Software Workarounds for Hardware Errors: Instruction Patch Synthesis

Tsung-Po Liu, Shuo-Ren Lin, and Jie-Hong R. Jiang, *Member, IEEE*

***Abstract*—Due to the ever-increasing complexity of system design, it becomes not uncommon for some design error escaping all verification efforts and settling in final silicon realization. As hardware-based fixing is much more expensive than software-based fixing, this paper proposes a methodology as a first step towards generating software workarounds for erroneous processor designs. A generic formulation is introduced based on Skolem and Herbrand function extraction from quantified Boolean formula (QBF) solving; reduction techniques are devised to further enhance practicality. Thereby a program can be recompiled at the assembly code level for correct execution on a buggy processor. Experimental results show the feasibility of the proposed method.**

***Index Terms*—Herbrand/Skolem function, processor design, quantified Boolean formula, software workaround.**

I. Introduction

BECAUSE of the ever-increasing complexity of system design, ensuring the correctness of an integrated circuit becomes more and more challenging. Design errors may occur in various design stages, and their corrections may require different degrees of efforts heavily depending on how late in the design phase that they are found. The later a bug is caught, the more it costs to fix. It is not uncommon that errors may unfortunately escape all verification efforts and be caught after tape-out. Fixing such late-found bugs may typically require engineering change orders (ECOs) for late design changes and design re-spins for recreating silicon chips; the buggy chips are useless and need to be discarded. It is often too expensive for most integrated circuit (IC) design companies to afford such fixing as the prices of photomasks, which are used in photolithography for IC fabrication, soar. A more affordable solution is by software-based rectification. Two well-known recent examples include the AMD first-generation Phenom processor in 2007 [1] and the Intel Core 2 Duo processor in 2008 [12]. For the former, customers were advised to turn off the translation look-aside buffer to avoid bugs. For the latter, there are 50-page errata and 75 known bugs, among which 19 required basic input/output system (BIOS) changes, 34 required software changes, and 33 had no known workarounds.

Manuscript received September XX, 20XX; revised November XX, 20XX.
This work was supported in part by the National Science Council under grants NSC 100-2923-E-002-008 and 101-2923-E-002-015-MY2.

Tsung-Po Liu and Shuo-Ren Lin are with the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei 10617, Taiwan (E-mail: ABert0210@gmail.com, ntp890517.ee96@g2.nctu.edu.tw).

J.-H. R. Jiang is with the Department of Electrical Engineering and the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei 10617, Taiwan (E-mail: jhjiang@cc.ee.ntu.edu.tw).

To make post-silicon ECOs more affordable, prior efforts [22], [26] proposed to embed programmable circuitry in processor design to allow hardware patches. Prior work [25] proposed to insert to a processor design some additional control logic, called the semantic guardian, which monitors a subset of the design's internal nodes and switches the system into a safe mode (in which only verified operations are allowed) when an invalidated configuration is encountered. In these methods, the additional logic incurs not only area but also performance overheads. Moreover, not all errors can be fixed by these methods. On the other hand, recent work [10] developed a automatic approach to characterizing error activation conditions, which provide useful information for software developers to modify their programs for correct execution on buggy designs. One application domain of this approach is in embedded systems, where software programs are specially crafted by system developers rather than general users. However, for generic processor designs, it is impractical giving users the error activation information for them to revise their own programs.

In contrast to prior work, this paper exploits a software-based solution and partially automates the workaround process. A method is proposed to generate fixing solutions by recompiling a program such that the resultant assembly code can be correctly executed on an erroneous processor. Thereby, software developers need not revise any part of their source codes for execution on a buggy processor. Because the workarounds can be directly applied for fixing problems in an assembly code, no modification to a high-level language compiler is needed. Moreover, our method helps identify non-trivial workarounds. As an example, consider the replacement of a problematic instruction `abs`, which produces incorrect absolute values for integer arithmetic represented in two's complement. A trivial workaround of the erroneous instruction can be the following assembly code.

```
# problematic instruction: abs r1 r2
01 slt r1 r2 r0  # r0 always holds value 0
02 beq r1 r0 Label1
03 sub r1 r0 r2
04 j Label2
05 Label1:
06 add r1 r0 r2
07 Label2:
```

(The instructions are described in detail in Table I.) The program first checks in lines 1 and 2 whether the value of register `r2` is less than zero. If yes, the program goes to line 3 to subtract the value of `r2` from zero and store the result in

register r1. The program then in line 4 jumps to line 7 and continues the subsequent program execution. Else, the program in line 2 jumps to line 5, stores the value of r2 in r1 in line 6, and then continues the subsequent program execution. In either case, the program executes four instructions to fix the problematic instruction. In contrast, our workaround provides the following concise solution.

```
# problematic instruction: abs r1 r2
01 sra r1 r2 31   # assume word size of 32 bits
02 xor r2 r1 r2
03 sub r1 r2 r1
```

The program uses only three instructions to replace the problematic instruction. In line 1, the sign bit of r2 is copied to all the bits of r1 by instruction sra. In line 2, instruction xor bitwise inverts r2 if the sign bit of r2 is 1, i.e., the value of r2 is a negative integer. Otherwise, r2 remains unchanged. By line 3, if value of r2 is originally a positive integer, then r1 equals r2; otherwise, r1 equals the absolute value of r2 by the representation of two's complement.

We formulate the software workaround problem as a problem of solving quantified Boolean formulas (QBFs) so that the solution, or patch, corresponds to the Skolem function model or Herbrand function countermodel of the underlying QBF. The Skolem/Herbrand functions are obtainable from QBF solvers with certification capability [6], [7]. As QBF solving is PSPACE-complete, reduction techniques and rectification templates are proposed to alleviate the intractability. Experiments on a MIPS-like processor show that workaround solutions to design errors, including those on the DLX bug list [9] and others, can be effectively generated.

The rest of this paper is organized as follows. After essential backgrounds provided in Section II, Section III presents the general solution to the software workaround problem. Reduction techniques and rectification templates are given in Section IV. Experimental results are shown in Section V. Section VI compares related work, and finally Section VII concludes this paper and outlines future work.

II. Preliminaries

A. Pipelined Processor

Processors, which perform some universal arithmetic, logic, and I/O instructions, are the heart of contemporary computing devices. *Pipelining* is a basic technique widely applied in processor design to increase data throughput and the number of instructions executed in a given time period. Nevertheless pipelining along with other techniques sophisticate design tasks and impose serious verification challenges.

The characteristics of a processor is mainly determined by its *instruction set architecture* (ISA) [19], which can be classified into two categories, those for a *reduced instruction set computer* (RISC), such as MIPS and ARM processors, and those for a *complex instruction set computer* (CISC), such as Intel and AMD processors. This paper focuses on the RISC architecture (particularly a MIPS-like ISA) while the proposed methodology is extendable to CISC designs as well. In contrast to CISC, the RISC architecture achieves complex computation through the combination of several fundamental instructions. It makes the design simple and popular in embedded systems.

R-type	opcode	rs	rt	rd
I-type	opcode	rs	rt	immediate
J-type	opcode	address		

Fig. 1. Three types of instruction format.

The instructions of a RISC can be divided into three types: the *register type* (*R-type*), *immediate type* (*I-type*), and *jump type* (*J-type*). For a MIPS processor, all of its instructions are of the same length. In this work, we follow the MIPS format and define a simplified in-house instruction format as shown in Figure 1. Compared to the instruction format in [19], the opcode in our format contains both the opcode and funct of [19]. In addition, the instructions performing shift are classified into I-type, rather than R-type as done in [19], such that the shift amount shamt of [19] will be specified by immediate in our case. These modifications make our instruction format more concise and easier to perform *parametric abstraction*, to be discussed in Section IV, than the conventional format.

Our considered instruction format is shown in Figure 1. R-type is the most complex among the three types of instructions. An R-type instruction involves an operation code opcode, two source registers rs and rt, and one destination register rd. An I-type instruction performs its operation opcode on a constant immediate and a source register rs, and stores the result in the destination register rt. Finally, a J-type instruction contains only two fields, operation opcode and the next target address of the program counter. The supported instructions of our processor are listed in Table I, where the register content at address i of the register file is denoted as $\$i$.

Our in-house processor is a 5-stage pipelined RISC design containing a 32×32 register file and a 20-bit program counter. It is a simplified version of the MIPS architecture and supports fewer instructions. As the block diagram shown in Figure 2, separated by pipeline registers ST_0, ..., ST_3, the pipeline stages include instruction fetch (IF), instruction decode (ID), execute (EX), memory (MEM), and write back (WB). In the figure, "PC" denotes the program counter, which stores the memory address of currently fetched instruction; "PCC" denotes the program counter controller; "REG" denotes the register file. Assuming correct memory implementation, we take the memory I/O including signal MEMDin as the primary input, and signals MEMAout, MEMDout, MEMW, and MEMR as the primary outputs of the design. Moreover, Iin and PC are treated as parts of the primary inputs and primary outputs, respectively.

B. State Transition Relation and Time-Frame Expansion

A pipelined processor can be modeled as a 6-ary transition relation $T(\vec{x}, \vec{r}, \vec{t}, \vec{y}, \vec{r}', \vec{t}')$, which constrains the set of consistent valuations to the variables, including primary inputs $\vec{x}$, current-state variables $\vec{r}$ and $\vec{t}$, primary outputs $\vec{y}$, and next-state variables $\vec{r}'$ and $\vec{t}'$. In terms of the circuit of

TABLE I
INSTRUCTION LIST

Category	Name	Instruction syntax	Meaning	Type
Arithmetic	Add	`add rd rs rt`	`$rd` = `$rs` + `$rt`	R
	Add immediate	`addi rt rs C`	`$rt` = `$rs` + `C`	I
	Subtract	`sub rd rs rt`	`$rd` = `$rs` − `$rt`	R
	Subtract immediate	`subi rt rs C`	`$rt` = `$rs` − `C`	I
	Multiply	`mult rd rs rt`	`$rd` = `$rs` ∗ `$rt`	R
	Multiply immediate	`multi rt rs C`	`$rd` = `$rs` ∗ `C`	I
	Absolute value	`abs rd rs`	`$rd` = \|`$rs`|	R
Logic	And	`and rd rs rt`	`$rd` = `$rs` & `$rt`	R
	And immediate	`andi rt rs C`	`$rt` = `$rs` & `C`	I
	Or	`or rd rs rt`	`$rd` = `$rs` | `$rt`	R
	Or immediate	`ori rt rs C`	`$rt` = `$rs` | `C`	I
	Exclusive or	`xor rd rs rt`	`$rd` = `$rs` ⊕ `$rt`	R
	Exclusive or immediate	`xori rt rs C`	`$rt` = `$rs` ⊕ `C`	I
	Set on less than	`slt rd rs rt`	`$rd` = (`$rs` < `$rt`)	R
Bitwise shift	Shift right logical	`srl rt rs C`	`$rt` = `$rs` >> `C`	I
	Shift left logical	`sll rt rs C`	`$rt` = `$rs` << `C`	I
	Shift right arithmetic	`sra rt rs C`	`$rt` = `$rs` >>> `C`	I
Data transfer	Load word	`lw rt rs C`	`$rt` = Memory[`$rs` + `C`]	I
	Store word	`sw rt rs C`	Memory[`$rs` + `C`] = `$rt`	I
Branch	Branch on equal	`beq rt rs Label`	if (`$rs` == `$rt`) go to `Label`	I
	Jump	`j Label`	go to address `Label`	J

`$i`: the register content at address i of the register file; `C`: an immediate value; `Label`: a label in a program, effectively an address after compilation.

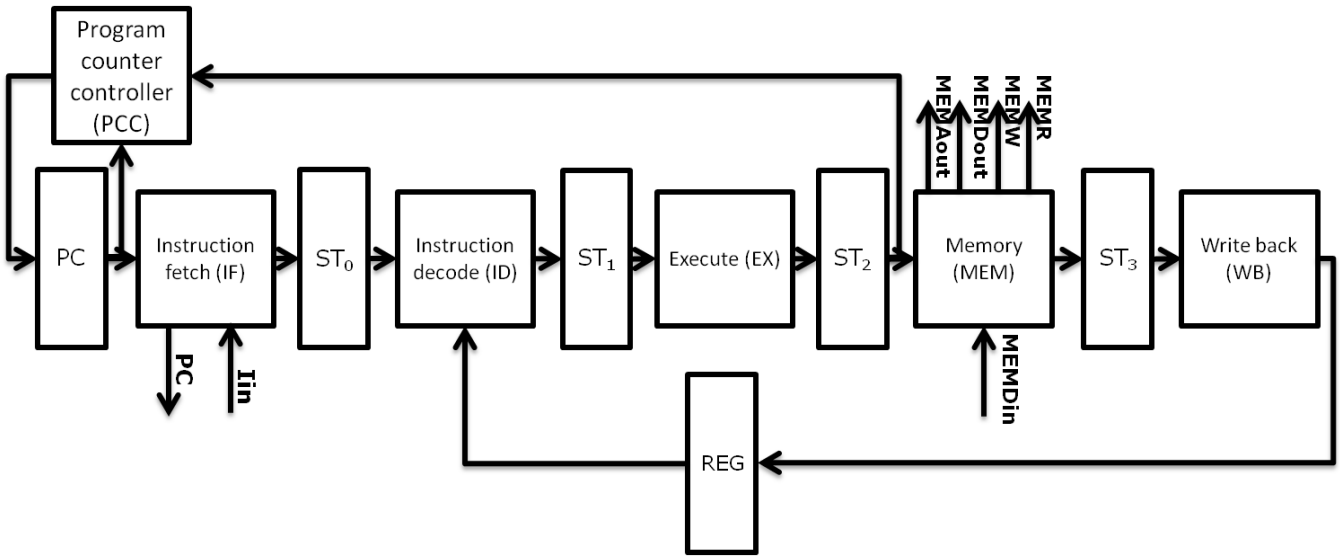

Fig. 2. Block diagram of 5-stage pipelined processor.

Figure 2, variables $\vec{x}$ include {Iin, MemDin}, $\vec{y}$ include {PC, MEMAout, MEMDout, MEMW, MEMR}, $\vec{r}$ and $\vec{r}'$ include the outputs and inputs of REG, respectively, and $\vec{t}$ and $\vec{t}'$ include the outputs and inputs of other registers (including ST_i and PC), respectively. In the sequel, we shall not distinguish a transition relation T and its underlying circuit. Moreover, we shall denote the transition relation of the specification processor as T_S and that of the erroneous processor as T_B. These subscripts of the transition relations should apply to their corresponding variables as well.

Time-frame expansion (TFE) is a well-known technique, e.g., commonly applied in bounded model checking [3]. A k-time-frame expansion unrolls a sequential circuit into a series of k repeated copies of its combinational block. In essence, the expanded circuit characterizes the entire state transition behavior of the sequential circuit in k clock cycles. Note that, for an n-stage pipelined processor, an n-time-frame expansion may be needed to simulate the entire execution of an instruction. In the sequel, we denote a transition relation T and its variables $\vec{v}$ expanded at time-frame i as T^i and $\vec{v}^{\,i}$, respectively.

C. Quantified Boolean Formula

QBFs generalize propositional formulas in incorporating existential and universal quantifiers. Many decision problems can be naturally formulated and succinctly encoded in QBFs. While we introduce only the key backgrounds, the reader is referred to [6], [7] for detailed exposition.

A *quantified Boolean formula* (QBF) Φ over variables $\vec{v} = \{v_1, \ldots, v_k\}$ in the *prenex conjunctive normal form* (PCNF) is of the form

$$Q_1 v_1 \cdots Q_k v_k . \phi,$$

where $Q_1 v_1 \cdots Q_k v_k$, with $Q_i \in \{\exists, \forall\}$ and variables $v_i \neq v_j$ for $i \neq j$, is called the *prefix* and ϕ, a quantifier-free CNF formula in terms of variables $\vec{v}$, is called the *matrix*. We shall assume that a QBF is in PCNF and is totally quantified, i.e., with no free variables.

Given a QBF, the *quantification level* $\ell : \{v_1, \ldots, v_k\} \to \mathbb{N}$ of variable v_i is defined to be the number of quantifier alternations between $\exists$ and $\forall$ from left (i.e., outer) to right (i.e., inner) plus 1. For example, the formula $\exists v_1, \exists v_2, \forall v_3, \exists v_4 . \phi$ has $\ell(v_1) = \ell(v_2) = 1$, $\ell(v_3) = 2$, and $\ell(v_4) = 3$.

A QBF is true (respectively false) if and only if there exist Skolem (respectively Herbrand) functions for the existentially (universally) quantified variables. In particular, the Skolem/Herbrand function of a variable v_i refers to a variable v_j only if $\ell(v_j) < \ell(v_i)$ and v_j is of the quantification type different from that of v_i. In essence, Skolem functions serve as a model to the truth of a QBF; Herbrand functions, on the other hand, serve as a countermodel to the falsity of a QBF. From a game theoretic viewpoint, QBF solving can be seen as a two-player game played by the existential player, who intends to enforce the formula to be true, and the universal player, who intends to enforce the formula to be false. Skolem functions form a winning strategy of the existential player, and Herbrand functions form a winning strategy for the universal player [6], [7]. As a matter of fact, Skolem and Herbrand functions are obtainable from QBF solvers, such as SKIZZO [5], SQUOLEM [13], and QUBE-CERT [18], with certification capability through RESQU conversion [6], [7]. In addition to certification purposes, Skolem and Herbrand functions can be useful in verification and synthesis applications, e.g., [6], [7], [21].

D. Problem Statement

Given a specification processor T_S and its erroneous implementation T_B (assuming they have the same I/O interface and register file), a *workaround* for a program P is a new program P' such that, under the same initial content of the register file, executing P' on T_B and executing P on T_S yield the same final content of the register file. [1]

In reality a computer program is a list of instructions. Assuming instruction `stall` (or null operation `nop`) is existent and correct, then a workaround can be generated by fixing problematic instructions[2] in a program one at a time because they can always be interleaved properly with `stall` insertion to resolve data dependency issues. Effectively, `stall` can flush the data of preceding instructions into REG and push the result of current instruction execution to the next stage. A well known example of data dependency issues in computer architecture is *data hazards*, which arise due to incomplete data processing in the pipeline. Stall insertion is one of common techniques to resolve the hazards. Note that the existence and correctness assumption about `stall` operation should be reasonable as it is the most elementary operation. However stall insertion is not indisputable due to its side effect of slowing down the consequent program execution.

Hence, for each problematic instruction I in the specification ISA, its fixing instruction sequence $I'_1, \ldots, I'_k$ in the implementation ISA is to be derived as the replacement of I in P in order to obtain the workaround program P'. The collection of such mappings from I to $I'_1, \ldots, I'_k$ forms a *mapping function* (or *patch*). We define the *p-instruction mapping problem* as finding the mapping function, called the *p-instruction mapping solution*, with fixing instruction sequences of length at most p. Consequently under the `stall` availability assumption, the workaround problem becomes merely instruction dependent rather than program (instruction-sequence) dependent. That is, stall-insertion avoids potential errors triggered by consecutive execution of instructions. It much simplifies the computation.

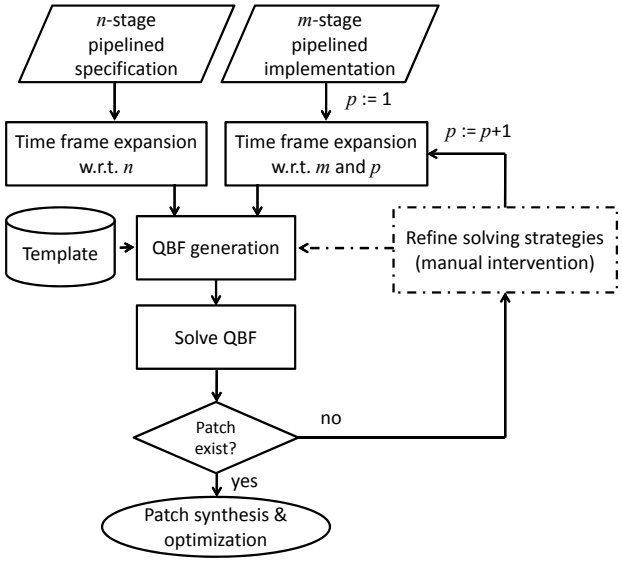

Fig. 3. Computation flow of compiler patch synthesis.

III. GENERIC PATCH SYNTHESIS

This section first presents the workaround synthesis flow, and then details the general QBF formulation and its variants.

A. Overview of Patch Synthesis

Figure 3 sketches the computation flow of compiler patch synthesis. Starting from $p = 1$, the computation checks whether a p-instruction mapping solution exists. If yes, the underlining QBF Φ is true and its Skolem function model (alternatively, the negated QBF $\neg\Phi$ is false and its Herbrand function countermodel) corresponds to the desired patch. Otherwise, p is incremented by 1 and the process repeats until a mapping solution is found or the computation resource is

[1] Note that the patch synthesis problem differs from the conventional program synthesis problem in that the new program should be in terms of the (correct as well as erroneous) instructions of an implemented processor rather than a correct processor.

[2] An instruction in the specification ISA is said *problematic* if its corresponding executions on T_S and T_B yield unequal results.

exhausted. Indeed the larger the value of p is, the harder the corresponding QBF can be evaluated (because by Tseitin's circuit-to-CNF conversion [24] the number of increased variables in the matrix of the QBF due to the increment of p by one is proportional to the circuit size of T_B).

Notice that the flow performs "bounded" synthesis in the sense that the instruction correction is with respect to a length bound. That is, given a length bound p, we test whether it is possible to fix all the considered erroneous instructions using instruction sequences of length up to p. Since the computation does not automatically infer the non-existence of fixes, the loop can continue forever, and often p cannot go too large due to the limiting factor of computing resources in QBF solving. Nevertheless, the computation flow can be guided towards partial fixing of a restricted subset of erroneous instructions when a total fix is not possible. When a single erroneous instruction is considered, the flow of Figure 3 guarantees the search for a patch with a minimum length p.

B. Basic QBF Formulation

Given an n-stage pipelined specification processor and an m-stage pipelined buggy processor, the p-instruction mapping problem, defined in Section II-D, can be formally expressed with the following QBF.

$$\forall \vec{x}_S^0, \exists \vec{x}_B^*, \forall \vec{r}_S^0, \exists \vec{r}^*, \exists \vec{t}^*, \exists \vec{x}_S^*, \exists \vec{x}^*, \exists \vec{y}^*. \quad \phi_T \wedge \phi_\perp \wedge \phi_E \tag{1}$$

where

$$\phi_T = \bigwedge_{i=-n}^{n-1} T_S^i \wedge \bigwedge_{j=-m}^{m+p-2} T_B^j \tag{2}$$

$$\phi_\perp = \bigwedge_{k=-n,k\neq 0}^{n-1} (\vec{x}_S^k = \texttt{nop}) \wedge \bigwedge_{l=-m,l\neq 0,\ldots,p-1}^{m+p-2} (\vec{x}_B^l = \texttt{nop}) \tag{3}$$

$$\phi_E = (\vec{r}_S^0 = \vec{r}_B^0) \wedge (\vec{r}_S^n = \vec{r}_B^{m+p-1}) \tag{4}$$

for

$$\begin{aligned}
\vec{x}_B^* &= (\vec{x}_B^0, \ldots, \vec{x}_B^{p-1}) \\
\vec{r}^* &= (\vec{r}_S^{-n}, \ldots, \vec{r}_S^{-1}, \vec{r}_S^1, \ldots, \vec{r}_S^n, \vec{r}_B^{-m}, \ldots, \vec{r}_B^{m+p-1}) \\
\vec{t}^* &= (\vec{t}_S^{-n}, \ldots, \vec{t}_S^n, \vec{t}_B^{-m}, \ldots, \vec{t}_B^{m+p-1}) \\
\vec{x}_S^* &= (\vec{x}_S^{-n}, \ldots, \vec{x}_S^{-1}, \vec{x}_S^1, \ldots, \vec{x}_S^{n-1}) \\
\vec{x}^* &= (\vec{x}_B^{-m}, \ldots, \vec{x}_B^{-1}, \vec{x}_B^p, \ldots, \vec{x}_B^{m+p-2}) \\
\vec{y}^* &= (\vec{y}_S^{-n}, \ldots, \vec{y}_S^{n-1}, \vec{y}_B^{-m}, \ldots, \vec{y}_B^{m+p-2})
\end{aligned}$$

Note that n and m need not be the same. For example, the specification can be a non-pipelined processor and the implementation can be a pipelined version. Also notice that variables $\vec{y}^*$ essentially play no role in Formula (1), and can be removed by cone-of-influence reduction. Moreover as to be explained, the Skolem functions of variables $\vec{x}_B^*$ correspond to the desired patches.

To understand the QBF, the formula $\phi_T \wedge \phi_\perp \wedge \phi_E$ can be intuitively depicted with the circuit shown in Figure 4, where some of the pins are omitted. To search a p-instruction mapping solution, the specification and buggy processors are forwardly unrolled n and $p+m-1$ time-frames, respectively, starting from the reference time index 0. By pipeline flushing [4], the execution result of the instruction at $\vec{x}_S^0$ in the specification processor settles at time n after `nop` insertion for $\vec{x}_S^1, \ldots, \vec{x}_S^{n-1}$; similarly, the result of the instruction sequence at $\vec{x}_B^0, \ldots, \vec{x}_B^p$ in the buggy processor settles at time $m+p-1$ after `nop` insertion for $\vec{x}_B^p, \ldots, \vec{x}_B^{m+p-2}$. On the other hand, to ensure proper initial values imposed on $\vec{t}_S^0$ for the specification processor and on $\vec{t}_B^0$ for the buggy processor, the time-frames of these two processors are backwardly unrolled n and m time-frames, respectively, with all instruction inputs filled with `nop`. Such time-frame expansion and `nop` insertion are expressed by ϕ_T and $\phi_\perp$, respectively. Finally, under the equivalence $\vec{r}_S^0 = \vec{r}_B^0$ of initial register files, the two results are then compared for equivalence by asserting ϕ_E.

Notice that the above backward unrolling is unnecessary when the `nop` instruction is implemented in such a way that the values of $\vec{t}_S^0$ and $\vec{t}_B^0$ can be uniquely determined regardless of the values of $\vec{r}_S^{-n}$ and $\vec{t}_S^{-n}$ and the values of $\vec{r}_B^{-m}$ and $\vec{t}_B^{-m}$, respectively. In this case, the values of $\vec{t}_S^0$ and $\vec{t}_B^0$ can be directly substituted with their respective logic values induced by the backward unrolling. In our implementation, we take advantage of this simplification for QBF solving.

The quantification structure of the prefix of Formula (1) is due to the requirement that, for every instruction $\vec{x}_S^0$ of the specification processor, there exists a fixing instruction sequence $\vec{x}_B^*$ of the buggy processor regardless of the initial content of the register file $\vec{r}_B^0$. If Formula (1) is true, Skolem functions for variables $\vec{x}_B^*$ exist and refer only to variables $\vec{x}_S^0$. They correspond to the desired mapping solution. Otherwise, no p-instruction mapping solution exists provided that Skolem functions should not depend on $\vec{r}_B^0$.

Notice that, as long as Formula (1) is true, the corresponding Skolem functions of the existential variables $\vec{x}_B^*$ provide valid mappings for all instructions in the ISA at once, no matter whether an instruction is problematic or not. (The identity mapping is obtained for correct instructions.) In fact, if the scope of problematic instructions can be narrowed down to a few instructions, then instruction-specific cofactoring (to be detailed in Section III-C1) can be applied to enhance the solving efficiency. Furthermore, the QBF formulation not only can use both correct and incorrect instructions in the ISA for error correction, but also can exploit instructions not in the original ISA. In the latter case, the error fixing should be performed in the machine code level rather than the assembly code level, since the instructions not in ISA are not expressible with assembly codes.

As shown in [7], for a given true QBF whose matrix is readily in a circuit form, deriving the Herbrand functions from its negation is sometimes easier than directly deriving its Skolem functions. (Note that Herbrand functions to the negated QBF are Skolem functions to the original QBF.) As was suggested in [6], [7] about QBF application on Boolean relation determination, Formula (1) can be negated by Tseitin's

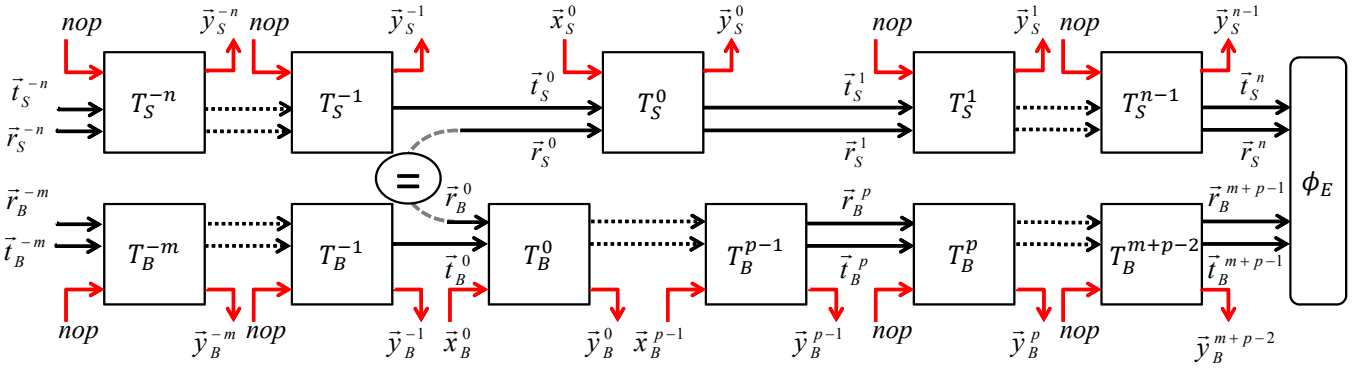

Fig. 4. Circuit for patch synthesis, where some pins are omitted.

circuit-to-CNF conversion [24] as

$$\exists \vec{x}_S^0, \forall \vec{x}_B^*, \exists \vec{r}_S^0, \exists \vec{r}^*, \exists \vec{t}^*, \exists \vec{x}_S^*, \exists \vec{x}^*, \exists \vec{y}^*. \\ \phi_T \wedge \phi_\perp \wedge (\vec{r}_S^0 = \vec{r}_B^0) \wedge (\vec{r}_S^n \neq \vec{r}_B^{m+p-1}) \tag{5}$$

which has only 3 quantification levels, in contrast to the 4 levels of Formula (1). Since the complexity of QBF solving is influenced by the number of quantification levels, this negated formula is more favorable. To be justified in Section V, Formula (5) is usually easier to solve than Formula (1).

C. Modified QBF Formulation

The previous basic formulation is generic in searching data-independent workarounds for arbitrary design errors. Due to its generality, QBF solving lacks effective strategies in reducing search space and is hardly scalable to designs with normal data widths and register file sizes. Nevertheless, by exploiting error information, the QBF may be adjusted to consider different situations and to ease the burden of QBF solving. On the other hand, some design errors may have no data-independent, but only data-dependent, workarounds. In these cases, the previous QBF formulation needs further modification.

1) Instruction-Specific Cofactoring: Since every problematic instruction in the ISA can be handled separately, we may restrict QBF solving with respect to a subset of problematic instructions by *cofactoring*, which substitutes logic values for variables in a formula. Thereby the search space can be substantially reduced for each divided QBF solving. For a design with a few errors, the approach can be particularly effective. Moreover, the binary codes unused in the ISA can be blocked to avoid wasteful search.

A useful trick is to cofactor on `opcode` only such that the searched workaround can be desirably independent of `rs`, `rt`, and `rd`. Precisely, the QBF formula can be modified as follows.

$$\exists \vec{x}_B^*, \forall \vec{r}_S^0, \exists \vec{r}^*, \exists \vec{t}^*, \exists \vec{x}_S^*, \exists \vec{x}^*, \exists \vec{y}^*. \\ (\phi_T \wedge \phi_\perp \wedge \phi_E)_{\vec{x}_S^0 = \texttt{opcode}} \tag{6}$$

where the subscript denotes the cofactor of $\vec{x}_S^0$ with respect to some `opcode`.

Note that, even if complete workarounds for all erroneous instructions do not exist, partial workarounds can still be used to fix programs whose instructions are rectifiable.

2) Mapping Function Simplification: When Formula (1) is easily satisfiable, that is, with many Skolem function models, it is sometimes possible to seek for simple mapping functions depending only on a subset of variables $\vec{x}_S^0$. Recall that the Skolem function of an existentially quantified variable may refer to the universally quantified variables with quantification levels smaller than that of the existentially quantified variable. We may exploit the fact that moving an existential quantifier in the prefix of a QBF outward strengthens the formula. That is, a model to the resultant QBF is also a model to the original QBF. Moreover, the Skolem functions of the strengthened QBF, if it is true, refer to fewer variables than those of the original QBF. Therefore the corresponding new mapping functions can potentially be simpler.

Specifically, Formula (1) can be strengthened to

$$\forall \vec{x_1}_S^0, \exists \vec{x}_B^*, \forall \vec{x_2}_S^0 \forall \vec{r}_S^0, \exists \vec{r}^*, \exists \vec{t}^*, \exists \vec{x}_S^*, \exists \vec{x}^*, \exists \vec{y}^*. \\ \phi_T \wedge \phi_\perp \wedge \phi_E \tag{7}$$

where variables $\vec{x}_S^0$ are split into two subsets $\vec{x_1}_S^0$ and $\vec{x_2}_S^0$. As a result, the Skolem functions for variables $\vec{x}_B^*$, if they exist, will refer to variables $\vec{x_1}_S^0$ only. Notice however that moving existentially quantified variables outward as in Formula (7) might turn a true QBF into a false one. In our current implementation, we did not exploit such optimization, but it could be helpful.

3) Data-Dependent Rectification: Formula (1) only exploits mapping functions that are data independent, that is, the Skolem functions of $\vec{x}_B^*$ are independent of $\vec{r}_S^0$. However there are design errors whose workarounds must be data dependent. In such cases, Formula (1) should be modified in a way that the data bits essential to workarounds need to be moved to a quantification level less than that of $\vec{x}_B^*$. Specifically, we have the new formula

$$\forall \vec{x}_S^0, \forall \vec{r_1}_S^0, \exists \vec{x}_B^*, \forall \vec{r_2}_S^0, \exists \vec{r}^*, \exists \vec{t}^*, \exists \vec{x}_S^*, \exists \vec{x}^*, \exists \vec{y}^*. \\ \phi_T \wedge \phi_\perp \wedge \phi_E \tag{8}$$

where variables $\vec{r}_S$ are split into two parts, $\vec{r_1}_S^0$, which workaround should depend on, and $\vec{r_2}_S^0$, which workaround should be independent of. By this modification, Skolem functions may depend on $\vec{r_1}_S^0$.

When converting the Skolem functions of Formula (8) back to rectification instructions, an if-then-else (ITE) style instruction sequence may be needed. Essentially, the data

dependence (if-condition) controls the branching to either the then-branch or the else-branch.

4) Equality Constraint Relaxation: The equality constraint ϕ_E of Formula (1) asserts that the final equivalence must hold for the entire register file. In certain circumstances we may relax such a strong condition and maintain the equivalence for part of the register file. Specifically, Formula (1) can be relaxed to

$$\begin{array}{c}\forall \vec{x}_S^0, \exists \vec{x}_B^*, \forall \vec{r}_S^0, \exists \vec{r}^*, \exists \vec{t}^*, \exists \vec{x}_S^*, \exists \vec{x}^*, \exists \vec{y}^*. \\ \phi_T \wedge \phi_\perp \wedge (\vec{r}_S^0 = \vec{r}_B^0) \wedge (\vec{r}_S^n[0:i] = \vec{r}_B^{m+p-1}[0:i])\end{array} \quad (9)$$

where $\vec{r}_S^n[0:i]$ represents the part of the register file whose address range over 0 to i.

The relaxation is feasible, for example, when a system designer intends to preserve part of the register file for special use only, but not accessible to normal programs. In fact, preserving some register file space for workaround synthesis may be useful and sometimes even necessary. It not only strengthens the rectification power, but also simplifies QBF because of the cone of influence reduction.

IV. Practical Workarounds

Due to the intrinsic complexity of QBF solving, modern QBF solvers remain hardly scalable to solving instances of industrial sizes although impressive progress has been made recently. In contrast to the generic methods of Section III-C, we present reduction techniques specific to design styles or circuit structures to enhance practicality.

A. Parametric Abstraction

In high-level design using hardware description languages, *data widths* and *register file sizes* can be parameterized for effective design space exploration to search an optimal solution satisfying various design constraints. In a parameterized design, its data width and register file size can be specified in terms of variable parameters rather than fixed constants. This parameterization can be exploited for datapath abstraction. For a parameterized design, its intended actual data width and register file size can be too large and verifying its correctness can be formidable. Intuitively the same design with a reduced data width and register file size (referred to as an *abstract design*) could much resemble the design of original size (referred to as a *concrete design*). Errors and their corrections found in the abstract design may well reflect errors and their corrections in the concrete design; after all, both designs share the very same code. The similarity between the abstract and concrete designs may be exploited for verification reduction. We call such a reduction method as *parametric abstraction*.

It should be noted that parametric abstraction guarantees neither soundness nor completeness in catching and fixing errors. For example, if a concrete design is subject to some erroneous manual circuit tuning, then its abstract design cannot faithfully reflect errors. However, when design errors are irrelevant to data widths and register file sizes, parametric abstraction can be effective.

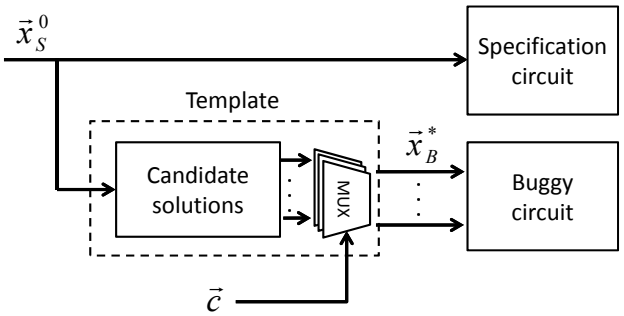

Fig. 5. Hardware template for patch synthesis.

B. Error Localization

When design errors locate only at or before the k^{th} stage of an m-stage pipelined processor ($k \leq m$), a sufficient condition for error correction is to ensure the specification and buggy circuits always compute the same result at the output of the k^{th} stage. As a result, only k time-frame expansion is needed since the equivalence constraint ϕ_E of Formula (4) can be simplified to check the equivalence of the outputs at the k^{th} stage. That is, we modify

$$\phi_E = (\vec{r}_S^0 = \vec{r}_B^0) \wedge (\vec{r}_S^k = \vec{r}_B^k) \wedge (\vec{t}_S^k = \vec{t}_B^k), \quad (10)$$

where we assume the specification and buggy circuits have the same set of pipeline stages and search for a 1-instruction mapping solution such that the outputs at the k^{th} stages are equivalent. The shortened time-frame expansion may simplify the QBFs to be solved.

C. Template-Based Solving

The knowledge about design errors can be transformed into *templates* to instruct QBF solvers searching for structured solutions and effectively reducing search space. The transformation relies heavily on a manual process however. In theory, templates are not necessary for workaround synthesis; in practice, they are essential in penetrating computation barriers.

We define *parametrically independent templates* as those that are not specific to particular design parameters, such as register file size and data width. Therefore with parametrically independent templates, the mapping solutions to parametrically abstracted design can potentially be applied to the original design. Note that the derivation of templates much relies on manual intervention. Knowledge about the errors of a design is crucial to derive effective templates.

A template can be expressed in a circuit form and integrated with the buggy design to guide instruction rectification by providing choices and restricting search space. As shown in Figure 5, the template constrains the instruction inputs $\vec{x}_B^*$ of the buggy design with some specific options, which are selected through multiplexers fed by some functions in terms of the instruction inputs $\vec{x}_S^0$ of the specification circuit. Such functions map from the valuations of $\vec{x}_S^0$ to the valuations of $\vec{x}_B^*$. The choices of candidate mapping solutions can then be made through the decision on the values of the control inputs $\vec{c}$ of the multiplexers. The template-based patch synthesis

corresponds to solving the following QBF

$$\forall \vec{x}_S^0, \exists \vec{c}, \forall \vec{r}_S^0, \exists \vec{x}_B^*, \exists \vec{r}^*, \exists \vec{t}^*, \exists \vec{x}_S^*, \exists \vec{x}^*, \exists \vec{y}^*. \quad \phi_T \wedge \phi_\perp \wedge \phi_E \wedge \phi_S \tag{11}$$

where ϕ_S are the constraints imposed by the template circuitry.

Although template design is mainly a manual process of trial by error, it can be assisted by the QBF solving formulation. Suppose a template cannot provide workarounds for all design errors. So the resultant QBF (Formula (1) with template modification) should be false and the Herbrand functions for variables $\vec{x}_S^0$ are simply constants, which correspond to an instruction that has not been fixed by the current template. The information can then be used to modify the template.

To illustrate, consider the example that `rd` and `rs` are mistakenly switched in the IF pipeline stage of a processor design. Suppose that under some improper template assumption the resulting QBF Formula (11) is false. So the Herbrand functions of $\vec{x}_S^0$ are derivable. As variables $\vec{x}_S^0$ are quantified outermost, their Herbrand functions do not refer to any existentially quantified variables and are essentially constants. Suppose these constants together correspond to the infeasible instruction `add r0 r1 r2` (recall that register `r0` always holds value 0). It reveals that `rd` is somehow problematic. One might guess that `rd` is incorrect in the instructions and provide a template, say, permuting the variables $\vec{x}_S^0$. In this case, a correct solution can be found in the next QBF solving.

D. Guidelines for Patch Synthesis

We summarize the aforementioned reduction techniques by the following guidelines in patch synthesis. Error localization is first performed by combinational equivalence checking on the original design without parametric abstraction. Erroneous instructions are then identified. Depending on how many instructions are erroneous, we may decide if instruction-specific cofactoring should be applied. If only a few instructions are erroneous, instruction-specific cofactoring can be effective. By solving the underlying QBF under parametric abstraction, we refine, if necessary, solving strategies, including building/modifying templates, applying data-dependent rectification, relaxing equality constraints, and other methods. The flow of Figure 3 is applied.

V. Experimental Results

The main computation flow of instruction patch synthesis was implemented in the C language within the Berkeley ABC system [8], whereas QBFs were evaluated with solver DepQBF [15] and Skolem/Herbrand functions were extracted with QBFcert [17], which embeds the ResQu conversion of [6], [7]. The experiments were conducted on a Linux machine with Xeon 3.3 GHz CPU and 64 GB RAM.

An in-house 5-stage pipelined MIPS design was created for case study. Its data width and register-file size were parameterized ranging from 2 to 32 bits and from 4 to 32, respectively, for the parametric abstraction purpose. As described in Table II, design errors were introduced based on the processor bug suite [9] of the University of Michigan. Its correct and erroneous versions with respect to a given parameter were then synthesized with Synopsys Design Compiler for experiment. In the following experiments, unless otherwise noted, the error correction computation was conducted on a parametrically abstracted design with 2-bit data width and a register file of size 4. (The benchmark instances are available at [23].) A workaround solution to the erroneous abstract design was then mapped back to the erroneous original design with 32-bit data width and a register file of size 32.

TABLE II
Error Description.

BUG	Description
BUG1	`rd` exchanged with `rs` in IF stage.
BUG2	write address increased by 1 in WB stage.
BUG3	MEMAout increased by 1 in MEM stage.†
BUG4	ALU control input bitwise inverted in EX stage.
BUG5	`sub` ignores 1st operand.
BUG6	`opcode` decode error in ID stage.‡
BUG7	2's complement implemented as 1's complement.
BUG8	`add` resultant increases 1 when the sign bit of `$rt` is 1.
BUG9	`add` performs `sub` when the sign bit of `$rt` is 1.
BUG10	`$rt` is 0 when `rt` is even.
BUG11	`srl` same as `sra`.
BUG12	`abs` is problematic.
BUG13	Mix of BUG3 and BUG5.
BUG14	Mix of BUG1, BUG6, and BUG7.
BUG15	Mix of BUG2 and BUG4
BUG16	Mix of BUG1, BUG4, and BUG9.

†MEMAout is the write address of external data memory.
‡Right-rotating shift 3 bits in `opocde`.

Table III shows the results of two sets of experiments: one without practical reduction (denoted "base QBF") and the other with practical reduction (denoted "practicality-enhanced QBF"). The former experiment considered basic QBFs of Formula (1) without any modification and reduction; the latter considered QBFs that were reduced with the aforementioned techniques in Section IV. Both experiments were conducted on parametrically abstracted designs. For each buggy design, its error is shown in Column 1, the numbers of expanded time-frames (denoted "#fram") and the maximum p values for p-instruction mapping (denoted "#inst") are shown in Columns 2 and 7, the numbers of QBF variables in Columns 3 and 8 (the numbers in the parentheses show the numbers of variables for the first three quantification levels in order), the numbers of QBF clauses in Columns 4 and 9, the runtimes of QBF solving in Columns 5 and 10, the certificate extraction times in Columns 6 and 11. Note that, for base QBFs, #inst = (#fram $-$ the number of pipeline stages (i.e., 5) $+$ 1); for practicality-enhanced QBFs, #fram can be less than 5 due to error localization.

The effectiveness of the proposed reduction techniques can be seen by comparing the CPU times in Columns 5 and 10. The reduction techniques were selectively applied according to error characteristics. (The acquirement of error knowledge remains a manual process in our experiment.) For errors that locate at some particular pipeline stages, such as BUG1, BUG3, BUG4, and BUG6, the error localization technique was applied to reduce the number of expanded time-frames. For errors that affect a few instructions, such as BUG8, BUG9, BUG11,

TABLE III
RESULTS ON WORKAROUND COMPUTATION

	Base QBF					Practicality-Enhanced QBF				
BUG	#fram/#inst	#var	#cls	solve (sec)	extract (sec)	#fram/#inst	#var	#cls	solve (sec)	extract (sec)
BUG1	5/1	181(11/11/8)	582	66.52	351.45	1/1	44(5/11/14)	96	0.02	0.01
BUG2	5/1	193(11/11/8)	589	67.48	251.68	5/1	68(5/11/6)	162	2.49	5.22
BUG3	5/1	181(11/11/8)	582	56.43	276.11	4/1	104(5/11/14)	320	2.14	4.21
BUG4	5/1	190(11/11/8)	580	72.63	280.83	3/1	182(5/11/14)	632	2.55	7.31
BUG5	7/3	363(11/33/8)	1281	379.85	170.32	7/3	407(5/33/14)	1544	121.74	198.22
BUG6	5/1	177(11/11/8)	582	59.94	272.72	2/1	50(5/11/6)	120	2.01	3.51
BUG7	6/2	280(11/22/8)	960	119.88	355.41	6/2	305(5/22/14)	1094	0.74	2.87
BUG8	6/2	274(11/22/8)	962	77.58(false)	NA	6/2	301(15/22/4)	1070	53.40	92.01
BUG9	6/2	279(11/22/8)	971	128.32	299.67	6/2	301(9/22/10)	1070	69.49	72.45
BUG10	6/2	262(11/22/8)	904	7.55(false)	NA	6/2	239(5/22/14)	842	145.56	82.63
BUG11	8/4	494(11/44/8)	1811	$>$ 100000	NA	8/4	439(0/44/14)	1596	3.01	2.26
BUG12	7/3	365(11/33/8)	1285	359.78	174.19	7/3	409(5/33/14)	1561	143.54	210.04
BUG13	7/3	363(11/33/8)	1310	408.92	183.35	7/3	407(5/33/14)	1575	182.86	274.15
BUG14	6/2	271(11/22/8)	967	131.49	510.46	6/2	294(5/22/14)	1069	82.66	84.67
BUG15	5/1	190(11/11/8)	655	82.24	274.54	5/1	215(5/11/14)	692	3.52	0.53
BUG16	6/2	271(11/22/8)	955	128.89	348.57	6/2	297(5/22/14)	1061	85.79	108.41

and BUG12, the instruction-specific cofactoring technique is particularly effective. Comparing Columns 3 and 4 to Columns 8 and 9, most formula sizes are reduced as a result of practicality enhancement. There are cases, such as BUG4, BUG5, BUG7, BUG9, and BUG12, where formula sizes increase due to the usage of templates. Nevertheless, as templates effectively reduce search space, QBF evaluation was made effective. On the other hand, the base QBFs of BUG8 and BUG10 are false under their specified time-frame expansions, and require enlargement of rectification power using data-dependent rectification and equality constraint relaxation, respectively, for workaround synthesis. For the mixed errors BUG13 to BUG16, their runtimes are close to their worst runtimes among their respective ingredient bugs. One exception is BUG14, which took 82.66 seconds for solving, whereas its most time-consuming ingredient bug BUG6 took only 2.01 seconds. The arisen inefficiency is due to the inapplicability of reduction techniques under the error combination. All of the derived Skolem functions for parametrically abstracted designs are extendable for error fixing for the original 32-bit design.

When QBF certificates are of concern, they vary in size case by case to some extent. Nevertheless our empirical results suggested that the certificate size of a practicality-enhanced QBF is typically about $1/4$ the certificate size of its baseline counterpart. It should be noted, however, that QBF certificate sizes do not correlate directly to the cost of patches.

TABLE IV
RESULTS OF USING NEGATED QBF

	Base QBF		Practicality-Enhanced QBF	
BUG	solve (sec)	extract (sec)	solve (sec)	extract (sec)
BUG1	4.29	6.31	0.01	0.01
BUG2	3.73	5.44	0.51	1.42
BUG3	3.19	5.02	0.01	0.01
BUG4	5.01	8.73	0.62	1.47
BUG6	6.26	7.24	0.02	0.58
BUG15	5.34	8.15	0.94	2.71

Table IV shows the runtimes in solving negated QBFs of Formula (5) for bugs with 1-instruction mapping. Compared to their non-negated counterparts in Table III, negated QBFs for 1-instruction mapping instances are easier to solve due to the reduction of quantification levels from four to three. However, for multi-instruction mapping instances, the increase of variables at the second quantification level makes negated QBFs less effective to solve than their non-negated counterparts.

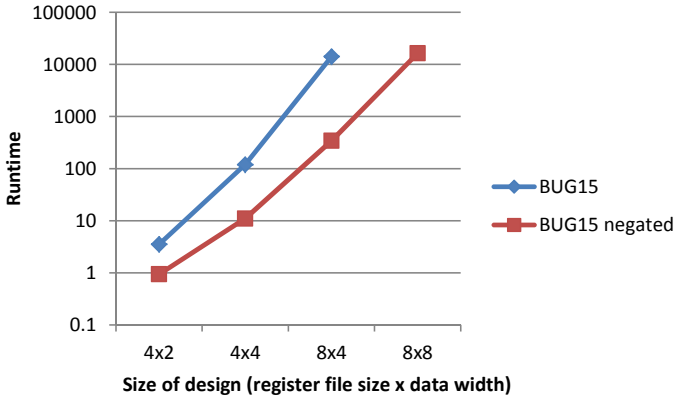

Fig. 6. Runtime vs. design size.

Parametric abstraction is indispensable to overcome expensive QBF evaluation as suggested by Figure 6, where the runtimes of solving Formulas (1) and (5) with practicality enhancement for BUG15 under different combinations of register file sizes and data widths. The exponential growth of runtime with respect to design size shows the infeasibility of QBF solving for the problem of original size. Nevertheless, the following experiments suggested that, when error localization is possible, QBF solving is applicable to instances of practical sizes even without parametric abstraction and other reduction techniques.

We studied the scalability of QBF solving for errors localized within different pipeline stages. The results are shown in Figures 7 and 8, where the x-axis corresponds to the design size (in bits) in terms of the product of register file size and data width, and the y-axis corresponds to CPU time (in seconds). In particular, BUG1, BUG2, BUG3, BUG4, and BUG6 were experimented with, whose errors are located at the first, fifth, fourth, third, and second pipeline stages, respectively. In the experiment, only the error localization technique is applied without other simplification methods. Note that, since

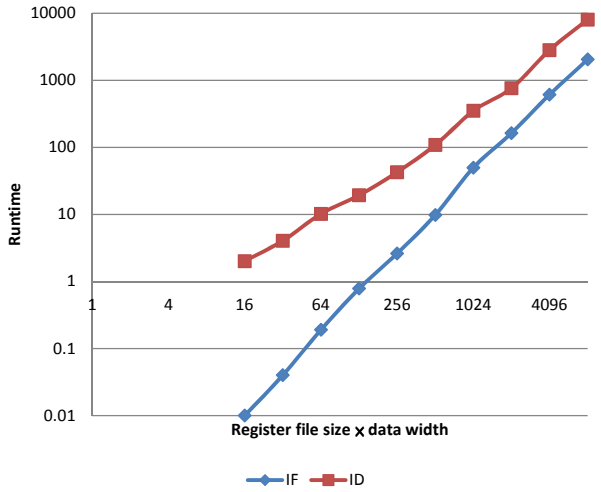

Fig. 7. Scalability analysis for errors localized at the first (i.e., IF) and second (i.e., ID) pipeline stages.

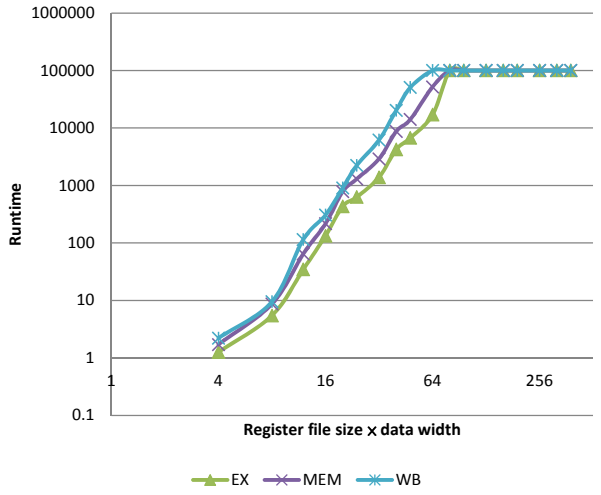

Fig. 8. Scalability analysis for errors localized at the third (i.e., EX), fourth (i.e., MEM), and fifth (i.e., WB) pipeline stages.

only error localization was applied, the QBF solving searches mapping solutions for all instructions in the ISA. The studied bugs are representative in that, by empirical experience, when the size of a design is specified, the runtime is mainly affected by the number of considered instructions.

As can be seen from Figure 7, for errors localizable within the first two pipeline stages, QBF solving is scalable to practical design sizes (with register file size up to 128 and word length up to 64 bits) despite the fact that the runtime increases exponentially with respect to the design size. On the other hand, as shown in Figure 8, for errors locating at the third, fourth, and fifth pipeline stages, QBF solving becomes inefficient as the circuits involve arithmetic logic units. (In fact, arithmetic logic units are not only problematic for QBF solvers, but also challenging even for SAT solvers.) In these cases, reduction techniques are crucial for feasible patch synthesis. Orthogonally, future advances of QBF solvers may push the solvability limit forward.

The results of program patching are shown in Tables V and VI, where in Column 2 #EI denotes the number of erroneous instructions among the 21 instructions shown in Table I, Columns 3, 7, 11 show the number of program execution cycles, Columns 5, 9, 13 show the number of lines of code after program patching, and Columns 4, 6, 8, 10, 12, 14 show the ratio of the corresponding number of a patched program to that of the original program. Three benchmark programs were considered, including a program for bubble sorting, a program for Fibonacci number generation, and a program composed of random instructions. Specifically, the bubble sort program was executed on sorting ten numbers under their worst case initial order; the Fibonacci program was executed on generating the 47^{th} Fibonacci number; and the random program consisted of 128 randomly generated instructions without branching ones. In these tables, clock cycle counts are compared between the original program running on the correct design and a patched program running on a buggy design; the numbers of lines of code (LOC) are compared between the original program and a patched program. As a matter of fact, the number of erroneous instructions used in a program mainly determines the increases of program execution cycles and lines of codes. (This fact cannot be directly seen from the tables because the number of erroneous instructions activated by a program cannot be statically determined.) The execution cycle and program size are increased due to multi-instruction fixing as well as stall insertion. Table V shows the results without stall insertion (assuming no errors resulting from pipelined execution), whereas Table VI shows the results with stall insertion between every pair of instructions (conservatively assuming errors may happen due to pipelined execution without attempting any stall minimization). Because the underlying processor has five pipeline stages, four `nop` instructions are needed to interleave a pair of instructions. Therefore, by ignoring the issue of multi-instruction fixing, the program size can increase four times in the worst case due to `nop` insertion. Also notice that, since the first two programs, namely bubble sorting and Fibonacci number generation, involve loops, which result in the increase of execution cycles in comparison with the lines of code. So an erroneous instruction that appears in a loop may result in more execution cycles than one that appears out of loops.

As mentioned in Section III-B, the QBF formulation may allow patch solutions not included in ISA. Indeed among the 21 single-instruction patch solutions to the 21 erroneous instructions of `BUG1`, six of them are not present in ISA (although solutions using instructions in ISA do exist, which can be found by the same QBF formulation but with variables $\vec{x}_B^*$ being constrained to solutions in ISA). This result suggests that our proposed formulation can potentially find highly non-trivial patch solutions.

VI. Related Work

Among related prior work on correcting processor errors, the efforts closest to ours include [26], [22], where programmable hardware is integrated into processor designs for error rectification. The error patching mechanisms of prior and our methods are fundamentally different. Prior methods trade hardware resources for error rectifiability in the processor design stage; we generate software patches after chip

TABLE V
STATISTICS OF PATCHED PROGRAMS (WITHOUT STALL INSERTION)

Design	#EI	Bubble Sort				Fibonacci Number				Random			
		#cycle	ratio	#LOC	ratio	#cycle	ratio	#LOC	ratio	#cycle	ratio	#LOC	ratio
Original	0	913	1.00	37	1.00	335	1.00	13	1.00	128	1.00	128	1.00
BUG1	21	913	1.00	37	1.00	335	1.00	13	1.00	128	1.00	128	1.00
BUG2	18	913	1.00	37	1.00	335	1.00	13	1.00	128	1.00	128	1.00
BUG3	1	913	1.00	37	1.00	335	1.00	13	1.00	128	1.00	128	1.00
BUG4	19	913	1.00	37	1.00	335	1.00	13	1.00	128	1.00	128	1.00
BUG5	1	935	1.02	41	1.11	523	1.56	17	1.31	144	1.13	144	1.13
BUG6	21	913	1.00	37	1.00	335	1.00	13	1.00	128	1.00	128	1.00
BUG7	2	924	1.01	39	1.05	429	1.28	15	1.15	144	1.13	144	1.13
BUG8	1	1229	1.35	49	1.32	479	1.43	19	1.46	136	1.06	136	1.06
BUG9	1	1229	1.35	49	1.32	479	1.43	19	1.46	136	1.06	136	1.06
BUG10	9	1611	1.76	60	1.62	621	1.85	23	1.77	176	1.38	176	1.38
BUG11	1	913	1.00	37	1.00	335	1.00	13	1.00	152	1.19	152	1.19
BUG12	1	913	1.00	37	1.00	335	1.00	13	1.00	130	1.02	130	1.02
BUG13	2	1207	1.32	45	1.22	525	1.57	19	1.46	144	1.13	144	1.13
BUG14	21	1826	2.00	74	2.00	670	2.00	26	2.00	256	2.00	256	2.00
BUG15	19	913	1.00	37	1.00	335	1.00	13	1.00	128	1.00	128	1.00
BUG16	21	1826	2.00	74	2.00	670	2.00	26	2.00	256	2.00	256	2.00

TABLE VI
STATISTICS OF PATCHED PROGRAMS (WITH STALL INSERTION)

Design	#EI	Bubble Sort				Fibonacci Number				Random			
		#cycle	ratio	#LOC	ratio	#cycle	ratio	#LOC	ratio	#cycle	ratio	#LOC	ratio
Original	0	913	1.00	37	1.00	335	1.00	13	1.00	128	1.00	128	1.00
BUG1	21	4565	5.00	185	5.00	1675	5.00	65	5.00	640	5.00	640	5.00
BUG2	18	4565	5.00	185	5.00	1675	5.00	65	5.00	640	5.00	640	5.00
BUG3	1	4565	5.00	185	5.00	1675	5.00	65	5.00	640	5.00	640	5.00
BUG4	19	4565	5.00	185	5.00	1675	5.00	65	5.00	640	5.00	640	5.00
BUG5	1	4587	5.02	189	5.11	1863	5.56	69	5.31	656	5.13	656	5.13
BUG6	21	4565	5.00	185	5.00	1675	5.00	65	5.00	640	5.00	640	5.00
BUG7	2	4576	5.01	187	5.05	1769	5.28	67	5.15	656	5.13	656	5.13
BUG8	1	4881	5.35	197	5.32	1819	5.43	71	5.46	648	5.06	648	5.06
BUG9	1	4881	5.35	197	5.32	1819	5.43	71	5.46	648	5.16	648	5.16
BUG10	9	5263	5.76	208	5.62	1961	5.85	75	5.77	688	5.38	688	5.38
BUG11	1	4565	5.00	185	5.00	1675	5.00	65	5.00	664	5.19	664	5.19
BUG12	1	4565	5.00	185	5.00	1675	5.00	65	5.00	642	5.02	642	5.02
BUG13	2	4859	5.32	193	5.22	1865	5.57	71	5.46	656	5.13	656	5.13
BUG14	21	5478	6.00	222	6.00	2010	6.00	78	6.00	768	6.00	768	6.00
BUG15	19	4565	5.00	185	5.00	1675	5.00	65	5.00	640	5.00	640	5.00
BUG16	21	5478	6.00	222	6.00	2010	6.00	78	6.00	768	6.00	768	6.00

fabrication without hardware, but performance, overhead. In terms of QBF formulation for design repairing, prior work [21] aims at correcting errors directly within logic circuits whereas we focus on avoiding processor errors in terms of software workarounds.

Our patch synthesis is related to program optimization, e.g., [16], [2], in that in both problems a new program is to be derived from a reference program. In program optimization, the optimized and reference programs must be equivalent under execution upon the same underlying processor. In patch thesis, the transformed and reference programs are equivalent under executions upon their respective buggy and ideal machines. Prior program optimization techniques can potentially be adopted for patch optimization. On the other hand, there are recent attentions on program synthesis, e.g., [11], [14], [20]. In program synthesis, missing code fragments conforming to some specification are to be derived. In [11], [20], the problem reduces to solving quantified formulas (possibly beyond a purely propositional domain), which have quantification structures similar to Formula (5).

Our patch synthesis differs from program optimization and synthesis in that processor circuitry often has to be taken into account. Because erroneous, in addition to correct, instructions can be used for workarounds, circuit constraints (i.e., ϕ_T, $\phi_\perp$, and ϕ_E of Formula (1)) have to be modeled at the bit level. This difference makes our formulas not easily lifted to the word level for potentially more effective solving, such as using satisfiability-modulo-theories (SMT) solvers. (Note that if only correct instructions are to be used for patching, then the circuit constraints are unnecessary, and the instruction patch synthesis problem can be treated as a standard program synthesis problem.)

VII. CONCLUSION

This paper has proposed a software approach to circumvent processor errors. A generic QBF formulation and its variants enhanced for practical applications have been presented. Modern QBF solvers capable of generating certificates can be exploited for workaround synthesis. The synthesized patch can then be used for automatic program recompilation. Case studies on an in-house five-stage pipelined MIPS design have shown the feasibility of the proposed method. The proposed

framework showed a first step towards achieving the goal of software workaround for hardware errors.

Even though current computation is limited by QBF solving and thus requires human intervention to reduce computational complexity, the proposed method can still be valuable in assisting engineers to spot nontrivial fixes (especially for embedded system applications, where programs are written by system developers). Future advances of QBF solvers may further automate the error correction process.

While the present work focused on instruction patch generation, how to optimize patched programs with minimal stall insertions remains future work. Existing techniques in automatic code synthesis and optimization could be useful in patch synthesis. Moreover since not all design errors (such as the single stack-at fault) can be fixed with parametric abstraction, other effective abstraction techniques await development. It is also important to characterize what kinds of errors are fixable/unfixable under our framework.

ACKNOWLEDGMENTS

The authors are grateful to Roderick Bloem, Chia-Wei Chang, Georg Hofferek, and Robert Könighofer for helpful discussions.

REFERENCES

[1] AMD. *Revision Guide for AMD Family 10h Processors*, Rev. 3.90, Advanced Micro Devices, Inc., 2012.

[2] S. Bansal and A. Aiken. Automatic Generation of Peephole Superoptimizers. In *Proc. Int'l Conf. on Architectual Support for Programming Languages and Operating Systems* (ASPLOS), pp. 394-403, 2006.

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y.-S. Zhu. Symbolic Model Checking without BDDs. In *Proc. Int'l Conf. on Tools and Algorithms for Construction and Analysis of Systems* (TACAS), pp. 193-207, 1999.

[4] J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In *Proc. Int'l Conf. on Computer Aided Verification* (CAV), pp. 68-80, 1994.

[5] M. Benedetti. Evaluating QBFs via Symbolic Skolemization. In *Proc. Int'l Conf. on Logic for Programming, Artificial Intelligence and Reasoning* (LPAR), pp. 285-300, 2004.

[6] V. Balabanov and J.-H. R. Jiang. Resolution Proofs and Skolem Functions in QBF Evaluation and Applications. In *Proc. Int'l Conf. on Computer Aided Verification* (CAV), pp. 149-164, 2011.

[7] V. Balabanov and J.-H. R. Jiang. Unified QBF Certification and its Applications. *Formal Methods in System Design*, 41(1):45-65, 2012.

[8] Berkeley Logic Synthesis and Verification Group. *ABC: A system for sequential synthesis and verification*. Online: `http://www.eecs.berkeley.edu/~alanmi/abc/`

[9] *Bug UnderGround: University of Michigan microprocessor bug suite*. Oneline: `http://bug.eecs.umich.edu/`

[10] C.-W. Chang, H.-Z. Chou, K.-H. Chang, J.-H. R. Jiang, C.-N. J. Liu, C.-H. Hsiao, and S.-Y. Kuo. Constraint Generation for Software-based Post-silicon Bug Masking with Scalable Resynthesis Technique for Constraint Optimization. In *Proc. Int'l Symp. on Quality Electronic Design* (ISQED), pp. 1-8, 2011.

[11] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of Loop-Free Programs. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation* (PLDI), pp. 62-73, 2011.

[12] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Document Number: 252046-035, Intel Corporation, 2012.

[13] T. Jussila, A. Biere, C. Sinz, D. Kröning, and C. M. Wintersteiger. A First Step Towards a Unified Proof Checker for QBF. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing* (SAT), pp. 201-214, 2007.

[14] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Oracle-Guided Component-Based Program Synthesis. In *Proc. Int'l. Conf. on Software Engineering* (ICSE), pp. 215-224, 2010.

[15] F. Lonsing and A. Biere. DepQBF: A Dependency-aware QBF Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:71-76, 2010.

[16] H. Massalin. Superoptimizer – A Look at the Smallest Program. In *Proc. Int'l Conf. on Architectual Support for Programming Languages and Operating Systems* (ASPLOS), pp. 122-126, 1987.

[17] A. Niemetz, M. Preiner, F. Lonsing, M. Seidl, and A. Biere. Resolution-Based Certificate Extraction for QBF. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing* (SAT), pp. 430-435, 2012.

[18] M. Narizzano, C. Peschiera, L. Pulina, and A. Tacchella. Evaluating and Certifying QBFs: A Comparison of State-of-the-Art Tools. *AI Commun.*, 22(4):191–210, Dec. 2009.

[19] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, 4th edition, 2008.

[20] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial Sketching for Finite Programs. In *Proc. Int'l Conf. on Architectual Support for Programming Languages and Operating Systems* (ASPLOS), pp. 404-415, 2006.

[21] S. Staber and R. Bloem. Fault Localization and Correction with QBF. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing* (SAT), pp. 355-368, 2007.

[22] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas. Patching Processor Design Errors with Programmable Hardware. *IEEE Micro*, 27(1):12-25, 2007.

[23] *Software Workaround Benchmarks*. Oneline: `http://alcom.ee.ntu.edu.tw/tools.htm`

[24] G. Tseitin. On the Complexity of Derivation in Propositional Calculus. *Studies in Constructive Mathematics and Mathematical Logic*, pp. 466-483, 1970.

[25] I. Wagner and V. Bertacco. Engineering Trust with Semantic Guardians. In *Proc. Design, Automation, and Test in Europe (DATE)*, pp. 743-748, 2007.

[26] I. Wagner, V. Bertacco, and T. Austin. Sielding against Design Flaws with Field Repairable Control Logic. In *Proc. Design Automation Conference* (DAC), pp. 344-347, 2006.

Tsung-Po Liu received the B.S. degree in Electrical Engineering from National Tsing Hua University, Hsinchu, Taiwan, in 2010, and the M.S. degree in Electronics Engineering from National Taiwan University, Taipei, Taiwan, in 2012. He is currently fulfilling his mandatory military service. His current research interests include software workarounds, formal verification, and logic synthesis.

Shuo-Ren Lin received the B.S. and M.S. degrees in Electronics Engineering from National Chiao Tung University, Hsinchu, Taiwan, in 2011, and from National Taiwan University, Taipei, Taiwan, in 2013, respectively. He is currently fulfilling his mandatory military service. His current research interests include logic synthesis, test pattern generation, and quantified Boolean formula certification.

PLACE
PHOTO
HERE

Jie-Hong R. Jiang (S'03-M'04) received the B.S. and M.S. degrees in Electronics Engineering from National Chiao Tung University, Hsinchu, Taiwan, in 1996 and 1998, respectively, and the Ph.D. degree in Electrical Engineering and Computer Sciences from the University of California, Berkeley, in 2004. During his compulsory military service, from 1998 to 2000, he was a Second Lieutenant with the Air Force, R.O.C. In 2005, he joined the faculty of the Department of Electrical Engineering at National Taiwan University, where he is a professor. His research interests include logic synthesis, formal verification, and computation models of physical and biological systems.

Dr. Jiang is a member of Phi Tau Phi and the Association for Computing Machinery (ACM).