Automatic Memory Reductions for RTL Model Verification

Panagiotis Manolios[†], Sudarshan K. Srinivasan[‡], and Daron Vroon[†]

[†]College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280 USA
{manolios, vroon}@cc.gatech.edu

[‡]School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332-0250 USA
darshan@ece.gatech.edu

ABSTRACT

We present several techniques for automatically reducing memories in RTL designs. This includes a new memory abstraction algorithm that allows us to greatly reduce the size of memories and a technique based on-term rewriting that further improves the abstraction. In contrast to previously proposed methods for abstracting memories of RTL designs, our methods are general—*e.g.*, they allow us to arbitrarily and directly compare memories—and they are sound and complete—*e.g.*, there are no false positives or negatives. In addition, the combination of our techniques allows us to automatically verify RTL pipelined machine designs beyond the reach of current state-of-the-art methods, as our experimental results show.

1. INTRODUCTION

Reference models for industrial designs tend to be defined at the Register Transfer Level (RTL). These models are used for performance evaluation, testing, generation of lower-level models such as netlists, etc. Ideally, hardware verification efforts should target these models as well. Unfortunately, due to the practical limitations of current verification technology, most techniques can only handle abstractions of RTL models. For example, in the case of pipelined machine verification, RTL designs are often abstracted to term-level models, which can be verified using various decision procedures such as UCLID [4] and DPLL(T) [10]. The problem with such approaches is that the connection between the RTL designs and their abstractions is often difficult to establish formally. In the case of term-level models, the data path and memories are abstracted away, only a small subset of the instruction set is implemented, and processor elements such as decoders and ALUs are replaced by uninterpreted functions. The resulting models are not executable and therefore difficult to relate back to the original RTL models either formally or empirically. Due to this limitation, the impact that such techniques have had in industry has been limited.

In principle, one can use SAT solving techniques to directly verify RTL designs, but this is infeasible. One of the central problems with this approach is that the read and write logic for memories, including register files and caches, leads to prohibitively large SAT problems.

ICCAD'06 November 5-9, 2006, San Jose, CA

Current state-of-the-art tools for RTL verification address this problem by abstracting memories away completely [8, 9]. Instead, the forwarding property of memories (*i.e.*, a read access from a memory gets the most recent value written to the read address) is preserved by constraining the memory interface signals. The problem with such methods is that removing the memories altogether means that they cannot be referred to directly; in fact, the only operations allowed on memories by such methods are reading and writing data. So, for example, it is not possible to test memories for equality and inequality in all contexts without testing the equality of every individual word in the memory, which can lead to a problem as large and intractable as the unabstracted version.

In this paper, we present a collection of novel techniques that allows us to attain drastic improvements over current methods for verifying RTL designs. The first of these is a sound and complete memory abstraction algorithm that reduces memories to a manageable size without eliminating them completely. This technique is more general than current state-of-the-art methods for memory abstraction, since it allows for direct reasoning about memories (*e.g.*, efficient comparison of memories for equality and inequality). In addition, where the functionality of our algorithm does overlap with that of current techniques (*i.e.*, memory reads and writes), our abstraction gives savings that are comparable to those provided by the current state-of-the-art methods.

In addition to the main memory abstraction algorithm, we present two auxiliary techniques that improve its effectiveness. The first of these is a hashing heuristic that reduces the size of Boolean formulas by increasing sharing. The second is a rewriting algorithm that removes unnecessary memory accesses, which allows us to further decrease memory sizes.

We have implemented our memory abstraction techniques in the Bit-level Analysis Tool (BAT), which can be used as a bounded model checker and *k*-induction engine for RTL models. BAT operates over a simple but powerful core language that could easily be used as the target language of synthesizable subsets of Verilog and VHDL. Experimental results demonstrate the effectiveness of our memory reduction techniques and the ability of BAT to prove theorems about RTL models that are beyond the range of current state-of-the-art methods and tools.

The rest of the paper is organized as follows. In Section 2, we briefly review previous RTL verification techniques. In Section 3, we provide a high-level description of our Bit Analysis Tool (BAT). Section 4 describes our memory reduction techniques. In Section 5, we evaluate our work using a number of benchmarks and also provide comparisons with other methods and tools. We end with conclusions and future work in Section 6.

2. RELATED WORK

A variety of techniques for verifying RTL designs that contain memories have been proposed in the literature. An early example is the CLI stack, where alists were used to only keep track of the list of address-value pairs used [2]. A similar technique appears in [20] and a variation of this approach is proposed in [4], where memory is abstracted using restricted lambda expressions. These last two methods are used in the context of term-level reasoning and the memory abstraction is performed manually. In contrast, our approach is automatic.

Ganai et al. [8, 9] give a method for automatically abstracting memories (we refer to this as the GMA approach) by removing the memory from the specification and retaining only the memory interface signals. The forwarding property of memory semantics is preserved by imposing constraints on the memory interface signals. The GMA approach is implemented in a SAT-based bounded model checker for RTL Verilog models. This approach is not as general as ours, as it does not provide the capability to reason directly about memory components. That is, it is limited to reasoning about reads and writes. In this domain our main abstraction algorithm gives comparable savings to the GMA approach, and our hashing and rewriting techniques provide additional savings.

A more recent approach for verifying RTL models is based on predicate abstraction and Counter Example Guided Abstraction and Refinement (CEGAR). The approach is implemented in the VCEGAR tool [12]. Given a Verilog model, VCEGAR abstracts the model using predicate abstraction. A model checker is used to check the property on the abstracted model. If a counter-example is found, it is evaluated on the original model. If it is spurious, the abstract model is refined. This process is repeated until either the property is proved, a real counter example is generated, or no new predicates can be determined. This method is not complete and fails on examples we can easily handle.

Another approach for verifying RTL Verilog models based on CEGAR is introduced in [1] and implemented in the Reveal system. This approach automatically abstracts Verilog models and properties to the logic of Counter arithmetic, with restricted Lambda expressions and Uninterpreted functions (CLU) logic, which can then be checked efficiently using a decision procedure. If a spurious counter example is generated, the abstract model is refined using minimal unsatisfiable subset (MUS) extraction. The Reveal system has been used to check parts of the safety property based on the Burch and Dill commuting diagram for a 4-bit pipelined machine [5]. Our tool can handle much more complex systems than this. As our experimental results show, BAT can be used to check both safety and liveness properties of more complex 32-bit pipelined machine models.

RTL designs can be verified using theorem provers but this requires extensive user effort. In [16], the authors present an approach for verifying RTL pipelined machine models using a combination of deductive reasoning and decision procedures. The pipelined machine models they verified are more complex than the machines verified in this paper, but their approach still requires person-weeks of time and theorem proving expertise, whereas verification with BAT is automatic.

3. BIT-LEVEL ANALYSIS TOOL

Our memory analysis and reduction techniques are implemented in the Bit-level Analysis Tool, or BAT, which solves RTL bounded model checking and *k*-induction problems. Given a machine description and LTL property in our own S-Expression based language (similar to Lisp or Scheme in syntax) and a number of steps, BAT checks the property using SAT-based methods and returns a counter-example if one exists. In this section, we give a brief overview of BAT, highlighting its key features.

We carefully designed the BAT tool to be simple to use, expressive, and as general as possible. Because of its generality, the BAT specification language can easily be the target language of synthesizable subsets of Verilog and VHDL. In fact BAT can handle features of these languages not present in other state-of-the-art bit-level analysis tools. For example, as can be seen in Figure 1, BAT supports user-defined functions, which are not supported in VCEGAR [12].

BAT is implemented in Lisp, and can be used interactively using Lisp's built-in development environment. This makes it easy to set up automatic interaction between Lisp and BAT. For example, users can write functions to generate BAT models given a set of input parameters, or even add features to the language by writing code to translate new features into the core BAT language. The code in Figure 1 was generated by a function that takes a positive integer, n, and generates an n-bit ALU specification. In this case, $n = 2$. Lisp makes writing model generators and language extensions particularly easy, since it has a built-in parser for S-expressions.

The BAT machine description has four required sections: `:vars`, `:init`, `:trans`, and `:spec`. The `:vars` section contains global variable declarations. These variables represent the state of the machine. The `:init` section is a Boolean formula over the variables declared in `:vars` that returns 1 (representing true) iff a machine state is a valid initial state of the machine. The `:trans` section is another Boolean formula over the current and next state of variables that describes the transition relation. Given two states, s and s', the transition relation will return true if s can transition to s' in one step. Finally, the `:spec` section describes the LTL formula to be checked for the machine. In addition, users may define functions in the `:functions` section, constants in the `:consts` section, and static definitions that do not change with the transition relation in the `:definitions` section.

The BAT language operates over three kinds of data types: bit vectors, memories, and multiple-value types (tuples of bit vectors and memories). Bit vectors can be given in Boolean, octal, hexadecimal, signed decimal or unsigned decimal representations. The Boolean, octal, and hexadecimal representations are symbols beginning with 0b, 0o, and 0x respectively, *e.g.*, 0b0011, 0o72, and 0x4A3. Decimal numbers are by default interpreted as a signed (2's compliment) bit vector. To specify the use of the unsigned representation, the character 'u' is added to the end, *e.g.*, 7u. There is no constant representation for memories. Multiple value expressions are written as `(mv ...)`.

The BAT language is strongly typed. Each bit vector has a fixed number of bits, and each memory has a fixed wordsize and number of words. Bit vectors with different numbers of bits cannot be compared or combined. For example, the bitwise and function, `and`, cannot be given a 3-bit bit vector and a 4-bit bit vector as arguments. However, the number of bits in a vector specified as a decimal number is ambiguous. For example, the number 7 can be represented as the bit vectors `0b0111`, `0b00111`, `0b000111`, etc. In these cases, the BAT type checker performs type inference to discover what type the integer should have based on its context. For example, if `x` is a 5-bit bit vector variable, and the BAT specification contains the formula `(and x 7)`, then 7 will be interpreted as `0b00111` in this instance. The only requirement in cases of type inference is that integers be representable in the number of bits dictated by the context. For example, the previous example would not work if `x` were a bit vector of length 2, since 7 requires 4 bits to be represented as a signed bit vector.

```
((:functions
  (maj (1) ((a 1) (b 1) (c 1))
     (or (and a b) (and b c) (and a c)))
  (fa (2) ((a 1) (b 1) (cin 1))
     (cat (maj a b cin) (xor a b cin)))
  (mux-4 (1) ((i0 1) (i1 1) (i2 1) (i3 1) (sel 2))
     (local
      ((nsel0 (not (sel 0)))
       (nsel1 (not (sel 1)))
       (v0 (and i0 nsel0 nsel1))
       (v1 (and i1 (sel 0) nsel1))
       (v2 (and i2 nsel0 (sel 1)))
       (v3 (and i3 (sel 0) (sel 1))))
      (or v0 v1 v2 v3)))
  (alu-slice (2) ((a 1) (b 1) (cin 1) (bn 1) (op 2))
     (local
      ((nb (xor bn b))
       (res0 (and a nb))
       (res1 (or a nb))
       (((cout 1) (res2 1)) (fa a nb cin)))
      (cat cout (mux-4 res0 res1 res2 1u op))))
  (alu-2-bit (4) ((a 2) (b 2) (bn 1) (op 2))
     (local
      ((c 2))
      (((t0 (c 0))
        (alu-slice (a 0) (b 0) bn bn op))
       ((t1 (c 1))
        (alu-slice (a 1) (b 1) t0 bn op))
       (zero (= c 0)))
      (cat t1 c zero))))
 (:vars (i1 2) (i2 2) (bn 1) (op 2)
        (out 2) (cout 1) (zero 1))
 (:init (and (= out 0)
             (= cout 0b1)
             (= zero 0b0)))
 (:trans (= (cat (next cout) (next out) (next zero))
            (alu-2-bit i1 i2 bn op)))
 (:spec (AG (<-> zero (not cout)))))
```

Figure 1: A simple ALU description in BAT

The example in Figure 1 defines a simple ALU. It contains only bit vectors, but demonstrates many of the features of the BAT language. The ALU description begins with function definitions. The first of these is `maj`, which is the majority function on 3 inputs. The first field in a function definition is the name, followed by the type. The type of `maj` is a bit vector of one bit. The third element of the definition is the function parameters, which are also typed. In this case, all of the inputs are bit vectors of length 1. The last element of the definition is the body of the function, which should be in terms of the parameters only (not the global variables). The majority function tests to see if any pair of the three inputs are both true.

The second function is a full adder, which returns a bit vector of length 2, which is the concatenation of the majority and the exclusive or of the three inputs. This is followed by a 4-bit multiplexer, which demonstrates the basic usage of the `local` construct. In its simplest form, a `local` is similar to a `let*` in Lisp or Scheme. Its first argument is a list of bindings, each of which consists of a variable name and an expression to which the variable should be bound. The bindings occur in the order in which they appear. For example, `nsel0` is set to `(not (sel 0))`, `nsel1` is set to `(not (sel 1))`, and `v0` is set to `(and i0 nsel0 nsel1)`. The semantics of these bindings are simply to replace each variable with the expression to which it is bound. So, for example, `v0` is bound to `(and i0 (not (sel 0)) (not (sel 1)))`.

The `alu-slice` and `alu-2-bit` functions demonstrate two other features of the `local`. In `alu-slice`, the last binding contains a list of variable declarations where we expect a single variable. Here `cout` and `res2` are declared to be 1-bit bit vectors. Recall that the full adder returns 2 bits. These bits are split between

```
((:vars (mem 8 4)
        (adr 3)
(val 4))
 (:init 0b1)
 (:trans 0b1)
 (:spec (= (get (set mem adr val) adr) val)))
```

Figure 2: A trivial example of memory usage in BAT.

`cout` and `res2`. The leftmost bit is the most significant in the BAT language, so `cout` gets the high bit of the output. In general, any number of variables can be bound at once in this manner, but their lengths must add up to the length of the expression to which they are being bound (which must be a bit vector). In `alu-2-bit`, an extra argument is given at the beginning of the local, which is a list of variable declarations. In this case, there is 1 2-bit variable, `c`, declared. Now the user may bind single bits or bit ranges of `c` in the bindings. For example, the first binding binds `t0` and the 0th bit of `c` simultaneously. All of the bits must be bound exactly once.

The `:vars` section declares the seven variables to be used as the machine state. The `:init` formula says that all initial states should have an `out` of `0`, a `cout` of `0b1`, and a `zero` of `0b0`. The `:trans` states that the concatenation of the next states of the `cout`, `out`, and `zero` variables should be the output of the `alu-2-bit` function applied to the `i1`, `i2`, `bn`, and `op` arguments. Finally, the `:spec` says that for every step of every computation, it should be the case that `zero` is the negation of `cout`.

Figure 2 shows a trivial example of a machine description using memories. A memory is defined by giving 2 positive integers for the type instead of 1, as is the case with bit vectors. The first of these is the number of words in the memory, and the second is the number of bits in each word. In this case, `mem` is declared to be a memory with 8 4-bit words. The use of memories is limited to `get`, `set`, `=`, and `if`. That is, memories can be read from (`get`), written to (`set`), tested for equality (`=`), and conditionally returned (`if`). The `set` function takes a memory, an address, and a value, and returns the memory resulting from updating the given address in the given memory to the given value. In this case, the `set` returns the memory resulting from setting address `adr` to `val`. Likewise, the `get` function takes a memory and an address, and returns the value of the word at the given address in the given memory. In this example, the `get` takes the result of the `set` function, and reads the address corresponding to `adr` from it. Note that the memory, the address, and the value are all arbitrary. The `:init` function does not place any restrictions on their value.

3.1 BAT Compilation

After syntax and type checking a model, BAT compiles it into a SAT problem. First, the initialization, transition, and specification formulas are converted to a formula containing only `set`, `get`, `if`, `=`, `<->`, `and`, `not`, and `next` formulas. All variables except memories are replaced by sequences of 1-bit bit vectors. This is roughly a Reduced Boolean Circuit (RBC) plus `if` and memory operations.

Next, BAT "unrolls" the transition and specification formulas the user-specified number of steps using standard techniques [7]. To unroll the description n steps, $n+1$ new variables of the form `Vi_v` are created for each variable, v, in our model (where $0 \leq i \leq n$). The initialization function, I is instantiated with `V0_v` for each variable, v. For each $0 \leq i < n$, we create formula T_i by substituting `Vi_v` for v and `Vj_v` for `(next v)` (where $j = i+1$) in the transition relation, T. We make the same substitutions in our specification, S to get S_i for each $0 \leq i \leq n$ if it is an `AG` or `AF` formula.

The resulting formula is $I \wedge \bigwedge_{i=0}^{n} T_i \wedge \bigwedge_{i=0}^{n} \neg S_i$ if S is an `AF`

form, and $I \wedge \bigwedge_{i=0}^{n} T_i \wedge \bigvee_{i=0}^{n} \neg S_i$ if S is an AG function.

Finally, we perform our memory abstraction and translate the resulting RBC into CNF.

4. MEMORY ABSTRACTION

In this section, we describe our memory reduction techniques. These include the main memory abstraction algorithm, as well as heuristic uniqueness reductions and a term rewriting algorithm that improve the overall effectiveness of the main algorithm.

4.1 Memory Reduction Algorithm

Through the rest of the paper, we denote log base 2 as lg. Let f be a Boolean formula resulting from the unrolling of a BAT machine description. Let M_f be the set of variables in f of some memory type.

In order to preserve memory equality and inequality for the abstract model, we need to apply the same abstraction to memories that are tested for equality, or memories that are used in the same context. For example, if our formula contains the subformula `(= m1 (if (= x y) m2 m3))`, then `m1`, `m2`, and `m3` need to be abstracted in the same way so that `m1` can be easily compared to `m2` or `m3`, whichever is returned by the `if` statement. The following two definitions help us capture this notion as an equivalence relation.

DEFINITION 1. *The* base memories *of a formula, f, which has a memory type is defined recursively as follows:*

- $base(\texttt{(if } e_1\ e_2\ e_3\texttt{)}) = base(e_2) \cup base(e_3)$,
- $base(\texttt{(set } e_1\ e_2\ e_3\texttt{)}) = base(e_1)$,
- $base(x) = \{x\}$ *when x is a variable.*

DEFINITION 2. *The* equality test relation for formula f, *denoted R_f is the relation over M_f such that* $R_f = \{(m_1, m_2) \in M_f \times M_f \mid (\exists e :: \{m_1, m_2\} \subseteq base(e)) \vee (\texttt{(= } e_1\ e_2\texttt{)} \sqsubseteq f \wedge m_1 \in base(e_1) \wedge m_2 \in base(e_2))\}$. *We denote the transitive closure of R_f as E_f.*

LEMMA 1. *E_f is an equivalence relation.*

We denote the equivalence class induced by E_f containing a memory variable, m, as $[m]_{E_f}$. Now, in order to compare memories, we need to know what addresses we use in `set` and `get` formulas containing memories of a given class. The idea is that the data at these addresses is now constrained by our formula, and must therefore be constrained in the same way for the abstract version of the formulas. The following definition serves this purpose.

DEFINITION 3. *The* address set *of a class of memories, $C \in \{[m]_{E_f} \mid m \in M_f\}$, in formula f, is the set*

$$A_C^f = \{e \sqsubseteq f \mid (\texttt{(set } e_1\ e\ e_2\texttt{)} \sqsubseteq f \vee \texttt{(get } e_1\ e\texttt{)} \sqsubseteq f) \wedge base(e_1) \subseteq C\}.$$

For all the addresses in the address set of a class of memories, we create a shorter bit vector for addressing the abstract memories.

DEFINITION 4. *Given an equivalence class of memory variables, C, induced by E_f, and an expression $e \in A_C^f$, an* abstract address *of e is a fresh (in f) variable of $\lceil \lg |A_C^f| \rceil$ bits. We denote the abstract address for e as $\hat{e}$.*

A first approximation at memory abstraction would be to replace each memory, m by a new memory with $|A^f_{[m]_{E_f}}|$ words. The idea would be that we only care about those words that we read and write, so the others can be thrown out. However, recall that the ability to test the equalities of memories is an important feature of BAT. The problem with our first approximation of memory abstraction is that even if we know that all the addresses we look at are the same, it is still possible for the words we haven't seen to be different. As a simple example, suppose we have two uninitialized memories, `m1` and `m2`, each with two words of two bits each. Now suppose we had the formula `(not (= (set m1 0 0) (set m2 0 0)))`. Our first approximation of a memory abstraction would reduce `m1` and `m2` each to 1 word apiece, corresponding to word 0 of each memory. Those reduced memories would be trivially equal. However, since word 1 of each memory is uninitialized, it is possible for word 1 of `m1` to be 0 and word 1 of `m2` to be 1. So, this abstraction is not equisatisfiable with the original function.

In order to allow for memory equality and inequality, we need to represent those parts of the memory that is never accessed. Since we never view these words, they have unconstrained values, so the SAT solver can give them any value it needs to in order to find a satisfying assignment. In order to abstract away a large portion of these unseen words while still allowing the SAT solver to assign values as it needs to, we use the following abstraction.

DEFINITION 5. *Given a function $m \in M_f$ with wordsize w, an* abstract memory *for m, denoted $\hat{m}$, is a memory variable that has $|A^f_{[m]_{E_f}}|w + \lceil \lg |[m]_{E_f}| \rceil$ bits, and the same wordsize as m.*

Since $\lceil \lg |[m]_{E_f}| \rceil$ bits have $|[m]_{E_f}|$ different configurations, this allows the SAT solver to give each memory in a given class a different value for the abstraction of the "unseen" parts of the memories.

Finally, we abstract the formula, f, by replacing memories and addresses with their abstract counterparts and constraining the abstract values appropriately to be sure that we maintain the forwarding property of memories.

DEFINITION 6. *The* memory abstraction *of f is obtained by doing the following.*

- *Create f_1 by adding constraints that imply that $e_1 = e_2 \Leftrightarrow \hat{e}_1 = \hat{e}_2$ for every pair of addresses e_1, e_2, and for each equivalence class, C, induced by E_f, $u2n(e) < \lceil \lg |A_C^f| \rceil$ for all $e \in A_C^f$, where u2n converts an unsigned bit vector to the natural number it represents.*
- *Substitute each memory variable $m \sqsubseteq f_1$ with $\hat{m}$ to create f_2.*
- *Substitute every subformula of f_2 of the form* `(set` $e_1\ e_2\ e_3$`)` *with* `(set` $e_1\ \hat{e}_2\ e_3$`)` *to get f_3.*
- *Substitute every subformula of f_3 of the form* `(get` $e_1\ e_2$`)` *with* `(get` $e_1\ \hat{e}_2$`)` *to get $\hat{f}$.*

We denote the abstract version of f as $\hat{f}$.

THEOREM 1. *f is satisfiable if and only if $\hat{f}$ is satisfiable.*

4.2 Cost

In order to compile a BAT specification to SAT, we need to transform memories into sequences of bits. In this section, we describe the compilation of memory-related formulas, and calculate their cost in number of SAT clauses generated. Let μ be an arbitrary memory-typed subformula of a formula, f. Let $v_\mu \in base(\mu)$,

$m = |[v_\mu]_{E_f}|$, $A^f_{[v_\mu]_{E_f}} = \{\alpha_1, \ldots, \alpha_n\}$, s be the number of bits in each α_i, and w be the wordsize of v_μ. Let $\hat{\mu}$ be the abstracted version of μ, and $\hat{\alpha}_i$ denote the abstract address for α_i. Note that $\hat{\mu}$ has $nw + \lceil \lg m \rceil$ bits, and each $\hat{\alpha}_i$ has $\lceil \lg n \rceil$ bits.

First, there is a one time cost for the address abstraction constraints. One way to express these constraints is the way they are given in Definition 6. While this is simple to state, it does not lead to the most efficient translation to SAT. Instead, we further limit the values of the $\hat{\alpha}_i$ as follows. First, for all $1 \leq j < i \leq n$, we create a variable, E^i_j, and constrain it with the formula $E^i_j \Leftrightarrow (\alpha_i = \alpha_j)$. Each of these is representable in $4s+1$ clauses, giving us a total of $(4s+1)\frac{1}{2}n(n-1)$ clauses. Next, we constrain each $\hat{\alpha}_k$ with the formula

$$[\forall 1 \leq i < k,\ (\neg E^k_1 \wedge \ldots \wedge \neg E^k_{i-1} \wedge E^k_i) \Rightarrow (\hat{\alpha}_k = n2u(i-1, \lceil \lg n \rceil))] \ \wedge$$
$$[(\neg E^k_1 \wedge \ldots \wedge \neg E^k_{k-1}) \Rightarrow \hat{\alpha}_k = n2u(k-1, \lceil \lg n \rceil)]$$

where $n2u(n,w)$ returns the w-bit unsigned bit vector representing the natural number, n. This sets $\hat{\alpha}_1$ to be 0, $\hat{\alpha}_2$ to be 0 if $\alpha_1 = \alpha_2$, or 1 otherwise, and so on. Notice that these constraints imply the simpler constraints from Definition 6. However, we have further constrained each $\hat{\alpha}_i$ to a small set of concrete values: we now know that $\hat{\alpha}_i < i$ for every $1 \leq i \leq n$. This will allow us to create more efficient translations for `set` and `get` formulas. Each $\hat{\alpha}_i$ requires $i\lceil \lg n \rceil$ clauses to constrain, leading to $\frac{1}{2}n(n-1)\lceil \lg n \rceil$ clauses. Adding this to the clauses needed to constrain the E^i_j, we get a total clause count of

$$\left(2s + \frac{1}{2}\lceil \lg n \rceil + \frac{1}{2}\right)(n^2 - n) \tag{4.1}$$

Given a form `(get` $\hat{\mu}$ $\hat{\alpha}_i$`)` we create a new bit vector variable, v, of w bits. Recal that $\hat{\alpha}_i < i$. We therefore know that this particular `get` form will return one of the first i words of $\hat{\mu}$. We therefore constrain v with the formulas $\forall 0 \leq j < i$, $(\hat{\alpha}_i = n2u(j, \lceil \lg n \rceil)) \Rightarrow (v = \hat{\mu}_i)$, where $\hat{\mu}_i$ is the ith word of $\hat{\mu}$. Finally, we replace the `get` form with v. Each constraint requires $2w$ clauses to represent in CNF, resulting in a total of $2wi$ clauses to read from address $\hat{\alpha}_i$.

For the form `(set` $\hat{\mu}$ $\hat{\alpha}_i$ e`)`, we again know that e will be written to one of the first i words of $\hat{\mu}$. For $0 \leq j < i$, and for all $0 \leq k < w$, we create the formula $(\hat{\alpha}_i = n2u(j) \ \vee \ \hat{\mu}^k_j) \ \wedge \ (\hat{\alpha}_i \neq n2u(j) \ \vee \ e^k)$, where e^k is the kth bit of e, and M^k_j is the kth bit of the jth word of $\hat{\mu}$. This formula simply returns e^k if $\hat{\alpha}_i = n2u(j)$, and $\hat{\mu}^k_j$ otherwise. We concatenate these formulas together with all the words at address i or higher to create the new memory. Each of these formulas requires 2 clauses, so the number of clauses required to write to address $\hat{\alpha}_i$ is $2wi$.

Finally, for memory equality, between two abstract memories with $nw + \lceil \lg m \rceil$ bits, which takes $2(nw + \lceil \lg m \rceil)$ clauses.

Therefore, if there are n unique accesses (`set`s or `get`s), and k equality tests, for a class of memories, the total number of clauses needed for memory-related formulas is

$$\left(2s + w + \frac{1}{2}\lceil \lg n \rceil + \frac{1}{2}\right)(n^2 - n) + 2k(n + \lceil \lg m \rceil) \tag{4.2}$$

4.3 Heuristic Uniqueness Reductions

From Equation 4.2, it is clear that the number of memory accesses is the factor that has the greatest effect on the efficiency of our abstraction. Therefore, prior to performing the main memory abstraction algorithm, we perform analyses in order to minimize the number of memory accesses. The first of these is an aggressive, heuristically guided uniqueness reduction. This is based on well-known techniques for SAT conversions [3], but we do not know of any work regarding the heuristics we use to guide our reductions.

We begin by storing subformulas in a hashtable. Each entry has a key of the function name and pointers to each argument of the function. The value is a pointer to a data structure representing the formula. Whenever we create a new formula, we check if it exists in our hashtable already. If so, we use its value. Otherwise, we create a new data structure, and add it to the hashtable. This way, syntactic equality and pointer equality are the same, and we can easily tell if two addresses are syntactically equal.

In order to improve sharing and syntactic equality, we leverage the associativity and commutativity of certain functions in our language. Each formula, as it is created is given a unique integer value. Arguments to commutative functions are sorted according to this value, so that two formulas will not be syntactically different just because they have their arguments in different orders.

We also leverage the associativity of certain functions. However, we have found that it is not always beneficial to normalize in this way. The problem comes when applying associativity destroys subformula sharing. For example, if there is a formula of the form `(and` e_1 `(and` e_2 e_3`)` e_4`)`, then aggressive normalization would transform this into `(and` e_1 e_2 e_3 e_4`)`. However, if the subformula `(and` e_2 e_3`)` exists elsewhere, the structure will be shared in the original formula, but not in the normalized formula. Since sharing is a powerful tool for decreasing the size of CNF formulas, performing this normalization is counterproductive. We therefore have a heuristic so that associativity normalization does not occur when normalization would decrease sharing in the final formula. As we show in Section 5, these heuristic uniqueness reductions have a significant effect on the performance of our memory abstraction technique.

4.4 Memory Rewriting

Another technique that we use to improve the effectiveness of our memory abstraction is the use of rewrite rules designed to simplify away unnecessary memory accesses. In this section, we describe our rewriting technique.

LEMMA 2. *The following hold for all values v_1 and v_2, all expressions of memory type m, m_1, and m_2, and all expressions of the appropriate bit vector type, e_1, e_2, and e_3:*

- `(get (set` m e_1 v_1`)` e_2`)` $=$ `(if (=` e_1 e_2`)` v_1 `(get` m e_2`))`
- $e_1 = e_2 \Rightarrow$ `(set` m e_1 `(get` m e_2`))` $= m$
- `(get (if` e_1 m_1 m_2`)` e_2`)` $=$ `(if` e_1 `(get` m_1 e_2`) (get` m_2 e_2`))`.

We apply these rules before abstracting the memories in order to reduce the number of `set`s and `get`s. The result is a sometimes dramatic reduction in the number of addresses we need to consider (see Section 5).

The first rule is useful when a given memory is only used for reading and writing, and not for comparison. Here, every `set` formula that lies inside a `get` formula gets rewritten away.

The second rule is applied in a similar manner as the first, allowing us to eliminate one address. The final rule pushes `get`s inside `if` statements. At first glance, this rule seems to increase the size of our formula, rather than decrease it. The outer `get` is replicated twice inside the `if` formula. However, due to our formula uniqueness reductions, this does not cause much of an increase in the size of the resulting CNF formula. In addition, this rule can greatly reduce memory sizes, leading to an overall savings. Suppose we had two expressions of memory type, m_1 and m_2 that had different base memories. Then consider the following formula:

`(get (set (if e1` m_1 m_2`) a1 v1) a2)`

Recall from Section 4.1 that the base memories of the `if` statement are the union of the base memories of m_1 and m_2. This means

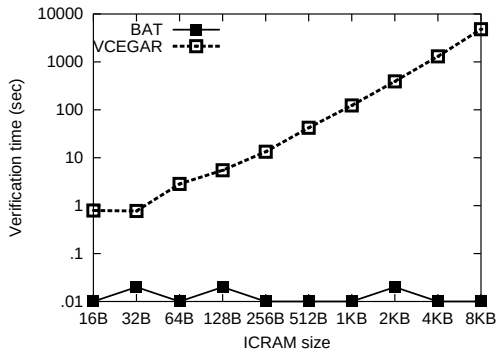


Figure 3: The graph compares the verification times for the *icram* benchmarks using both BAT and VCEGAR.

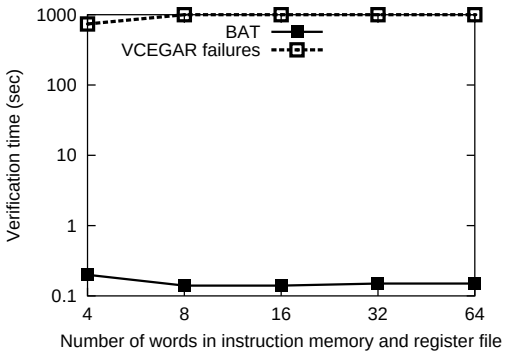


Figure 4: The graph compares the running times of BAT and VCEGAR obtained from verifying the 2 stage pipelined machine benchmarks.

that these memories will be in the same equivalence class, and all the addresses read or written to in either base memory must be represented in the abstraction of both base memories. Now consider what happens when we apply the final rewrite rule:

```
(if e1
    (get (set m1 a1 v1) a2)
  (get (set m2 a1 v1) a2))
```

Now the `if` has a bit vector type rather than a memory type, since the `get` happens inside the "then" and "else" clauses. Therefore, the base memories of m_1 and m_2 are no longer equated (unless there is some other place in the overall formula where they are equated). So, for example, if there were 10 sets in m_1 and 10 sets in m_2, whose addresses do not overlap, then the application of this rule decreased the size of the abstract memories for the base memories of m_1 and m_2 from 20 to 10 each.

5. RESULTS

We evaluate the memory abstraction technique implemented in BAT using five benchmark sets. We compare BAT with the VCEGAR tool and find that we can efficiently verify problems with large memories that VCEGAR cannot handle, and with other benchmarks, we get several orders of magnitude speed up in verification time over VCEGAR. All the experiments described in this paper were run on a 3.06 GHz Intel Xeon with an L2 cache size of 512KB. We used the Siege SAT solver (variant 4) [17] to check the SAT problems generated by BAT.

Instruction Cache RAM (ICRAM) Unit (*icram*) Benchmarks: The *icram* benchmarks are obtained from the Sun PicoJava II microprocessor's [19] Instruction Cache RAM unit, and were also used to evaluate the VCEGAR tool in [12]. The property verified is that the data input is written to the higher 32 bits of the location in the RAM corresponding to the input address if the memory write signal is enabled. We translated the PicoJava's ICRAM unit (described in Verilog) and the property to a BAT specification.

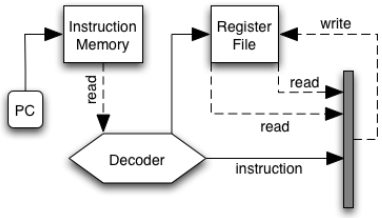
The various *icram* benchmarks are obtained by varying the size of the RAM unit. The results obtained by checking the property using both BAT and VCEGAR are shown in Figure 3. Note that the y-axis is a log scale. The VCEGAR verification times increase exponentially in the size of the RAM, while they remain constant for BAT. In fact, BAT was able to solve the specification without calling the SAT solver by using the rewrite rules described in Section 4.

Two Stage Pipelined Machine (*2f*) Benchmarks: The *2f* is a flushing based refinement theorem [14] for a simple 2 stage pipelined machine that implements only an add instruction and has two memory elements, which are the instruction memory and the register file as shown in Figure 5(a). We used the *2f* benchmarks to compare BAT with VCEGAR, and found that VCEGAR cannot handle any of these benchmarks. Since the verification of pipelined machines is an interesting class of problems [13, 18, 11], we give a detailed analysis of the memory accesses patterns and the reductions in the size of the memory elements that can be obtained using BAT's memory abstraction technique for these benchmarks.

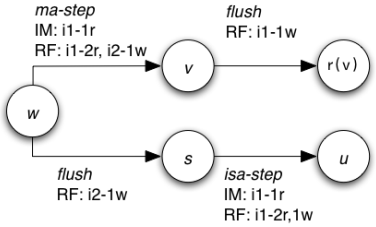
Figure 4 compares the running times from verifying the *2f* benchmarks using both BAT and the VCEGAR tool. The various *2f* benchmarks are obtained by increasing the number of words in the instruction memory and the register file. BAT is able to handle all the benchmarks in less than a second. For the benchmarks with 4 words in both the memory elements, VCEGAR fails to find any solution at all. On all others, VCEGAR is unable to find a solution in 1000 seconds.

Figure 5(b) shows the memory accesses that occur for *2f*. The states *w*, *s*, *v*, *u*, and $r(v)$, in Figure 5(b) are used to describe the property being checked. These states are obtained starting from an arbitrary initial pipelined machine state *w* using operations *ma-step*, *isa-step*, *flush*, and some other operations that do not access any of the memory elements (not shown in the figure as they are not relevant to our analysis). The *ma-step* and the *isa-step* operations correspond to single steps of the pipelined machine and its instruction set architecture. *isa-step* always incurs a read access to the instruction memory, and two read accesses and one write accesses to the register file. If there is a valid instruction in the first latch of a pipelined machine state, then *ma-step* incurs a read access to the instruction memory, and two read accesses and one write accesses to the register file. The *flush* operation steps the pipelined machine forward without fetching any new instructions by introducing a bubble in the first latch (in this case, the only latch) in the pipeline. If there is a valid instruction in the first latch, then the *flush* operation incurs only a write access to the register file. The *flush* operation reads the instruction memory once and the register file twice, but the result of these accesses are not used to update any state variables.

We call the accesses to the memory that have an effect on the final property being checked as relevant accesses and those that do not have an effect as irrelevant accesses. It is not always possible for BAT to determine if an accesses is relevant or not. The number of words in the abstraction of a memory element is the sum of all unique and relevant read and write accesses to that mem-



(a) High-level organization of the 2Stage pipelined machine.



(b) Analysis of memory access for the 2Stage pipelined machine benchmark.

Figure 5: 2Stage pipelined machine benchmark.

NW	D		D-U		D-R		D-UR	
	n	T(secs)	n	T(secs)	n	T(secs)	n	T(secs)
8	0	.07	0	.07	8	0.22	17	0.54
16	0	.13	0	.14	16	0.78	33	2.21
32	0	.27	0	.27	32	4.29	65	12.47
64	0	.61	0	.57	64	43.13	129	FAIL
128	0	1.21	0	1.36	128	FAIL	257	FAIL
256	0	3.16	0	3.25	256	FAIL	513	FAIL
512	0	12.45	0	9.95	512	FAIL	1,025	FAIL

Table 1: Verification statistics for the Out-of-order memory update (*omu*) benchmark.

N	3c (secs)	5fb (secs)	5f (secs)
2^2	0.79	0.77	1.29
2^4	1.58	2.97	7.13
2^8	4.19	3.22	34.80
2^{16}	13.47	15.10	282.67
2^{32}	50.74	58.33	1,278.65

Table 2: Variation in BAT verification times for the *3c* and *5f* benchmarks with increases in data path and memory sizes. N indicates the number words in the register file and memories.

ory element. For the *2f* benchmark, the instruction memory incurs only one read access corresponding to instruction *i1* in Figure 5(b), so the abstracted instruction memory should have only one word. The register file incurs two reads and a write corresponding to instruction *i1*, and a write corresponding to instruction *i2*, and so the abstracted register file should have four words. The size of the abstract instruction memory and data memory obtained using BAT are one and four, respectively, irrespective of the size of the original memories, which is consistent with the abstractions obtained from our analysis. Note that all these accesses occur twice when stating the flushing theorem as can be seen from Figure 5(b). Therefore, if we do not use uniqueness analysis, the abstract instruction memory and register file have 2 and 8 words, respectively.

Out-of-order Memory Update (*omu*) Benchmarks: The *omu* benchmark models out-of-order retirement of instructions—a feature that can be found in many commercial microprocessors such as the Intel XScale [6]—as a sequence of out-of-order updates to a memory. The effectiveness of using rewrite rules described in Section 4 for abstracting memories is demonstrated using the *omu* benchmark. In the *omu* benchmark, an initial memory *mem* with 65536 words is modified by a sequence of writes, where each write in the sequence writes to a different memory location, resulting in memory *mem1*. This sequence of writes is again used to modify memory *mem*, but, in a different order resulting in memory *mem2*. The property checked is that a read from *mem1* at one of the locations updated by the write sequence is equal to a read from the same location at *mem2*.

The verification results for the *omu* benchmark are shown in Table 1. In the table, NW is the number of writes to the memory, n is the number of words in the abstract memory, and T is the total verification time. The columns D, D-U, D-R, D-UR in Table 1 indicate using BAT with default options (includes uniqueness analysis and the use of rewrite rules), with uniqueness analysis turned off, with rewrite rules turned off, and with both uniqueness analysis and rewrite rules turned off, respectively. A "FAIL" entry in the table indicates that BAT ran out of memory and could not solve the problem.

When rewrite rules are used, the verification problem is solved by using BAT's simplification engine and no SAT solver is required. If rewrite rules are not used, the number of words in the resulting memory abstractions are large, and are significantly more difficult for BAT to analyze.

Five Stage Pipelined Machine (*5f*) Benchmarks: The *5f* benchmark is a flushing based refinement theorem [14] for a 5 stage pipelined machine model that implements simple ALU, load, store, and branch instructions. To support these instructions, the model has an instruction memory, a register file, and a data memory. We also use BAT to check a buggy version of *5f*, we call *5fb*.

The flushing based refinement theorem for *5f* and *5fb* is similar to that for *2f*, the difference being in the number of *flush* operations required to state the correctness property. The 5 stage pipelined machines requires at least 5 *flush* operations. The number of *flush* operations used to state the theorem can be an upper bound on the actual number of *flush* operations required. The number of words in the abstracted register file, data memory and instruction memory are 12, 7, and 2, respectively. If uniqueness analysis is not used, the number of words in the abstracted register file and data memory increase to 24 and 20, respectively.

Table 2 shows the variation in BAT running times for the *5f* and the *5fb* benchmarks, with increase in the number of words in the memory elements (N). The data path width (*s*) also increases as N increases. For example, for the model that has 2^{32} words in the memory elements, *s* is 32. The increase in the verification times is primarily due to the increase in the data path width, as the number of words in the abstracted memories does not change with increase in N, except when N is 2^2 as the actual memories are smaller than their abstractions. Note that we can verify a 32-bit 5 stage pipelined machine in under 1,279 seconds. As far as we know, BAT is the first verification tool to verify a 32-bit 5 stage pipelined machine at the register transfer level. We did not use VCEGAR to check the *5f* and the *5fb* benchmarks, as they are far more complex than the *2f* benchmarks, which VCEGAR was not able to handle.

Three Stage Pipelined Machine (*3c*) Benchmarks: The *3c* bench-

mark is a commitment based refinement theorem [14] for a 3 stage pipeline, which has an instruction memory, a register file, and a data memory. Table 2 shows the BAT verification times for the *3c* benchmarks obtained by increasing the number of words in the memory elements (N). Irrespective of the size of the original memories, the abstracted instruction memory, register file, and data memory have only 5, 8, and 3 words. The increase in the verification times is primarily due to the increase in the data path width. We did not use VCEGAR to check the *3f* benchmarks, as they are far more complex than the *2f* benchmarks, which VCEGAR was not able to handle.

6. CONCLUSIONS AND FUTURE WORK

We have introduced a collection of techniques for automatically reducing memories in bit-level designs. Our experimental evaluation shows that we attain significant improvements over the current state-of-the-art methods, *e.g.*, as far as we know, our method is the first that can automatically prove the correctness of non-trivial pipelined machines with large (32-bit) memories and register files. The key techniques we introduced include a method for abstracting memories that also reduces the size of addresses and a term-rewriting technique that improves the effectiveness of the main algorithm. Our methods are also more general than previous work because they are sound and complete and they allow us to directly compare memories in arbitrary contexts. There are multiple interesting opportunities for future work, including: combining our work with a counterexample guided abstraction-refinement framework; exploiting more advanced term-rewriting techniques; automatically abstracting the data path; improving our translation to SAT; and using compositional reasoning [15] to handle more complex problems.

7. REFERENCES

[1] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In F. Hirose, editor, *Asia South Pacific Design Automation Conference (ASP-DAC'06)*, pages 19–24. IEEE, 2006.

[2] W. R. Bevier, W. A. Hunt, Jr., J. S. Moore, and W. D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[3] P. Bjesse and A. Boralv. DAG-aware circuit compression. In *IEEE/ACM International Conference on Computer Aided Design, 2004 (ICCAD-2004)*, pages 42–49, 2004.

[4] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification (CAV'02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2002.

[5] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.

[6] L. Clark, E. Hoffman, J. Miller, M. Biyani, Y. Liao, S. Strazdus, M.Morrow, K. Velarde, and M. Yarch. An embedded 32-bit microprocessor core for low-power and high-performance applications. *IEEE Journal of Solid-State Circuits*, 36(11):1599–1608, 2001.

[7] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19, July 2001.

[8] M. K. Ganai, A. Gupta, and P. Ashar. Efficient modeling of embedded memories in bounded model checking. In R. Alur and D. Peled, editors, *Computer Aided Verification (CAV'04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 440–452. Springer, 2004.

[9] M. K. Ganai, A. Gupta, and P. Ashar. Verification of embedded memory systems using efficient memory modeling. In *Design, Automation and Test in Europe (DATE'05)*, pages 1096–1101. IEEE Computer Society, 2005.

[10] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04 (Boston, Massachusetts)*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

[11] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*. Springer-Verlag, 1999.

[12] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke. Word level predicate abstraction and refinement for verifying rtl verilog. In W. H. J. Jr., G. Martin, and A. B. Kahng, editors, *Design Automation Conference (DAC'05)*, pages 445–450. ACM, 2005.

[13] S. Lahiri, S. Seshia, and R. Bryant. Modeling and verification of out-of-order microprocessors using UCLID. In *Formal Methods in Computer-Aided Design (FMCAD'02)*, volume 2517 of *LNCS*, pages 142–159. Springer-Verlag, 2002.

[14] P. Manolios and S. Srinivasan. Automatic verification of safety and liveness for XScale-like processor models using WEB-refinements. In *Design Automation and Test in Europe, DATE'04*, 2004.

[15] P. Manolios and S. Srinivasan. A complete compositional reasoning framework for the efficient verification of pipelined machines. In *ACM-IEEE International Conference on Computer Aided Design (ICCAD'05)*, November 2005.

[16] P. Manolios and S. K. Srinivasan. Verification of executable pipelined machines with bit-level interfaces. In *ACM-IEEE International Conference on Computer Aided Design (ICCAD'05)*, November 2005.

[17] L. Ryan. Siege homepage. See URL `http://www.cs.sfu.ca/ ~loryan/personal`.

[18] J. Sawada and W. A. Hunt, Jr. Processor verification with precise exceptions and speculative execution. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 135–146. Springer-Verlag, 1998.

[19] Ultrasparc processors. See URL `http://-www.sun.com/processors/technologies.html`.

[20] M. N. Velev, R. E. Bryant, and A. Jain. Efficient modeling of memory arrays in symbolic simulation. In O. Grumberg, editor, *Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 388–399. Springer, 1997.