighting them, rather than explicitly selecting one metric over the other. We would expect that increasing the weight of the largest set size portion of the metric would tend to lead to larger, faster decoders (as the system would favor selecting larger bit sequences over reducing replication). Similarly, we would expect increasing the weight of the replication metric to lead to smaller (though potentially slower) decoders. Decoders generated under various weight schemes do not, however, bear out this hypothesis. ARM decoders generated to reduce replication did show a substantial decrease is size, but there was little difference between decoders favoring the maximum set size and decoders for which both portions of the cost metric were weighted equally. Decoders for IA32 completely contradicted the expected pattern, becoming slightly larger as more relative weight was placed on replication reduction. We believe these surprising results are due to the effect of our tree compression algorithm, which depends on similarity between subtree to reduce state. This algorith affects the relationship between the bit selection heuristic and final decoder size in fairly non-trivial ways, making the results difficult to predict.

We compare decoders generated with this new heuristic against the better performing decoders generated previously. Results showed that decoders were comparable in both data size and raw performance. Decoders generated for the ARM instructions set were ten to fifteen cycles slower and 15-40% larger than the best performing decoders from the previous results (depending on cost function weighting). These results reflect the inherent intricacies in the ARM instruction set, making it more difficult to perform automatic decoder generation. By contrast, the IA32 decoder were approximately 4% smaller (approximately 123k), and performed slightly better than the previous batch. In both cases, altering the weights on the cost function had very little impact on decoder performance, only on decoder size.

6.5 - Non-contiguous Bit Sequence Selection

The observation that switch statement based decoders provide better performance than table based decoders and can be optimized more completely allows us to consider incorporating non-contiguous bit sequences into the decoder tree. We hope non-contiguous bit sequence selection can be used to decrease the overall size of the decoder tree by eliminating extra nodes. We also hope that careful selection of such sequences can improve decoder performance, as the effect of non-contiguous sequences is to allow multiple selections without the overhead of passing to the next level of the decode tree. Switch statement based decoders are vital for such an analysis, as the need to accommodate the extra state information in a table based decoder would make such a consideration prohibitive.

Our evaluation of non-contiguous sequences develops directly from the new cost function described previously. Each viable selection range is evaluated and the lowest cost range is selected (as with contiguous range selection). We then selectively join other ranges, beginning with the lowest remaining cost, until a preset cost limit or until the maximum number of bits allowed in a single step is reached. Thus, we attempt to mitigate the cost of incorporating non-contiguous sequences by selecting sequences that best differentiate the set. By this approach, we are also able to evaluate the effect of allowing progressively less useful (higher cost) non-contiguous sequences by increasing the cost limit.

Our approach to non-contiguous bit sequences turns out not to reduce decoder size. Though it reduces the number of actual decoder states (as expected), the increased size of each of these states (and potential effect on tree compaction) turn out to increase decoder size slightly as the number of non-contiguous sequences increases. While disappointing, this is not an unreasonable result, as increased non-contiguity is achieved by utilizing increasingly high-cost bit range selections. While the 'expected tree depth' of these higher cost ranges is capped by the lower cost ranges that are also included, the effects of instruction replication are not, thus the larger decoders.

Varying the maximum allowed cost for non-contiguous bit sequences had no apparent affect on ARM decoder generation. Though several low cost sequences were available, these tended to be subsets of the lowest cost sequence and were thus subsumed by it. We also know from decoder profiling that the working set size for the ARM decode tree is extremely small. No relaxation of the maximum cost constraint caused non-contiguity at the head of the decode tree, the level at which over 90% of instruction references resolve.

Variation in the cost constraint does turn out to have an impact on the generated IA32 decoder. The results of various cost constraints are shown in Figure 6.5.1. These results consider performance variations under various weights for the cost metric. We observe that while all three weight schemes have similar performance for contiguous bit selection (cost = 0.0), they diverge somewhat as increasingly high cost non-contiguous sequences are allowed. Our initial expectations were to see an increase in performance for increased non-contiguity up to a point, after which the performance would decrease. This 'knee' would correspond to the point where the detrimental effects of incorporating higher cost sequences outweighs the benefit of covering a greater number of bits at each stage of the decode process. In the observed results, only decoders weighted to minimize replication showed the expected trend. Decoders generated with equal weight suffer from an initial decrease in performance, but then seem to fall into the expected pattern, while decoders weighted to focus on minimizing largest subset size do not appear to follow the pattern at all. Since decoder performance comes down to a subtle balance between the cost of non-contiguous bit selection and the benefit of reducing potential tree depth, it is difficult to attribute these variations to any specific cause.

Figure 6.5.1: Cycle Count Results for Various Non-Contiguous Bit Selection Constraints (IA32 ISA)

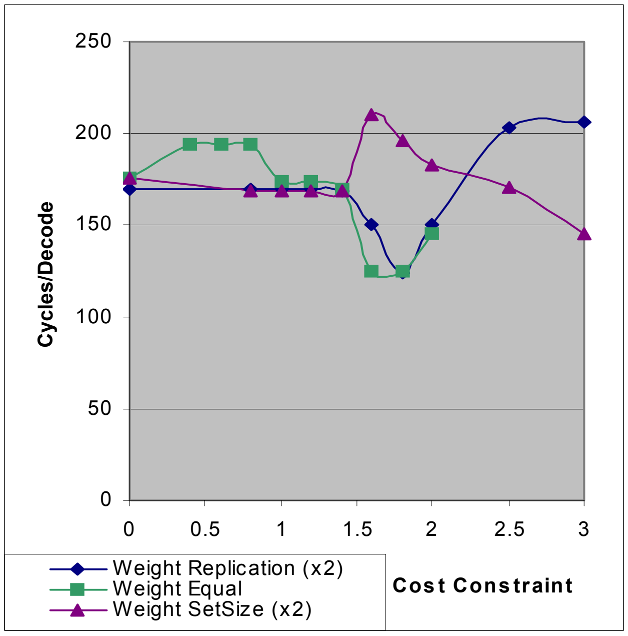


Table 1: Summary Comparison of SimpleScalar and Rosetta Decoders (IA32 ISA)

Decoder	Perf.	WS (I)	Miss (I)	WS (D)	Miss (D)	Footprint
SimpleScalar	310	5792	0.0148	1424	0.0128	~70k
Rosetta	170	5792	0.0165	1408	0.0158	~124k

Key:
Perf: Decoder performance (average cycles/decode)
WS (I/D): Working set size in (i/d)-cache (bytes)
Miss (I/D): Cache miss ratio for cache just larger (< 16 bytes larger) that WS(I/D)
Footprint: Total size of all decoder related data (bytes)

6.6 - Profiled Reordering Revisited

As a final performance tuning metric, we attempt once again to profile decoder traversals, this time re-ordering the case blocks of various switch statements in order of use. While this does not affect the data cache performance of the compiler generated switch jump tables, it should increase locality in the i-cache, making the common case path through the decoder tree an in-order execution of instructions. We are motivated by the same profiling data that showed the ineffectiveness of data reordering in phase one. In this domain however, we would expect reordering to benefit from the small working set and relative consistency of data flow. Results show a slight performance improvement (~10 cycles on average) corresponding to a consistent, but extremely small decrease in cache misses. Thus, while we do at least see a consistent effect, we once again do not feel that this represents an area to focus optimization efforts in this domain.

Section 7: Summary and Conclusions

We present a specification language for describing the syntax of instruction set architectures, and the beginnings of a language for describing complete instruction set syntax and semantics. We use this specification language to produce decoders for ARM7 (a RISC ISA) and IA32 (a CISC ISA). We attempt to optimize these decoders through inclusion of unbound bits during decoder generation, decoder state compaction, and decoder state reordering through cache conscious data placement techniques. Our first round of evaluations demonstrated that allowing expansion of bit positions that are not globally bound during decoder generation can lead to substantially faster decoders than more restrictive approaches, and that state compaction can be applied quite effectively in this domain. Our results also indicate that, due to very small working set sizes, decoder state re-ordering for memory placement does not have a noticeable affect on cache or raw performance metrics. One particularly interesting result indicates that switch statement based decode functions, despite potential delays due to indirect branching, can often be optimized to run faster than table based decoders.

Based on these results, we perform a more detailed evaluation of switch statement based decoders, using a new cost function to better automate decoder generation and exploiting the hardcoded structure of switch-statement based decoders to evaluate the use of non-contiguous bit sequences. We find that making allowances for carefully constrained non-contiguous sequence selection can have a beneficial impact on decoder performance. Finally, we evaluate re-ordering of switch statements to improve i-cache performance. We find that, while case reordering does have an impact on performance (unlike data reordering), the effect is insignificant.

Table 1 shows a metric by metric comparison between the SimpleScalar IA32 decoder (table based, one of our baselines) and some average values for a decoder generated from the Rosetta toolset (specifically, a switch statement decoder selecting only contiguous bit sequences). This brief summary clearly demonstrates that we are able to generate decoders that perform comparably against a carefully hand-coded decoder. By generating decoders from a straightforward description language, we are able to reduce the time and effort required for coding and debugging, allowing us to quickly explore a wide range of optimizations and trade-offs.

The next step in this research involves expanding the specification language to include instruction semantics. The primary challenge here is in describing widely disparate architectures in a canonical form. We hope eventually to be able to apply this specification to a range of useful problems such as compiler or software re-targeting, simulator and test infrastructure generation, and high speed code translation.

References:

[1] Java Language Specification, http://java.sun.com, 2001.
[2] Crusoe Processor, http://www.transmeta.com/crusoe/, 2001.
[3] D.Vengroff. decgen - A Decoder Generator. http://www.ece.udel.edu/~vengroff/asl/tools/decgen/decgen.htm, 1996.
[4] N. Ramsey and M.Fernandez. The New Jersey Machine-Code Toolkit. *Usenix Technical Conference.* New Orleans, LA, 1995.
[5] N. Ramsey and M. Fernandez. Specifying Representations of Machine Instructions. *ACM Transactions on Program ming Languages and Systems.* May 1997.
[6] S.Onder and R.Gupta. Automatic Generation of Microarchitecture Simulators. *Proceedings of International Conference on Computer Languages. 1998.*
[7] D.C. Burger and T.M. Austin. The SimpleScalar Tool Set, V. 2.0. Technical Report 97-1342, Computer Science Depar ment, Univ. of Wisconsin Madison, 1997.
[8] B.Calder, C.Krintz, S.John and T.Austin. Cache-Conscious Data Placement. *IS-ASPLOS.* San Jose, CA. October 1998.
[9] T. Chilimbi, M.Hill, and J.Larus. Cache-Conscious Struc ture Layout. *Proceedings of ACM SIGPLAN.* May 1999.
[10]T. Ball and J. Larus. Efficient Path Profiling. *Proceedings of Micro-29.* Paris, France. December 1996.
[11]A.Aho, R.Sethi, J.Ullman. Compilers: Principles, Tech niques, and Tools, Addison-Wesley, 1988.
[12]R.Sugumar and S.Abraham. Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Char acterization. *Proceeding of ACM SIGMETRICS.* May 1993.