HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems

John Regehr Alastair Reid
School of Computing, University of Utah

ABSTRACT

Embedded software must meet conflicting requirements such as being highly reliable, running on resource-constrained platforms, and being developed rapidly. Static program analysis can help meet all of these goals. People developing analyzers for embedded object code face a difficult problem: writing an abstract version of each instruction in the target architecture(s). This is currently done by hand, resulting in abstract operations that are both buggy and imprecise. We have developed Hoist: a novel system that solves these problems by automatically constructing abstract operations using a microprocessor (or simulator) as its own specification. With almost no input from a human, Hoist generates a collection of C functions that are ready to be linked into an abstract interpreter. We demonstrate that Hoist generates abstract operations that are correct, having been extensively tested, sufficiently fast, and substantially more precise than manually written abstract operations. Hoist is currently limited to eight-bit machines due to costs exponential in the word size of the target architecture. It is essential to be able to analyze software running on these small processors: they are important and ubiquitous, with many embedded and safety-critical systems being based on them.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.2.4 [**Software/Program Verification**]: Miscellaneous

General Terms

Reliability, languages, verification

Keywords

Abstract interpretation, static analysis, program verification, object code

ASPLOS'04, October 7–13, 2004, Boston, Massachusetts, USA.

1. INTRODUCTION

Static program analysis is a broadly useful technology that can help developers create embedded software that meets the conflicting goals of high reliability and minimal resource usage. In particular, dataflow analysis by abstract interpretation [10] is an important analysis technique that is the basis for a wide variety of validation and optimization algorithms. For example, *constant propagation*, a simple analysis that discovers parts of a program's execution during which the value of a storage location does not change, has been used to support sophisticated analyses such as bounding worst-case execution time [27]. In this paper we are concerned with the analysis of object code, which is useful for discovering low-level properties of programs or analyzing systems where source code is unavailable.

An abstract interpreter executes a program symbolically by manipulating abstract values. For example, constant propagation requires two kinds of abstract values: those that represent constants (e.g., "4") and one that represents values that cannot be proven to be constant, denoted $\perp$. Operations on these values are easy to understand and implement: values representing constants are manipulated just as they would be on a regular CPU, and $\perp$ is contagious—any operation with a non-constant input produces $\perp$ as a result. The problem with constant propagation is that it is a fairly weak analysis; when control-flow paths are merged it quickly loses information due to conservative approximation.

The *interval* and *bitwise* domains support stronger analyses that are more resistant to losing information. The interval domain finds cases where the value of a storage location can be shown to lie inside a sub-interval, e.g., $[2..5]$, of the natively supported range of values. The bitwise domain is a ternary logic that models each bit as having either a known or unknown value. For example, a 4-bit register might hold the bitwise value $01\perp\perp$, indicating that its top two bits contain constant values zero and one, while its bottom two bits cannot be proven to be constant. These two domains are more powerful than the constant propagation domain—in fact, they both subsume it—and they have been used to analyze and optimize many aspects of embedded software (some are described in Section 2). The problem with implementing abstract interpreters based on these more sophisticated domains is that it is hard to implement the abstract operations. Each abstract value corresponds to some set of concrete values, and our experience is that it is difficult for programmers to reason about these sets. Each abstract operation should be correct (returning a safe estimate of the processor's state), precise (losing as little information as possible), and as efficient as possible. Particularly tedious tasks are implementing abstract effects on a processor's condition codes and creating abstract versions of operations that do not match the domain, such as a bitwise "add" or an interval "xor."

This paper presents *Hoist*, a system that automates the implementation of abstract machine-level ALU operations for the interval and bitwise domains. Hoist reduces developer effort to almost nothing; it requires only a small amount of metadata about each instruction—all other information is extracted from a microprocessor or simulator that supports the target instruction set. The second main advantage of Hoist is that it produces abstract operations that are trustworthy, having been subjected to an extensive battery of tests. Third, Hoisted operations lose as little information as possible to conservative approximation; they are maximally precise within the constraints of each domain. Hoist's fourth advantage is that it makes no assumptions about the function implemented by an instruction; it works for arbitrary ALU operations. A limitation of Hoist, on the other hand, is that it only generates code for arithmetic and logical operations. We do not consider instructions that manipulate specific hardware, e.g., those that reset the watchdog timer or put the processor to sleep, or those that implement control flow.

Abstract operations are necessary to support analyses, but they are difficult to implement. As a basis for comparison with Hoist, implementing the bitwise abstract interpreter in our stacktool [32] took several weeks, followed by several months of sporadic debugging and refinement before it became really useful. Furthermore, systematic testing of the hand-written abstract operations using Hoisted abstract operations as a reference turned up more than half a dozen subtle errors.

Our work contrasts with other frameworks for easing the creation of object code analyzers such as the one by Fritz et al. [16]. They focus on generating the body of an abstract interpreter given a high-level description of a processor architecture, but leave the implementation of low-level transfer functions to the developer. Our work is almost perfectly complementary; it does not facilitate porting the body of the analyzer, but rather focuses on the derivation of the abstract operations.

Hoist treats instructions as black boxes: they can be arbitrary functions from bit-vectors to bit-vectors. This brute-force approach, using no high-level symbolic representations, keeps developer effort low and exploits the large asymmetry that exists between the capabilities of desktop processors and small embedded chips. Even so, at present Hoist only supports architectures with word sizes of eight or fewer bits due to costs exponential in the word size. These small processors represent an important domain: they run safety-critical applications and are ubiquitous. For example, out of the 71 microprocessors onboard a BMW 745i, 53 are 8-bit chips [23]. Programs running on small microprocessors are an ideal target for static analysis since they are typically deployed in large numbers in hard-to-reach areas (making bugs costly), they are highly resource constrained (making resource-bounding analyses desirable), and they display relatively few of the dynamic behaviors like recursion and heap allocation that complicate static analysis [14]. We evaluate the effectiveness of our work using sensor network nodes: a domain where 8-bit chips are commonly used because they are inexpensive and energy-efficient.

The remainder of this paper is organized as follows. Section 2 provides background on abstract interpretation and the two domains we consider. In Section 3 we present the key motivation for our work: the difficulty of implementing precise abstract operations manually. Section 4 describes Hoist in detail and Section 5 evaluates its performance both at code generation time and at run time. In Section 6 we discuss related work and in Section 7 we discuss future work and scaling issues. Section 8 presents our conclusions.

2. BACKGROUND

This section motivates our work and briefly reviews abstract interpretation and the standard analysis domains used in this paper.

2.1 Analyzing object code

Static analysis and transformation of object code has been used to analyze worst-case execution time [27], show type safety [39], insert dynamic safety checks [38], obfuscate programs [26], optimize code generated by a compiler [19], analyze worst-case stack depth [5, 32], validate compiler output [34], find viruses [6], and decompile programs [7]. A few of the many specific reasons for analyzing object code are that:

- Source code is often unavailable: it may be proprietary, may have been lost, or may never have existed, for systems written in assembly.
- Low-level properties such as stack usage and execution time cannot be effectively analyzed at the source level because compilers have considerable flexibility while performing the translation.
- Embedded systems commonly interleave C or C++ with assembly language. The semantics of mixed-language code may be unclear, making it difficult to analyze.
- Specialized embedded processors often provide architectural features that can speed up applications. It may be more cost-effective to use post-pass optimization to rewrite binaries to take advantage of these features rather than modifying a complex compiler.

2.2 Abstract interpretation

Abstract interpretation [10] is a framework for static program analysis. By manipulating abstract values, which represent sets of concrete values, an abstract interpreter can compute the properties of many program executions using relatively few analysis steps. For example, rather than separately analyzing the behavior of an embedded system for each of the possible values returned by a temperature sensor, an abstract interpreter would simply analyze the case where the sensor returns a single abstract value representing the set of all possible temperature inputs. Abstract interpretations deliberately make approximations to avoid undecidability and to achieve reasonable space and time efficiency.

Abstract domains. Abstract values are modeled using *domains*, or partially ordered lattices of abstract values, where each abstract value x corresponds to a unique set of concrete values $\gamma(x)$ and, conversely, each set of concrete values Y corresponds to a (usually not unique) abstract value $\alpha(Y)$. The smallest element of the lattice $\perp$ represents a complete lack of information; its concretization is the set of all possible values. The partial order of the lattice is defined by:

$$x \sqsubseteq y \stackrel{\text{def}}{=} \gamma(x) \supseteq \gamma(y)$$

In other words, smaller abstract values represent larger sets of concrete values, and are consequently less precise estimations of the contents of a storage location. The greatest lower bound operation $\sqcap$ is the largest (most precise) abstract value such that:

$$\gamma(x \sqcap y) \supseteq \gamma(x) \cup \gamma(y)$$

In other words, the concretization of the greatest lower bound of two values must be a superset of the union of the concretization of the individual values. We refer to $\sqcap$ as the *merge* operator for two abstract values; it is used to create a safe and precise estimate of the state of the machine when two control-flow paths are merged, for example after analyzing both branches of an if-then-else construct.

In this paper an "imprecise" result is technically correct but has lost some information. It is important to distinguish between the different kinds of imprecision that occur during abstract interpretation. First, an abstract value may be imprecise because the abstract domain cannot represent a given concrete set. For example, consider a program where a certain storage location contains only the values "4" and "6" in all executions. A constant propagation analysis must conclude that the storage location contains the value $\perp$, because this domain is inherently not expressive enough to return a more precise result. Second, an abstract value may be imprecise because of approximations made in the implementation of an abstract interpreter. For example, assume that two different interval domain analyzers for the program above respectively estimate the storage location to contain $[4..6]$ and $[0..6]$. Both results are correct but only $[4..6]$ is maximally precise, given the constraints of the interval domain. This paper is about avoiding this second kind of imprecision. All other things being equal a more precise analysis is preferable, but usually precision is gained at the expense of memory and CPU consumption at analysis time.

Abstract operations. The focus of this paper is on deriving an abstract operation g for every machine-level logical and arithmetic operation f that is provided by a given processor architecture. An abstract operation must satisfy:

$$\gamma(g(x)) \supseteq \{f(y) \mid y \in \gamma(x)\}$$

To understand this equation, consider an abstract value x. We can apply the concrete function f to each member of the set of concrete values represented by x. The set of concrete values so obtained must be a subset of the concretization of the abstract value returned by applying the abstract function g to x.

The trivial abstract function, which always returns $\perp$, is correct—but useless. The challenge, then, is to obtain a more precise abstract function. In this paper, we use $f^\sharp$ to denote the most precise abstract version of a concrete function f for a given domain. The maximally precise abstract function can be computed by:

$$f^\sharp(a) \stackrel{\text{def}}{=} \alpha(f(c_1)) \sqcap \ldots \sqcap \alpha(f(c_m)) \;\; \text{if } \gamma(a) = \{c_1, \ldots, c_m\}$$

That is: concretize the abstract value that is the argument to the function, apply the corresponding concrete function to each concrete value, and then merge the concrete results together to form an abstract value. The maximal precision follows from the fact that we are merging together only values that need to be merged, and the facts that $\sqcap$ is commutative, associative, and maximally precise.

The straightforward implementation of $f^\sharp(a)$ is far too inefficient to use in a program analysis tool, even for 8-bit architectures. On the other hand, the main contribution of Hoist is to derive a fast, small implementation of the most precise abstract function for two different domains.

2.3 Bitwise domain

The bitwise abstract domain is a ternary logic in which each bit either has a concrete value or is unknown. Formally, each bit has a value from the set $\{0, 1, \perp\}$ and the concretization function is:

$$\begin{array}{rcl} \gamma(0) & = & \{0\} \\ \gamma(1) & = & \{1\} \\ \gamma(\perp) & = & \{0,1\} \end{array}$$

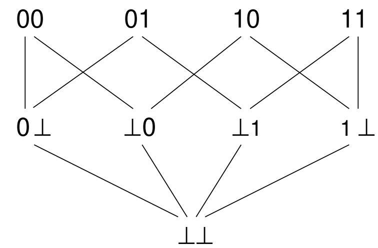

Figure 1: The bitwise lattice for $N = 2$. In the general case this lattice has height $N + 1$ and contains 3^N elements.

Consequently, the merge function for bits is:

$$a \sqcap_{bit} b \stackrel{\text{def}}{=} \begin{cases} a & \text{if} \quad a = b \\ \perp & \text{if} \quad a \neq b \end{cases}$$

An abstract word is the composition of multiple bits. We use N to denote the number of bits in a native machine word. The merge function for words is simply:

$$a \sqcap_{bits} b \stackrel{\text{def}}{=} (a_0 \sqcap_{bit} b_0, a_1 \sqcap_{bit} b_1, \ldots, a_{N-1} \sqcap_{bit} b_{N-1})$$

The bitwise lattice, shown for two bits in Figure 1, is useful for reasoning about partially unknown data that is manipulated using bitmasks and other logical operations. For example, in our previous work on bounding the stack memory consumption of embedded software in the presence of interrupts [32], the bitwise domain was crucial for estimating the status of the interrupt mask at each program point. The first use of the bitwise domain for analyzing software that we are aware of is Razdan and Smith's 1994 paper [31].

2.4 Interval domain

The interval domain exploits the fact that although many storage locations contain values that change at run time, it is often the case that these values can be proven to occupy a sub-interval of the entire range of values that can be stored in a machine word. For example, it might be expected that a variable used to index an array of size i would never have a value outside the interval $[0..i-1]$. In the interval domain, abstract values are tuples [low, high] where low $\leq$ high. The concretization function is:

$$\gamma([\text{low}, \text{high}]) \stackrel{\text{def}}{=} \{\text{low}, \text{low} + 1, \ldots, \text{high}\}$$

Two intervals can be merged as follows:

$$[a_l, a_h] \sqcap_{int} [b_l, b_h] \stackrel{\text{def}}{=} [\min(a_l, b_l), \max(a_h, b_h)]$$

The interval lattice, shown for two-bit unsigned integers in Figure 2, is best used to model arithmetic operations. It has been used as part of a strategy for eliminating array bounds checks [17], for bounding worst-case execution time [15], and for synthesizing optimized hardware by statically showing that the high-order bits of certain variables were constant at run time [36]. The interval domain appears to have been introduced by Cousot and Cousot [10].

3. IMPLEMENTING ABSTRACT OPERATIONS MANUALLY

Consider an `add` instruction that has the following assembly language representation:

```
add dst, src
```

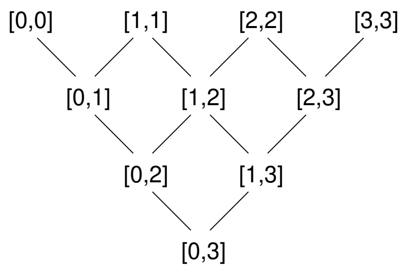

Figure 2: The unsigned interval lattice for $N = 2$. In the general case this lattice has height 2^N and contains $2^{N-1}(2^N + 1)$ elements.

In a CPU simulator this might be implemented as:

```
reg[dst] = (reg[src]+reg[dst]) % (MAXUINT+1);
```

in addition to some code updating the condition code flags. We assume that the processor running the simulator has a larger word size than the processor being emulated, and hence `MAXUINT+1` does not overflow.

An abstract `add` in the unsigned interval domain is complicated slightly by the case analysis necessitated by the potential for the low and high ends of the result interval to independently wrap around:

```
lo = reg[src].lo + reg[dst].lo;
hi = reg[src].hi + reg[dst].hi;
if ((lo > MAXUINT) ^ (hi > MAXUINT)) {
  reg[dst].lo = 0;
  reg[dst].hi = MAXUINT;
} else {
  reg[dst].lo = lo % (MAXUINT+1);
  reg[dst].hi = hi % (MAXUINT+1);
}
```

Computing the abstract condition codes is also more difficult than it is in the concrete case, and both result and condition codes are even more painful for the signed interval domain where there are more ways to wrap around.

A bitwise abstract `add` is much trickier, and it is in situations such as this where implementors often resort to crude approximations. For example, a first cut at the bitwise `add` might return an entirely unknown result if any bit in any input is unknown. A better approximation is to return a result with m known bits if the bottom m bits of both arguments are known. For example, if bits 0–3 in both arguments are known, then the `add` functions normally in this range and returns $\perp$ for bits 4 and higher. On the other hand, if bits in position 4 are the only unknown bits in the inputs, and if the bits in position 5 in both inputs contain zeros, then any possible carry out of position 4 will be absorbed, and the `add` can function normally again in bits 6 through $N - 1$. Further improvements along these lines are possible but unattractive—the general case where known and unknown bits are freely mixed is difficult to reason about, as is the analogous case of computing the `xor` operation precisely for arbitrary interval values. In practice, developing a sufficiently good approximation for each machine-level operation is a laborious and error-prone process requiring refinement of approximations when the analysis returns imprecise results. This difficulty was the direct motivation for Hoist.

The necessity of precise abstract operations. One might be tempted to believe that examples like the precise bitwise `add` above are irrelevant because program variables tend to be manipulated either arithmetically or logically, but not both. This is far from being the case: compilers and skilled assembly language programmers take advantage of many low-level idioms and quirks in order to create compact and efficient code, and a successful analysis of object code must take this into account. For example, in machine code it is common to multiply using a combination of shifting and adding, clear a register using exclusive-or or subtract, move a flag bit from a register into a condition code using add-with-carry, and perform modular arithmetic using bitmasks. An additional benefit of using Hoist is that, since the abstract operations have been thoroughly tested and are maximally precise, it makes development of the remaining parts of an abstract interpreter easier by narrowing down the places where problems are likely to be found.

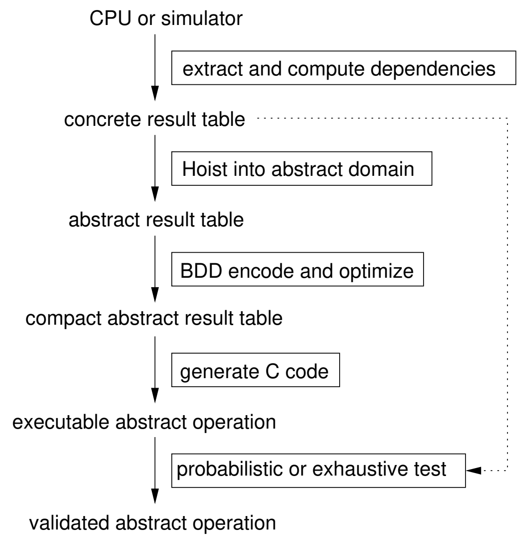

Figure 3: The Hoist toolchain for deriving abstract operations

4. DERIVING ABSTRACT OPERATIONS AUTOMATICALLY

Hoist automates the derivation and efficient encoding of maximally precise abstract machine-level operations. Figure 3 shows the toolchain that accomplishes this. For each instruction we *extract* a concrete table of results (Section 4.1), *lift* the concrete table into each abstract domain (Section 4.2), compactly *encode* the abstract table as a binary decision diagram (BDD) (Section 4.3), *generate* C code implementing the BDD (Section 4.4), and *test* the abstract operation over a wide range of inputs (Section 4.5).

4.1 Extracting concrete result tables

The first stage of deriving an abstract operation is to exhaustively establish the behavior of the corresponding concrete operation. This entails figuring out what parts of the machine state the instruction reads and writes as well as finding the actual mapping from inputs to outputs.

Extracting results. The most straightforward way of extracting concrete results is to run a small assembly language program on the microprocessor under study. The program enumerates all input values, applies each instruction to those values, and writes the results to an output device. Since embedded processors are often slow and have limited I/O capabilities, we found it to be faster and easier to run the test program in a simulator for the processor. For example, using the 57.6 kbps serial interface on a Berkeley mote (which is typical of the degree of connectivity found on a small embedded system) it takes about an hour and twenty minutes to gather up the entire output for a single binary 8-bit operation. A simulator can produce the same data in a matter of minutes. Simulators are freely available for most embedded processors and can usually log writes to output ports to a file. All four simulators for the Atmel AVR architecture that we looked at provide this functionality. In this paper, we used two AVR simulators: atemu [1] and simulavr [35]. The other two simulators were heavily graphical, making it difficult to use them in a programmatic way. Simulavr is designed to support most members of the AVR family, while atemu focuses on accurate emulation of a collection of mica2 sensor network nodes [20].

Extracting result tables from multiple simulators provides a useful cross-check. In fact, comparing the result tables did find a subtle bug in one of the simulators: the `lsr` (logical shift right) instruction in atemu was sometimes incorrectly setting a condition code flag. This bug was confirmed by the atemu developers and fixed in a subsequent release.

Computing dependencies. It would be possible to require that developers specify the precise input and output fields for each instruction. For example, "the `sbc` instruction reads the carry flag, zero flag, and two input registers, and writes the carry, zero, negative, overflow, sign, and half-carry flags, in addition to an output register." We found it more convenient and less error prone to specify a superset of each instruction's dependencies and then to compute the exact dependencies automatically. This permits all AVR instructions that Hoist supports to be placed into four equivalence classes: nullary, unary, binary register, and binary immediate. Instructions in all four classes may read and write the processor status register, which contains the condition codes.

We define a bit of machine state to be an *input dependency* if changing the value of the bit affects the behavior of the instruction. Similarly, a bit is an *output dependency* if executing the instruction potentially changes the value of the bit. For every instruction, each bit of machine state is either an input dependency, an output dependency, both, or neither. It would have been perfectly possible to use BDDs (see Section 4.2) to compute these dependencies, but since BDD operations are relatively expensive we do this in a preprocessing step. In other words, we improve the performance of Hoist by only presenting BDDs with inputs that are known to actually be input dependencies, and only using them to generate outputs that are actually known to be output dependencies. Most mis-specifications of an instruction's dependencies will be caught early because the generated assembly language fragment will be syntactically incorrect.

Pseudo-unary operations. Extra analysis precision can often be obtained when both inputs to a binary instruction come from the same storage location. A good example is exclusive-or: obviously, for any concrete value x, $\mathrm{xor}(x, x) = 0$. However, the analogous abstract relation $\mathrm{xor}^\sharp(x, x) = 0$ is false! Consider, for example, computing $\mathrm{xor}([1, 10], [1, 10])$. The abstract operation loses the information that both of its arguments represent the same concrete value, forcing it to return a very imprecise result. This is harmful in practice because compilers and programmers make such heavy use of idioms like this that a static analysis of object code cannot succeed without recognizing them. On many architectures, including AVR, an exclusive-or instruction uses less code space than does loading an immediate zero value, and so xor is the preferred method for clearing a register.

$$
\begin{aligned}
f^\sharp(a) &= f_0^\sharp(a) \\
f_i^\sharp(a) &= \begin{cases} \alpha(f(a)) & \text{if } i = N; \\ f_{i+1}^\sharp(\mathrm{clr}_i(a)) \sqcap_{bits} f_{i+1}^\sharp(\mathrm{set}_i(a)) & \text{if } a_i = \bot; \\ f_{i+1}^\sharp(a) & \text{otherwise.} \end{cases}
\end{aligned}
$$

Figure 4: Hoisting a concrete unary function into the bitwise domain

To provide operators that take advantage of the knowledge that their inputs are the same, we define a pseudo-unary operation corresponding to each binary register operation; it can be safely invoked when analyzing an instruction that specifies the same physical register for both of its inputs. For example, corresponding to the binary operation `eor r3,r3`, we define a unary operation $\texttt{eor}_1\ \texttt{r3} \stackrel{\text{def}}{=}$ `eor r3,r3`. (By convention, the AVR exclusive-or instruction is called `eor`.) This yields several useful pseudo-unary operations such as $\texttt{eor}_1$ and $\texttt{sub}_1$ (zero a register), $\texttt{and}_1$ and $\texttt{or}_1$ (test a value, setting condition codes appropriately), and $\texttt{adc}_1$ (rotate left through carry), as well as a number of less compelling operations such as $\texttt{mov}_1$ (nop) and $\texttt{sbc}_1$ (set register to `0xff` if carry set—`sbc` is the AVR subtract-with-carry instruction). From the point of view of an abstract interpreter, it never hurts to use a pseudo-unary abstract operation when the source and destination registers are the same, and in many cases extra information can be gained.

4.2 Creating abstract result tables

The second step in our toolchain lifts a concrete result table into an abstract domain. The maximally precise abstract version of a concrete function can be computed directly from the definition given in Section 2. A naive implementation for an operation with two N-bit inputs can require $O(4^N)$ steps to compute a single abstract result, where an abstract result table contains on the order of 4^N results—prohibitively expensive even for an 8-bit architecture.

This section describes how to reduce these costs by using caches and dynamic programming techniques. For both domains our strategy depends on a recursive subdivision of abstract values using the following equality which follows from the definition of $f^\sharp$:

$$f^\sharp(a) = f^\sharp(b) \sqcap f^\sharp(c) \ \text{ if } \gamma(a) = \gamma(b) \cup \gamma(c)$$

This allows us to compute $f^\sharp(a)$ in terms of two simpler calculations $f^\sharp(b)$ and $f^\sharp(c)$. By choosing b and c carefully, we can cache intermediate results effectively.

4.2.1 Unary operations

Elements of the bitwise abstract domain can be subdivided using the following equality:

$$\gamma(a) = \gamma(\mathrm{set}_i(a)) \cup \gamma(\mathrm{clr}_i(a)) \text{ if } a_i = \bot$$

where the functions $\mathrm{set}_i(a)$ and $\mathrm{clr}_i(a)$ respectively return their input a with bit i set or cleared. This yields the recursive algorithm shown in Figure 4. It has the same complexity as a naive enumeration, but is readily amenable to dynamic programming techniques by caching the result of each recursive call to $f^\sharp$. Using a cache of 3^N abstract results, each entry in the abstract result table can be computed in $O(1)$ amortized time; building the entire table requires $O(3^N)$ time.

Elements of the interval domain can be subdivided using the following equality:

$$f^\sharp([lo, hi]) = f^\sharp([lo, m-1]) \sqcap f^\sharp([m, hi]) \ \text{ if } \ lo \le m \le hi$$

This yields the recursive algorithm illustrated in Figure 5. The al-

$$f^\sharp([lo,hi]) = \begin{cases} \alpha(f(lo)) & \text{if } lo = hi; \\ f^\sharp([lo,m-1]) \sqcap_{int} f^\sharp([m,hi]) & \text{otherwise.} \end{cases}$$
where
$j = \lfloor \log_2(hi - lo + 1) \rfloor$
$m = \lfloor \frac{hi}{2^j} \rfloor \cdot 2^j$

Figure 5: Hoisting a concrete unary function into the interval domain

gorithm chooses m to be of the form $a \cdot 2^j$, where a and j are chosen to maximize the size of j.

For a unary operation, the complexity of this recursive function is $O(2^N)$, but it allows an efficient implementation by caching the result of $f^\sharp$ over all intervals of the form:

$$[a \cdot 2^j, (a+1) \cdot 2^j - 1]$$
where
$$(a+1) \cdot 2^j \leq 2^N$$

A cache of this form has $2^{N+1} - 1$ entries. For example, for N=8, the cache has just 511 entries. Using this function, the worst case computation time per result is $O(N)$ and the entire table can be computed in $O(4^N)$ time. An alternative cache design would be to cache the result over all intervals, yielding a faster implementation—constant amortized time per result—but requiring much more space.

4.2.2 *Binary operations*

We have presented algorithms and caching techniques for unary operations, but the important cases to handle are binary operations for which a naive algorithm would require $O(16^N)$ time per entry and $O(64^N)$ to compute the entire table. To adapt these algorithms to the binary case, we recursively subdivide one argument at a time. When the first argument is reduced to either a cache lookup or a concrete operation, we invoke a second recursive function to subdivide the second argument. This doubles the runtime of the operations and squares the sizes of the caches. That is, an abstract bitwise operation requires a cache of size $O(9^N)$, $O(1)$ time per result, and $O(9^N)$ time for the entire table. An abstract interval operation requires a cache of size $O(4^N)$, requires $O(2N)$ time per entry, and $O(16^N)$ time for the entire table.

4.3 Encoding result tables using BDDs

The abstract result tables may be inconveniently large. Using a straightforward encoding, the results for a binary 8-bit instruction require 82 MB for the bitwise domain and 2 GB for the interval domain. Clearly a better encoding is necessary; for this purpose we use binary decision diagrams (BDDs). We used BuDDy, a BDD package from the IT-University of Copenhagen [25].

A binary decision diagram [4] represents a Boolean function as a decision graph. That is, a BDD is a directed acyclic graph with a single root node, each terminal node labeled with zero or one, and each nonterminal node labeled by a variable and having two outgoing edges corresponding to the cases where the variable evaluates to zero or to one. For any assignment of the variables, the function value is determined by tracing a path from the root to a terminal node following the appropriate branch from each node. A function with m input bits and n output bits can be represented by a vector of n BDDs, each with m variables.

An ordered binary decision diagram (OBDD) imposes an order on variables such that all paths from the root to a terminal encounter the variables in ascending order. OBDDs can represent many common functions compactly, though they require nodes exponential in the number of inputs to represent certain functions such as integer multiplication.

We used standard techniques to construct the BDDs for each function. Each value in the bitwise domain is represented by two bit-vectors in the BDD: one vector determines whether each bit is known or unknown, and the other determines the concrete value, if known. Similarly, intervals are represented by a pair of bit-vectors: one holds the lower bound and one holds the upper bound. The choice of variable ordering can have a significant impact on the size of an OBDD. After some experimentation, we settled on an ordering that interleaves bits from both bit-vectors so that bits i from both vectors are adjacent. For binary functions, the best results were obtained by interleaving bits from all four input vectors.

To construct a BDD representation of a function, we enumerate all inputs, compute the corresponding results, and add the appropriate elements to the BDD. The runtime of this step, and of Hoist as a whole, is dominated by the relatively slow BDD operations which we invoke a large number of times.

Optimizing the encoding. We found it useful to perform an optimization step after constructing each BDD. Concrete operations return a valid result for all possible inputs; this is not true for our encoding of the abstract operations. For example, one would never try to compute $f^\sharp([x,y])$ with $x > y$. Impossible bitwise values are those that have an unknown bit whose value field is set to one. The practical consequence of impossible values is that the abstract result table has many unused entries; this overconstrains the BDD which, by default, returns zero when presented with an impossible input. These zeros are not free: BDD nodes are required to produce them. The solution is to use Coudert and Madre's minimization procedure [9] to make the output of the BDD a "don't care" for impossible input values. This simplifies the BDD, reducing BDD sizes by 42–85%. The BDD outputs for don't-care inputs are not harmful because a correct abstract interpreter will never produce impossible abstract values as inputs to a function, and in any case impossible values can be detected using simple tests.

4.4 Converting BDDs into C

Although translating a BDD into C code computing the corresponding Boolean function is conceptually trivial (each node in the BDD becomes a test of an input variable), generating efficient code requires some care. We found little literature on this topic, probably because BDDs are usually used directly, to answer questions about a system, rather than indirectly, to produce code that will be used to answer questions about a system.

Our first observation was that there is considerable sharing between BDDs implementing the output bits for an abstract operation. Therefore, any subexpression of the overall BDD computation that is used more than once is assigned a name and placed in separate expression. Once-used expressions, on the other hand, are inlined into their referencing expression. Second, eagerly evaluating all intermediate expressions is inefficient: they should instead be evaluated on demand. Finally, we found that gcc 3.3.2 has problems compiling the generated code when aggressive optimizations are enabled: the optimizer displays pathological behavior and has not been observed to terminate in reasonable time. To achieve good results we had to turn off the more aggressive optimizations while compiling the generated C code.

4.5 Testing

Our experience in staring at the results of static analyses of object codes suggested that spotting errors in the result of an abstract function is hard and that spotting suboptimal results is virtually impossible. Partially unknown data is nonintuitive and the sheer number of abstract values is overwhelming. Consequently, we wrote several test harnesses while creating Hoist.

Low-level tests. On every run of our BDD construction program we check that all concrete inputs have the expected concrete output and that several million randomly generated abstract inputs have the expected abstract output, where the expected output is computed by brute-force enumeration. This approach caught many errors early in development and has greatly increased our confidence in the correctness and accuracy of the final results. It is also possible to test a generated abstract operation exhaustively, but in practice we seldom do so because exhaustive tests are slow and we have never observed one to find an error where the probabilistic test did not.

Higher-level tests. Out of a sense of professional paranoia we tested Hoist in three additional ways. The first test validates an abstract operation in addition to the glue code that makes it available to an actual program analyzer; this is useful because there is potential for errors in the glue, in the dependency-testing code described in Section 4.1, and in the dependencies specified by developers. The second test ensures that each abstract result returned by a Hoisted function is always at least as precise as the result returned by a corresponding hand-written abstract function. This was intended as a sanity check for Hoist, but instead of finding bugs in the automatically generated functions, it revealed about half a dozen subtle errors in our hand-written abstract condition code computations. Our final test is an end-to-end validation of an entire abstract interpreter. We analyze a test program and dump the analyzed state of the machine at each program point. Then, we run the same program in a CPU simulator that has been modified to ensure, after executing each instruction, that the current concrete machine state is a member of the set of machine states obtained by applying the concretization function to the abstract machine state at that address.

In summary, in Figure 3 we describe a tested abstract operation as "validated" because we have used several independent tests to check its behavior against ground-truth: an actual CPU or simulator for the target architecture. A bug anywhere in our toolchain, even in BDD code not developed by us, will be detected if it results in incorrect output.

5. RESULTS

This section evaluates Hoist by answering the following questions: How precise are the generated abstract operations? How long does it take to generate them? How large is the generated code? How fast do the operations run? All measurements were performed on a 1533 MHz AMD Athlon with 512 MB RAM, using gcc 3.3.2 and BuDDy 2.2.

Our target architecture is Atmel's AVR: a modern, compiler-friendly family of embedded processors that provides 32 8-bit registers and variable amounts of other storage. For example, the ATmega128 chips [2] found on the Berkeley mica2 motes [20] have 128 KB of flash memory for program storage and 4 KB of SRAM.

Our basis for comparison is the stacktool, a program analyzer that we previously developed [32]. The purpose of the stacktool is to estimate the worst-case stack memory usage of an embedded system, in order to avoid the possibility of memory corruption through stack overflow. A major component of stacktool is an abstract interpreter for AVR binaries, where each abstract operation had been implemented by hand and subsequently refined to obtain sufficiently accurate results across a broad range of programs. We were able to obtain tight bounds on the stack depth even when this depended on analyzing data flow through multiple registers and many instructions. All comparisons with stacktool are for the bitwise domain because stacktool does not support intervals.

	bitwise		interval	
	arith	logical	arith	logical
unary	0.25 s	0.30 s	1.5 s	1.8 s
binary	0.63 h	0.68 h	33.7 h	22.4 h

Figure 6: Time to build BDDs for a typical abstract function in each category

	bitwise		interval	
	arith	logical	arith	logical
unary	1.4	1.4	6.0	3.0
binary	14	4.9	103	32

Figure 7: Size of generated code in KB of x86 object code for a typical abstract function in each category

5.1 Microbenchmarks

Deriving abstract operations. Figure 6 shows the time taken for Hoist to construct typical 8-bit arithmetic and logical operations for each abstract domain. In general it takes much longer to construct a BDD for a binary operation, and considerably longer to construct a BDD for an operation over the interval domain than for an operation over the bitwise domain. The time taken to build a BDD representation of an abstract binary operation for the interval domain is large (approximately 1.5 days) but tolerable since each BDD only has to be built once to create an abstract interpreter for a new processor architecture. We discuss scaling issues in Section 7.

Size of generated code. Figure 7 shows the size, in kilobytes of x86 object code, of compiled BDDs for typical 8-bit arithmetic and logical operations for each abstract domain. The trend is that binary and interval operations are larger, but the size of even the largest is quite reasonable compared to explicitly storing 43 million abstract values. The total amount of code added to stacktool when using Hoist-generated bitwise versions of all 34 ALU instructions is 272 KB.

Precision of the derived operations. By construction (and extensive testing), we know that the abstract operations generated by Hoist are maximally precise for each domain. To quantify the benefits of this increased precision, we compared the Hoisted operations against operations that we factored out of the stacktool so they could be invoked separately. We define the "precision" of an operation as the average number of bits of information in the result of the operation when it is supplied with all possible abstract inputs. For the numbers reported here we approximate this value by testing many randomly generated abstract values. For the bitwise domain, the precision is the average number of known bits in the results. Figure 8 compares the precision of Hoisted operations against hand-written stacktool operations, measuring the effect on the result register and condition code register separately. Despite having written stacktool, we had not realized that stacktool almost always returns a completely undefined result register value for arithmetic operations. On inspection, we found that the result register was defined only if all bits of both inputs were defined. This only happens

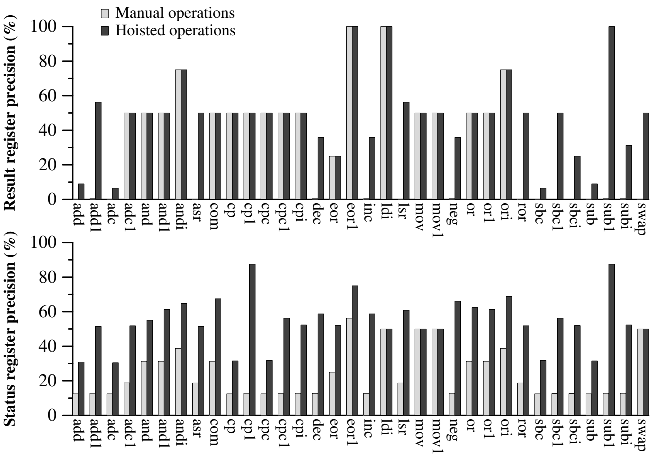

Figure 8: Precision of Hoisted and hand-written abstract operations for the bitwise domain. All figures are the average percentage of known bits in the outputs for 50 million random abstract inputs. Note that the hand-written operations lose almost all precision for the result register for arithmetic operations.

in a small part of the input space. The precision of arithmetic operations can easily be improved by returning an m-bit result if the bottom m-bits of both arguments are defined. This implementation improves the precision of most arithmetic operations to roughly 0.8 bits, which is still less than half as accurate as the Hoisted operations. Overall, Hoist increases the number of known bits in the result by 59% and in the condition codes by 130%.

To provide some indication of the precision of the generated interval operations, we wrote our own interval operations. For example, our hand-written interval "add" operation is shown in Section 3. Our hand-written interval "and" operation converts its arguments to the bitwise domain, performs a (completely precise) abstract "and" operation in that domain, and then converts the result back to an interval. We did not attempt to compute the condition codes. The precision metric for the interval domain is $N - \log_2(|x|)$ where $|x|$ is the size of an interval. Since the interval domain is well suited for arithmetic operations, it is unsurprising that both versions of "add" are equally precise, at 6.4%. The Hoisted version of "and," on the other hand, is almost four times as precise than the hand-written version (16% vs. 4.1%).

Time to evaluate abstract operations. Figure 9 compares the throughput, in thousands of operations per second, of four different implementations of each type of abstract operation: naive use of the definition of $f^\sharp(a)$ from Section 2, using the caching schemes described in Section 4.2, using functions generated by Hoist, and using functions written by hand. Throughput is computed by measuring how long it takes to apply the operation to 10 million randomly chosen abstract values. The main point to notice is that the hand-written and the BDD-based versions give comparable performance (though the BDD version is usually slower). The cached operations used to generate the BDDs perform adequately for unary operations but are prohibitively slow for binary operations despite using a cache of over 1 MB each.

		bitwise		interval	
		arith	logical	arith	logical
unary	naive	73	71	172	171
	naive+cache	1,069	1,000	573	563
	Hoist	1,189	1,132	2,127	2,069
	manual	1,230	1,209	3,409	3,409
binary	naive	1.75	1.66	0.25	0.25
	naive+cache	155	139	60	60
	Hoist	819	820	1,351	1,355
	manual	833	840	2,192	1,199

Figure 9: Throughput of naive, cached, Hoisted, and hand-written abstract operators in thousands of operations per second

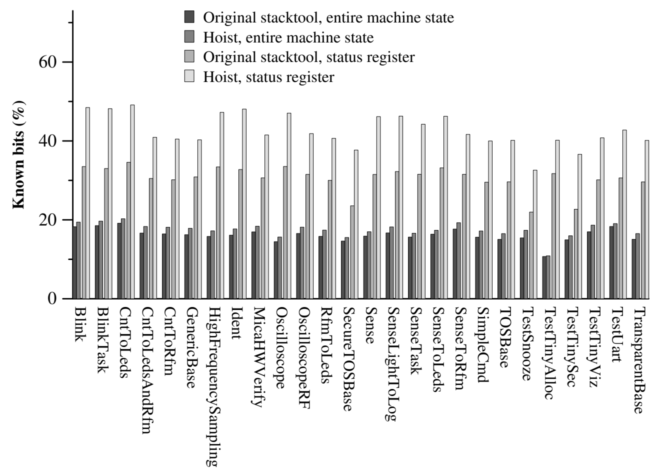

Figure 10: Precision of abstract interpreter using hand-written and automatically generated abstract operations

5.2 Macrobenchmarks

To ascertain the large-scale impact of using abstract operations derived using Hoist, we linked them into the stacktool, replacing the hand-written ALU operations. Stacktool can be configured to perform either a context sensitive or insensitive analysis; we present only the context sensitive results since they are the most useful in practice, and since the effect of using Hoisted abstract operations does not seem to vary between the context sensitive and insensitive analyses. Our test suite is a collection of 26 TinyOS programs [20]: embedded codes for networked sensor nodes that were compiled from up to about 30,000 lines of C.

Precision of stacktool. Our metric for precision is the percentage of static bits of machine state that stacktool can prove hold constant values over all executions. We present two precision results: one for the entire machine state modeled by stacktool, which consists of 32 registers plus the status register (containing the condition code flags), and one for just the status register. It is particularly important that the status register be modeled precisely because it is critical to identifying dead branches in the code, as well as computing a precise estimate of the global interrupt preemption graph—the interrupt mask bit is also contained in the status register.

Figure 10 presents the results of this experiment. On average, the overall precision is improved by a respectable 8% while the precision of the status register is improved by a significant 40%.

These improvements are lower than the 59% and 130% improvements seen in some of the microbenchmarks for several reasons. First, only a small fraction of the operations in a program are binary arithmetic operations (i.e., those most improved under the bitwise domain). Second, many variables encountered in embedded systems have a very limited range [36] and the difference in precision is less pronounced for smaller values.

Speed of stacktool. The original stacktool takes between 0.03 and 8.3 s to run, depending on which of our 26 test cases it is analyzing. The stacktool that is based on Hoisted operations takes between 0.03 and 8.9 s, exhibiting a mean 22% slowdown and a very modest maximum increase in run time: less than 1 s. There are two distinct causes for the slowdown. First, the individual abstract operations are slower, as shown by the microbenchmarks. Second, the increased precision of the Hoisted abstract operations causes the stacktool's abstract interpreter to take longer to reach a fixpoint—in fact, it executes about 50% more abstract operations than the original version for a given input.

Anecdotal results. For several months we have been using the Hoisted AVR operations in day-to-day development of the stacktool. Although the old hand-written operations are still available via a compile-time switch, our experiences with the new operations have been uniformly positive. For example, we have recently started to experiment with pointer analysis in the stacktool. Using the Hoisted operations, we can often substantially narrow down the set of possible targets for store instructions. The hand-written abstract operations, on the other hand, usually lose all precision when analyzing the index registers that contain memory addresses. Similarly, we added a feature to the stacktool giving it a first-class model of the stack pointer; here again, the added precision of the Hoisted operations relative to the hand-written operations was invaluable. In summary, although our original hand-written abstract operations had been tuned until they could effectively model the interrupt mask for many AVR programs, even more tuning would have been required to support the new analysis features described here. Hoist, on the other hand, gives us this precision for free and eliminates the question of whether a little more tuning might improve the results, permitting us to concentrate on more interesting tasks.

6. RELATED WORK

Perhaps the closest work to Hoist is that of Yorsh, Reps, and Sagiv [33, 41], which takes a completely different approach to computing maximally precise abstract operations. Their problem is both easier and harder than ours. It is harder because their abstract values are represented symbolically and are manipulated using a theorem prover. They generate a sequence of approximations to the answer, using counterexamples to find weaknesses of the current approximation. Their approach is easier than ours in that we produce a symbolic representation of $f^\sharp$ whereas they only produce a symbolic representation of $f^\sharp(a)$ for some specific value a. A significant difference between the two approaches lies in the tradeoff between power and performance. Their use of a theorem prover and symbolic representations lets them tackle complex domains such as those supporting shape analysis. However, it takes 27 s to compute $f^\sharp(a)$ for a given value of a; our use of BDDs and simpler representations limits us to simpler domains, but it takes just 1–2 microseconds to compute $f^\sharp(a)$.

A second area of research related to ours is concerned with generating program analyzers. These projects have generally focused on choosing appropriate domains, on frameworks for performing fixpoint computations, and on handling control flow [13, 24, 37, 40]. Our work is complementary to nearly all of this work and, as far as we know, is the first project to generate abstract operations for analyzing object code. For example, PROPAN [16, 22] by Kästner et al. is a system designed to support rapid retargeting of a machine-code analyzer, and yet its capabilities are complementary to those of Hoist since it requires manual specification of abstract operations.

The third body of research related to ours focuses on extracting metadata or other high-level information by observing the behavior of an existing artifact such as a compiler or assembler. For example, Derive [21] infers the encoding of instructions by supplying appropriate inputs to an assembler and observing its output. The superoptimizer [28] tries to find a minimum-length sequence of instructions implementing a function using exhaustive search. Finally, Collberg [8] feeds carefully chosen code fragments to an existing compiler and uses the results to derive a new code generator for his compiler. The idea that these projects have in common with Hoist is the use of implicit specifications: finding a semantically simple interface to a complex system and exploiting it to infer useful properties about the system's behavior.

7. FUTURE WORK

Although Hoist is already useful, there are many possible directions for future work.

Supporting additional abstract domains. Simple abstract domains, such as those that track the sign or the even/odd status of storage locations, could be easily handled within the Hoist framework. We have not bothered with them because they are probably too simple to be useful, and in any case they are subsumed by the bitwise and interval domains. Useful abstract domains that could be supported in Hoist, that are not subsumed by the interval and bitwise domains, include mod-k residues [12], reduced interval congruences [3], and anti-intervals that can represent the knowledge that, for example, $x \neq 0$. On the other hand, Hoist probably cannot support *relational* abstract domains, even for 8-bit architectures: they contain too many elements. Relational domains describe relationships between collections of variables and include the octagon [29] and polyhedron [11] domains.

Supporting additional architectures. The Hoist technique is extremely general: operations can be arbitrary functions from bit-vectors to bit-vectors. Clearly the instructions for architectures other than AVR fit into this model. However, we have not yet Hoisted other instruction sets because there is a fair amount of AVR-specific glue code built into our framework. We are in the process of factoring out the non-portable code.

Supporting additional kinds of instructions. In principle there is no problem Hoisting control flow instructions as well as the logical and arithmetic instructions that we already handle. We have not yet done so for two reasons. First, in our experience abstract control flow is not very difficult to implement manually. Second, the interface between control flow instructions and the rest of the abstract interpreter is substantially more complex than the interface for instructions that only manipulate data.

Exploiting interactions between domains. When a program is analyzed using two or more abstract domains, the whole is often more than the sum of the parts. For example, if a storage location is known to be approximated by the interval $[160, 210]$ and also by the bitwise value $\perp\perp\perp 11011$, then an analyzer can infer that the location contains the concrete value 187. We have used Hoist to create a maximally precise combiner for the bitwise and interval domains. Due to scalability problems we probably cannot optimally combine more than two or three domains. We plan to use Granger's domain product operation [18], which iteratively refines many-way combinations using pair-wise operations as building blocks.

Improving scalability. The Hoist system is the product of a particular set of tradeoffs. One consequence of these tradeoffs is that it is currently not practical to generate operations for architectures with word sizes larger than eight bits. We intend to achieve scalability by making different tradeoffs between build time, precision, run time, and developer effort.

To trade precision for build time we can reduce the size of a domain. For example, for the 16-bit bitwise domain, we could omit all values with 12 or more unknown bits. That is, we would leave somewhat precise values alone but decrease analysis resolution for values that are already imprecise. If the result of an operation is one of these missing values, it can be approximated by rounding down to $\perp$.

To trade run time for build time we can construct a BDD representing one of the caches described in Section 4.2. This will reduce BDD construction time if the cache is significantly smaller than the final result, as it is for the interval domain.

To trade human effort for build time we could use symbolic representations of instructions, dropping our assumption that instructions are black boxes. We are working towards Hoisting both the bitwise and interval domains using symbolic representations that are at roughly the level of detail found in a typical reference manual for a processor. An alternative to having developers type in formulas would be to reuse an existing machine description format such as λ-RTL [30]. Symbolic instructions also make it easier to exploit redundancies between instruction sets. For example, if the shift-left instruction behaves the same across a number of 16-bit architectures, we only have to generate code implementing this operation once. In the long run we hope that Hoisting an instruction set into an abstract domain will simply entail piecing together appropriate previously-generated results.

A final way to improve Hoist's scalability would be to parallelize the generation of abstract operations; this would be easy since there are no dependencies between operations.

8. CONCLUSIONS

Abstract versions of machine instructions are needed to support a wide variety of important analyses and optimizations for object code, and yet implementing these operations is tedious and error-prone. The contribution of this paper is Hoist, a toolchain that gives developers a better alternative: supply a small amount of metadata about each instruction, run Hoist, and then simply link the generated code into an abstract interpreter. In addition to automating a difficult programming task, Hoist has achieved a significant increase in analysis precision and it also validates its own output using an extensive array of tests. Our experience in comparing Hoist with a set of hand-written abstract operations has provided evidence that the developers of operations like this should test them either exhaustively or with random inputs in order to detect errors and operations that are excessively imprecise.

We have made the Hoisted AVR operations freely available:
`http://www.cs.utah.edu/~regehr/hoist/`

Acknowledgments. We would like to thank John Carter, Eric Eide, Matthew Flatt, Wilson Hsieh, Jay Lepreau, Ben Titzer, and the reviewers for their help in improving this paper. This material is based upon work supported by the National Science Foundation under Grant No. 0209185.

9. REFERENCES

[1] Atemu: A sensor network emulator/simulator/debugger. Center for Satellite and Hybrid Communication Networks, University of Maryland, 2004. `http://www.cshcn.umd.edu/research/atemu/`.

[2] Atmel, Inc. ATmega128 datasheet, 2002. `http://www.atmel.com/atmel/acrobat/doc2467.pdf`.

[3] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Proc. of the Intl. Conf. on Compiler Construction (CC)*, Barcelona, Spain, April 2004.

[4] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[5] Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In *Proc. of the 23rd Intl. Conf. on Software Engineering (ICSE)*, pages 47–56, Toronto, Canada, May 2001.

[6] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proc. of the 12th USENIX Security Symp.*, Washington, DC, August 2003.

[7] Cristina Cifuentes. Interprocedural data flow decompilation. *Journal of Programming Languages*, 4(2):77–99, 1996.

[8] Christian S. Collberg. Reverse interpretation + mutation analysis = automatic retargeting. In *Proc. of the ACM SIGPLAN 1997 Conf. on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV, June 1997.

[9] Olivier Coudert and Jean Christophe Madre. Implicit and incremental computation of primes and essential primes of boolean functions. In *Proc. of the Design Automation Conf. (DAC)*, pages 36–39, Anaheim, CA, June 1992.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th Symp. on Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, CA, January 1977.

[11] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of the 5th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 84–97, Tucson, AZ, January 1978.

[12] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *Proc. of the 25th Symp. on Principles of Programming Languages (POPL)*, pages 12–24, San Diego, CA, January 1998.

[13] Matthew B. Dwyer and Lori A. Clarke. A flexible architecture for building data flow analyzers. In *Proc. of the 18th Intl. Conf. on Software Engineering (ICSE)*, pages 554–564, Berlin, Germany, March 1996.

[14] Jakob Engblom. Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis. In *Proc. of the 5th IEEE Real-Time Technology and Applications Symp. (RTAS)*, Vancouver, Canada, June 1999.

[15] Jakob Engblom, Andreas Ermedahl, Mikael Nolin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *Journal of*

Software Tool and Transfer Technology (STTT), 4(4):437–455, August 2003.

[16] Nicolas Fritz, Daniel Kästner, and Florian Martin. Automatically generating value analyzers for assembly code. In *Proc. of the 2003 Workshop on Compilers and Tools for Constrained Embedded Systems (CTCES)*, San Jose, CA, October 2003.

[17] John K. Gough and Herbert Klaeren. Eliminating range checks using static single assignment form. In *Proc. of the 19th Australian Computer Science Conf.*, Melbourne, Australia, January 1996.

[18] Philippe Granger. Improving the results of static analyses of programs by locally decreasing iterations. In *Proc. of the Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 68–79, New Delhi, India, December 1992.

[19] John L. Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(3):422–448, July 1983.

[20] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, Cambridge, MA, November 2000.

[21] Wilson C. Hsieh, Dawson R. Engler, and Godmar Back. Reverse-engineering instruction encodings. In *Proc. of the 2001 USENIX Annual Technical Conf.*, pages 133–146, June 2001.

[22] Daniel Kästner. PROPAN: A retargetable system for postpass optimizations and analyses. In *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Vancouver, Canada, June 2000.

[23] Stephen Lawton. Eternally yours at 8 bits. *Electronic Business*, October 2002.

[24] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Proc. of the 29th Symp. on Principles of Programming Languages (POPL)*, Portland, OR, January 2002.

[25] Jørn Lind-Nielsen. BuDDy–A binary decision diagram package. `http://www.itu.dk/research/buddy/`.

[26] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. of the 10th ACM Conf. on Computer and Communications Security (CCS)*, Washington, DC, October 2003.

[27] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Real-Time Systems*, 17(2/3):183–207, November 1999.

[28] Henry Massalin. Superoptimizer: A look at the smallest program. In *Proc. of the 2nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 122–126, Palo Alto, CA, October 1987.

[29] Antoine Miné. The octagon abstract domain. In *Proc. of the 8th Working Conference on Reverse Engineering (WCRE)*, Stuttgart, Germany, October 2001.

[30] Norman Ramsey and Jack W. Davidson. Machine descriptions to build tools for embedded systems. In *Proc. of the 1998 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 176–192, Montreal, Canada, June 1998.

[31] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proc. of the 27th Intl. Symp. on Microarchitecture (MICRO)*, pages 172–180, San Jose, CA, November 1994.

[32] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. In *Proc. of the 3rd Intl. Conf. on Embedded Software (EMSOFT)*, pages 306–322, Philadelphia, PA, October 2003.

[33] Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *Proc. of the 5th Intl. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Venice, Italy, January 2004.

[34] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *Proc. of the 31st Symp. on Principles of Programming Languages (POPL)*, Venice, Italy, January 2004.

[35] Simulavr: An AVR simulator. `http://savannah.nongnu.org/projects/simulavr`.

[36] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proc. of the ACM SIGPLAN 2000 Conf. on Programming Language Design and Implementation (PLDI)*, pages 108–120, Vancouver, Canada, June 2000.

[37] G. A. Venkatesh. A framework for construction and evaluation of high-level specifications for program analysis techniques. In *Proc. of the ACM SIGPLAN 1989 Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–12, Portland, OR, July 1989.

[38] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM Symp. on Operating Systems Principles (SOSP)*, pages 203–216, Asheville, NC, December 1993.

[39] Zhichen Xu, Barton Miller, and Thomas Reps. Safety checking of machine code. In *Proc. of the ACM SIGPLAN 2000 Conf. on Programming Language Design and Implementation (PLDI)*, Vancouver, Canada, June 2000.

[40] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Proc. of the 20th Symp. on Principles of Programming Languages (POPL)*, pages 246–259, Charleston, SC, January 1993.

[41] Greta Yorsh, Thomas Reps, and Mooly Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Proc. of the 10th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Barcelona, Spain, March 2004.