Formal Verification of Backward Compatibility of Microcode*

Tamarah Arons[1], Elad Elster[2], Limor Fix[1], Sela Mador-Haim[3], Michael Mishaeli[2], Jonathan Shalev[1], Eli Singerman[1], Andreas Tiemeyer[1], Moshe Y. Vardi[4], and Lenore D. Zuck[5]

[1] Design Technology, Intel Corporation, e-mail firstname.lastname@intel.com
[2] Mobile Micro-processor Group, Intel Corp., e-mail firstname.lastname@intel.com
[3] Technion, Israel, e-mail selam@cs.technion.ac.il
[4] Rice University, e-mail vardi@cs.rice.edu
[5] University of Illinois at Chicago, e-mail lenore@cs.uic.edu

Abstract. Microcode is used to facilitate new technologies in Intel CPU designs. A critical requirement is that new designs be backwardly compatible with legacy code when new functionalities are disabled. Several features distinguish microcode from other software systems, such as: interaction with the external environment, sensitivity to exceptions, and the complexity of instructions. This work describes the ideas behind MICROFORMAL, a technology for fully automated formal verification of functional backward compatibility of microcode.

1 Introduction

The performance and functionality requirements from current CISC (Complex Instruction Set Computing) CPUs mandate a dual-layer design. While the external (architectural) appearance is that of a CISC CPU, the internal mechanisms employ RISC (Reduced Instruction Set Computing) methodologies. A microcode layer captures the architectural intent of the processor and translates between the architecture and the hardware layer, which contains the microarchitectural implementation details [Sta02].

Improvements in successive generations of CPU designs are measured not only in terms of performance improvements, but also in functional enhancements, such as security (e.g., LaGrande technology), hardware virtualization (e.g., Vanderpool technology), and the like. A significant portion of the implementation of such new functionalities is done in microcode. Thus, a mature architecture such as the IA32 is accompanied by a large base of microcode, which is essentially very low-level software.

When adding new functionality to an existing CPU design, the validation team faces a major verification challenge of ensuring backward functional compatibility, to guarantee that legacy software in the marketplace works without changes on the new CPU. Since functionality enhancement is often implemented in microcode, this verification challenge has to be met at the microcode level.

* This research was supported in part by SRC grant 2004-TJ-1256, NSF grants CCF-0456163, CCR-9988322, CCR-0124077, CCR-0311326, IIS-9908435, IIS-9978135, EIA-0086264, ANI-0216467, BSF grant 9800096, and by Texas ATP grant 003604-0058-2003.

In this paper we describe MICROFORMAL, design technology being developed at Intel to automatically and formally verify the functional backward compatibility of microcode.

There is scant work on formal verification at the microcode level. Several papers describe efforts to prove that the microarchitectural level of a microprocessor implements correctly the instruction-set architecture [Cyr93,SM96,SH02]. The approach of these works is that of computer-aided deduction, while we are seeking a more automated solution. Furthermore, our focus is on functional compatibility between successive generations of microcode, which we refer to as *source* and *target*. Closer to our work is the automated equivalence verification approach, as applied, for example, in [CHRF00,HUK00,FH02,CKY03]. These works aim to prove equivalence of low-level code with higher-level code, using symbolic execution and automated decision procedures. (We note that while equivalence verification is a common verification technique nowadays [HC98], its industrial application is generally limited to hardware.)

Automated equivalence verification has also been successfully applied to *translation validation*, which is an automated verification technique for showing that target code, generated by an automatic translator (e.g., an optimizing compiler) accurately translates source code. Rather than proving the correctness of the *translator* itself, the translation-validation approach attempts to prove the correctness of each *translation* separately, by proving the equivalence of the source and target codes, using symbolic execution and automated decision procedures [PSS98,Nec00,ZPFG03].

Our work differs from these works in several aspects. The first difference is that our focus is not on equivalence, but rather on backward compatibility. Nevertheless, we show that equivalence verification techniques are applicable here. We define *backward compatibility of target with source* as equivalence under restrictions that disable the new functionalities. We view this as an important conceptual contribution, as the problem of backward compatibility under functionality enhancements is quite common. For example, when adding features to the telephony systems, users who do not subscribe to the new features or have instruments that do not support the new features, should notice no change when the system is upgraded.

A second, and more challenging difference is our focus on microcode. While microcode is software, it is extremely "machine aware", as execution of microcode is heavily dependent on the microarchitectural state. Previous applications of equivalence verification to software, for example, in translation validation, typically consider *closed systems* that do not interact with their environment (except for initial and final I/O). In order to make this approach applicable to microcode, we have to make the dependence of the microcode on the microarchitectural state, as well as the effect that the microcode has on this state, explicit. Furthermore, we assume that the microarchitectural state does not essentially change during the execution of the microcode, unless explicitly modified by the microcode. This assumption matches the intuition of microcode designers. (An alternative approach would be to consider microcode as a *reactive system* [HP85]. This would make the verification task considerably harder, as defining equivalence of programs in fully open environments is rather nontrivial [AHKV98].)

In addition to considering interactions with the environment, at the microcode level, executions leading to exceptions are considered normal and can terminate in many dif-

ferent states. Thus, in our framework, we need to deal with exceptions in a rather elaborate way and define equivalence to mean that the rich exception structure is preserved. In contrast, prior works did not report on handling exceptions, cf. [Nec00]. The need to model interactions with the environment and executions that terminate in exceptions poses a nontrivial challenge to the application of automated equivalence verification to the problem of microcode backward compatibility. Furthermore, the need to apply the technique to industrial problems of today's scale and complexity poses another formidable challenge.

Our approach is also considerably different from recent approaches to formal property verification of software, where both theory and practice have been gaining momentum. Recent tools include SLAM [BR02], BLAST [HJMS03] and others. A common feature of these works is the focus on abstracting data away as much as possible. In microcode, the data types are simpler than those in higher-level software, but control flow and data flow are tightly integrated, so standard abstraction techniques, such as predicate abstraction, are not easily applicable.

One challenge in developing generic tools for microcode verification is the complexity and variability of the instruction sets of modern microprocessor designs. Rather than work directly with microcode, our tool uses an intermediate representation language (IRL) that is general and is suitable for a wide family of low-level languages. The translation between the actual code and IRL is accomplished by means of *templates*, where each template consists of IRL code that is operationally equivalent to a microinstruction. IRL can also be used to provide microcode with formal semantics and drive microcode emulators.

Once both source and target are translated into IRL, they are processed by MICROFORMAL, which constructs the verification conditions (VCs). The VCs are checked by a validity checker. Their validity establishes that the target is backwardly compatible with the source. Failure to establish validity produces a counterexample in the form of input values that causes the behaviors of source and target to diverge.

A major obstacle to the success of our methodology is the complexity of checking the VCs. Since we are dealing with low-level code, the correctness is expressed in terms of bit vectors. Unfortunately, to date, there are no efficient validity checkers that handle bit vectors, and we are forced to reduce bit vectors to bit level and use a propositional validity checker (i.e., a SAT solver). Another problem is the size of the VCs: even for relatively small microprograms (several hundred lines), the size of the VCs is often prohibitively large, i.e., beyond the capacity of most validity checkers. We use various simplification techniques, such as decomposition and symbolic pruning, to make verification practically feasible.

2 Modeling Microcode

Microprograms are essentially low-level, machine-oriented programs with a sequential control flow. The basic, atomic statements of microprograms are microinstructions. These are implemented in hardware and executed by various units of the CPU. The basic data types of microprograms are registers (architectural, control and temporary).

A microinstruction can be thought of as a function that (typically) gets two register arguments, performs some computation and assigns the result to a third register.

The control flow of microprograms is facilitated by `jump-to-label` instructions, where the labels appear in the program or indicate a call to an external procedure. It is possible to have indirect jumps, in which case the target label resides in a register and is known only at run time.

The semantics of microinstructions involves both the variables that appear in the instructions and several other auxiliary variables that reflect the status of the hardware. In addition to the explicit effect of microprograms on their hardware environment via microinstructions, microprograms also have implicit interaction with hardware by reading/setting various shared machine-state variables (e.g., memory, special control bits for signaling microarchitectural events, etc.). The latter is used for governing the microarchitectural state and is mostly modeled as side-effects (not visible in the microprogram source code) of specific microinstructions.

The first challenge in applying formal verification to microcode is defining a suitable intermediate representation. We need a way to fully capture the functional behavior of microprograms, including the microinstructions they employ. To that end we have introduced a new modeling language, which we call IRL– Intermediate Representation Language. IRL is expressive enough to describe the behavior of microprograms and their interaction with the hardware environment at the "right" abstraction level, yet its sequential semantics is simple enough to reason about with formal tools.

IRL is a simple programming language that has bits and bit vectors as its basic data types. IRL basic statements are conditional assignments and `goto`s. Vector expressions in IRL are generated by applying a rich set of bit-vector operations (e.g. logical, arithmetic, shift, concatenation and sub-vector extraction operations) to bit-vector arguments. IRL has the following characteristics:

1. Simple, easy to read and understand with a well-defined semantics;
2. Generic, extendible, and maintainable. It is a formalism that is not tied up to the microcode language of a specific CPU;
3. Explicit. That is, all operations of a microprogram can be explicitly represented, with no implicit side effects;
4. Can be the target of a compiler from native microcode.

To make it convenient to (automatically) translate microprograms to IRL and to bridge the gap between native microcode and IRL, we coupled IRL with a *template* mechanism by which each microinstruction has a corresponding IRL template. The template signature represents the formal arguments of a microinstruction and its body is a sequence of (plain) IRL statements that compute the effect of the microinstruction and store the result in the designated argument. In addition, each template's body also reflects the side effects of a microinstruction computation by updating the relevant auxiliary variables.

The template language includes convenient abstract data types, such as structures and enumeration, and coding aids such as *if-then-else* and case statements. These constructs make the code easier to read and maintain. During translation these constructs are transformed into basic IRL.

Fig. 1 presents a small microcode-like example that illustrates some basic features of microcode. There, a value is read from a location in memory determined by the `memory_read` parameters (address and offset) and the result added to another value. Registers, as used in the example, are bitvectors of width 64. The memory is an array `array[32][64] memory`. That is, an address space of 32 bits is mapped to entries of 64 bits. The `zeroFlag` variable is a a single bit zero flag.

```
BEGIN_FLOW(example) {
  reg1 := memory_read(reg2, reg3);
  reg4 := add(reg1, reg5);
};
```

Fig. 1. A simple program

Each of the two operations are defined by a template. The `add` template is described in Fig. 2. The template has three formal parameters, corresponding to the two input registers and the target register of the microinstruction (Fig. 1). The first parameter of a template is the target variable, which is instantiated with the variable on the left-hand side of the assignment (`reg4` in our case). Note that a side effect of the add microinstruction— setting `zeroFlag`— is specified explicitly in the template. (For simplicity, we ignore the possibility of add overflow.)

```
template uop_add(register result,
 register src1, register src2) {
 result := src1 + src2;
 zeroFlag := (result = 0);
};
```

Fig. 2. A template for `add`

The `memory_read` microinstruction (Fig. 3) is somewhat more complex and includes a possible exception. The address is calculated as `tmp_address + offset`. If this is out of the memory address range of 32 bits, then an `address_overflow` exception is signalled. Exceptions are a normal part of microprograms, and are modeled as executions at the end of which various parameters are checked. If no exception occurs, the memory contents at this address are placed in register result and the system bit `found_valid_address` is set to true. Such "side-effects" are typical of microinstructions – an intrinsic part of their functionality is that they read and set global system bits. In this example the microcode is translated to the (basic) IRL program of Fig. 4.

Note that the `address_overflow` exit has a parameter of type bit[32]. In general, an exit has parameters defining the *observables* at this exit. Similarly, observables have to be defined also for the `entry` and for the `end` exit. When defining backward compatibility we need to ensure that, when run under the same conditions, the source and target microprograms would reach the same type of exit with the same observable values. In contrast, there are variables whose values are ignored or lost at the program

```
exception address_overflow(bit[32]);
template uop_memory_read(register result,
                register tmp_address, register offset) {
  TMP0 := tmp_address + offset;
  if (TMP0 > 0xFFFFFFFF)
     exit address_overflow(TMP0[63:32]);
  result := memory[TMP0[31:0]];
  zeroFlag := (result = 0);
  found_valid_address := 1;
};
```

Fig. 3. Simple `memory_read` template

```
1. entry (reg1, reg2, reg3, reg4, reg5, memory, pc);
2. TMP0 := reg2 + reg3;
3. (TMP0 > 0xFFFFFFFF) ? exit address_overflow(TMP0[63:32]);
4. reg1 := memory[TMP0[31:0]];
5. zeroFlag := (reg1 = 0);
6. found_valid_address := 1;
7. reg4 := reg1 + reg5;
8. zeroFlag := (reg4 = 0);
9. exit end (found_valid_address, reg1, reg2, memory);
```

Fig. 4. Basic IRL for the original program: Numbered statements

exit and we have no interest in comparing them, e.g. temporary registers. Furthermore, different expressions may be relevant at different exits – in our example the value of `TMP0` is relevant at the `address_overflow` exit, but not at the `end` exit, where its "temporary" value is discarded. Defining the correct observables for the various exits is not a trivial exercise – too few observables may result in real mismatches being missed, too many observables may result in false negatives when comparing expressions which need not be equal. In practice, the observability expressions are built and stabilized over time; starting with the microcode validator's first approximation, the expressions are tuned during several iterations of false negatives and false positives. Doing so requires a deep understanding of the microcode, and may involve consultation with the the microcode designers.

We now consider an extended version of this system in which memory addresses can also include bases (Fig. 5). The extended `memory_read` microinstruction takes an extra base parameter, and is described in Fig. 6. The IRL corresponding to the microcode of Fig. 5, using the extended read, is in Fig. 7.

3 The Formal Model

In this section we introduce our formal model of computation, *exit-differentiated transitions systems*, which defines the semantics of IRL programs similar to the way transi-

```
BEGIN_FLOW(example) {
 reg1 := memory_read(reg2, reg3, reg6);
 reg4 := add(reg1, reg5);
};
```

Fig. 5. Microcode example II: An extended version

```
template uop_memory_read(register result,
    register tmp_address, register base, register offset) {
  TMP0 := tmp_address + offset;
  TMP0 := TMP0 + base;
  if (TMP0 > 0xFFFFFFFF)
     exit address_overflow(TMP0[63:32]);
  result := memory[TMP0[31:0]];
  zeroFlag := (result = 0);
  found_valid_address := 1;
};
```

Fig. 6. A `memory_read` for the extended version

tion systems are used to give semantics to "Simple Programming Language" (SPL) in [MP95]. We then give a formal definition to the notion of restricted equivalence, termed *compatibility*, of two IRL programs. We use the program of Fig. 4 as a running example.

A microprogram usually allows for several types of exits, such as the `end` exit and the `address_overflow` exit in our example. At each exit type we may be interested in different observables. For example, at the `address_overflow` exit we care only about the values of the overflow bits, while at the `end` exit we care about the values of `found_valid_address`, `reg1` and `reg4`. When comparing two microprograms for equivalence or backward compatibility, we would expect that when the two terminate at corresponding exits, the values of the corresponding observables would match. Similarly, when comparing the two microprograms, we make some assumptions about their entry conditions, such as that registers have the same initial values. We refer to the entry and exit points as *doors*. Two microprograms are termed equivalent if whenever their entry values match, they exit through the same doors with matching exit values.

By "matching values" we usually mean that two variables have the same value. However, it is sometimes the case that a comparison between variables is insufficient, and a more involved comparison must be made e.g., a sub-vector or a conditional expression like "if (cond) then $a[7:0]$ else $b[9:2]$". We therefore compare values that are defined by *well-typed observation functions* over the system variables.

In order to capture the formal semantics of equivalence with respect to doors, we define *exit-differentiated transition systems*, EDTS, an extension of the symbolic transition systems of [MP95] to systems with differentiated exit types. An EDTS $S = \langle V, \Delta, \Theta, \rho \rangle$ consists of:

– V: A finite set of typed *system variables*. The set V always includes the program counter `pc`. A V*-state* is a type-consistent interpretation of V. We denote the symbolic value of variable v in state s by $s.v$;

```
1. entry (reg1, reg2, reg3, reg4, reg5, reg6, memory, pc);
2  TMP0 := reg2 + reg3;
3. TMP0 := TMP0 + reg6;
4. (TMP0 > 0xFFFFFFFF) ? exit address_overflow(TMP0[63:32]);
5. reg1 := memory[TMP0[31:0]];
6. zeroFlag := (reg1 = 0);
7. found_valid_address := 1;
8. reg4 := reg1 + reg5;
9. zeroFlag := (reg4 = 0);
10. exit end (found_valid_address, reg1, reg2, memory);
```

Fig. 7. Basic IRL for the updated program: Numbered statements

- Δ: A finite set of *doors*. Each door is associated with an an observation tuple $\mathcal{O}_\delta$ of well-typed observation functions over V. In addition, we have a partial function ν from pc to Δ, where $\nu(\mathtt{pc}) = \delta$ if pc is associated with door δ and $\perp$ if it is associated with no door. The final state of every computation is associated with some door. There is a single distinguished entry door, entry, all other doors are exit doors;
- Θ: An *initial condition* characterizing the initial states of the system. For every state s, $\Theta(s) \longrightarrow \nu(s.\mathtt{pc}) = \mathtt{entry}$;
- $\rho(V, V')$: A *transition relation* relating a state to its possible successors;

The semantics of IRL programs is defined in terms of EDTS in a straightforward way (cf. semantics of SPL in terms of symbolic transition systems [MP95]). For example, the EDTS associated with the IRL of Fig. 4 is:

$$
\begin{aligned}
V &= \{\mathtt{pc}, \mathtt{TMP0}, \mathtt{reg1}, \ldots, \mathtt{reg5}, \mathtt{found\_valid\_address}, \mathtt{memory}\} \\
\rho &= \bigvee_{\ell=1..7} \rho_\ell \\
\Theta &= (\mathtt{pc} = 1) \\
\Delta &= \{\mathtt{entry}, \mathtt{address\_overflow}, \mathtt{end}\} \text{ with associated observation tuples} \\
&\quad \mathcal{O}_{\mathtt{entry}} : \quad (\mathtt{reg1}, \ldots, \mathtt{reg5}, \mathtt{memory}) \\
&\quad \mathcal{O}_{\mathtt{address\_overflow}} : (\mathtt{TMP0}[63:32]), \\
&\quad \mathcal{O}_{\mathtt{end}} : \quad (\mathtt{found\_valid\_address}, \mathtt{reg1}, \mathtt{reg4}, \mathtt{memory})
\end{aligned}
$$

where each ρ_ℓ describes the transition associated with $\mathtt{pc} = \ell$.

As discussed earlier, execution of microcode is heavily dependent on the microarchitectural state. The translation of microcode to IRL makes this dependence fully explicit. While microcode has invisible side effects, these are fully exposed in the IRL. This, together with the assumption that the microarchitectural state does not change during the execution of the microcode unless explicitly modified by the microcode, makes our IRL programs deterministic; given an initial assignment to the variables, behavior is completely determined. Making microcode dependence on the microarchitectural state fully explicit enables us also to handle *busy-waiting loops*. In such loops, the microcode

waits for the hardware to provide a certain readiness signal. Since loops pose a challange to symbolic simulators, we abstract busy-waiting loops, which are quite common, by adding an auxiliary bit for each such loop. This bit "predicts" whether the readiness signal will be provided. When this bit is on, the loop is entered and immediately exited, and when the bit is off, the loop is entered but not exited.

A computation of an EDTS is a maximal sequence of states $\sigma : s_0, s_1, \ldots$ starting with a state that satisfies the initial condition, such that every two consecutive states are related by the transition relation. For $j = 1, 2$, let $P^j = \langle V^j, \Theta^j, \rho^j, \Delta^j \rangle$ be an EDTS. We say that P^1 and P^2 are *comparable* if for every $\delta \in \Delta^1 \cap \Delta^2$, the corresponding observation tuples in both programs are of the same arity and type.

We are interested in notions of compatibility. Often we find that a new system has the same functionality as the old system, plus new functionality. We use *restrictions* to constrain the set of initial values and thus compare only the legacy functionality of the new system. We define systems P^1 and P^2 to be $\mathcal{R}$*-compatible* with respect to a restriction $\mathcal{R}$, if they are comparable and, and under the restriction of $\mathcal{R}$, every execution in the same environment (with matching entry observables) ends in the same exit state, with matching exit observables. Restrictions are defined as predicates over $V^1 \cup V^2$. Our main focus is on terminating computations; the only non-terminating computations that we consider are those resulting from getting stuck in infinite busy-wait loops, for which we verify that both source and target microprograms diverge under the same conditions.

To illustrate the notion of compatibility, consider again our example codes. Let P^1 be the EDTS of the "source" program of Fig. 4, and let P^2 be the EDTS of the "target" program of Fig. 7. The two systems are clearly comparable. It is easy to see that these two flows do not reach the same exit under all conditions. For example, if initially $\texttt{reg2} + \texttt{reg3} = \texttt{0xFFFFFFFF}$, P^1 exits at `end`. If, in addition, `reg6` is initially non-zero, P^2 exits at `address_overflow`. Under the restriction that in system P^2 `reg6` is initially zero (since `reg6` is not among P^1's variables, there is no danger of restricting P^1's behavior by this restriction), the two are compatible. This is a typical example of a restriction in the new system that disables the new functionality to ensure backward compatibility.

4 Simulation and Verification

We use symbolic simulation to compute the effect of the program on a symbolic initial state. As defined in Sec. 3, a state is a type-consistent interpretation of the variables. Variable values are symbolic expressions over the set of initial values and constants that appear in the program. In our example, in the initial state s_0, before any statements have executed, all variables have symbolic values. After statement 2 is simulated, the value of `TMP0` is the expression $s_0.\texttt{reg2} + s_0.\texttt{reg3}$ over these symbolic values.

If microprograms were merely lists of assignments, this would suffice. However, they also include conditional and jump statements. Since the state is symbolic, it is at times impossible to determine the evaluation of a condition. For example, at statement 3, the program either exits or continues according to the evaluation of the test $(\texttt{TMP0} > \texttt{0xFFFFFFFF})$, which is equivalent to evaluating whether $s_0.\texttt{reg2} + s_0.\texttt{reg3}$

$>$ 0xFFFFFFFF; it is easy to find assignments to s_0.reg2 and s_0.reg3 under which the condition is true, and assignments under which it is false. Both are feasible, under different initial conditions.

We use *symbolic paths* to store, for every control-distinct (as opposed to data-distinct) path the current symbolic variable values, the path condition, and the exit door, once it has been found. Formally, a symbolic path π is a triple (c, s, δ), where c is the condition ensuring that this path is taken, s is a state containing the relevant symbolic values of variables, and δ is an exit door if such is reached, $\perp$ otherwise. The path conditions of a program are exhaustive (their disjuntion is one) and mutually exclusive (no two path conditions can be satisfied simultaneously).

For example, after statement 2 has executed, there is a single symbolic path (TRUE, s_2, $\perp$) where the value of TMP0 in s_2 is s_0.reg2 + s_0.reg3. When statement 3 is executed, the symbolic path is split into two:

$$(s_0.\texttt{reg2} + s_0.\texttt{reg3} > \texttt{0xFFFFFFFF}, s_2, \texttt{address\_overflow})$$

and

$$(s_0.\texttt{reg2} + s_0.\texttt{reg3} \leq \texttt{0xFFFFFFFF}, s_2, \perp).$$

Statement 3 does not effect the values of variables in the state, but rather the path condition. The symbolic path conditions are always mutually exclusive, and their disjunction is TRUE. We note that some of the symbolic paths will be *non-viable*, that is, their condition evaluates to FALSE. The remainder represent runs of the EDTS.

Symbolic paths fully capture the possible behavior of non-iterative programs. Handling loops (other than busy-waiting loops) requires the identification of loop invariants and is currently beyond the capability of MICROFORMAL. Also, handling indirect jumps is quite challending and cannot always be handled by MICROFORMAL.

4.1 Verification

We recall that systems P^1 and P^2 are $\mathcal{R}$-compatible if every execution in the same environment (matching entry observables) satisfying $\mathcal{R}$ will, under the same conditions, ends in the same exit door, with matching observables. For $j = 1, 2$, Let $\pi^j = (c^j, s^j, \delta^j)$ be a symbolic path in P^j, and s_0^j be the symbolic initial state of P^j. We require that

$$\mathcal{O}_{\texttt{entry}}(s_0^1) = \mathcal{O}_{\texttt{entry}}(s_0^2) \wedge \Theta^1(s_0^1) \wedge \Theta^2(s_0^2) \wedge c^1 \wedge c^2 \wedge \mathcal{R}(s_0^1, s_0^2) \longrightarrow \\ \delta^1 = \delta^2 \wedge \mathcal{O}_{\delta^1}(s^1) = \mathcal{O}_{\delta^1}(s^2) \qquad (1)$$

We recall that a computation is a path whose initial state satisfies Θ and that our equivalence definition refers to paths starting with matching entry observables. The path conditions, too, are conditions over the symbolic initial values. This formula is thus equivalent to requiring that any two computations π^1 and π^2 starting at matching initial states ($\mathcal{O}_{\texttt{entry}}(s_0^1) = \mathcal{O}_{\texttt{entry}}(s_0^2)$) reach the same door with matching observables, provided that the relevant restrictions $\mathcal{R}$ hold. We note that if $c^1 \wedge c^2$ is unsatifiable, then the formula is trivially true. However, since the path conditions are exhaustive, some instances of this formula are non-trivial.

Returning to our example, P^1 has two symbolic paths,

$$\pi_1^1 = (s_0^1.\texttt{reg2} + s_0^1.\texttt{reg3} > \texttt{0xFFFFFFFF},\ s_2^1,\ \texttt{address\_overflow})$$
$$\pi_2^1 = (s_0^1.\texttt{reg2} + s_0^1.\texttt{reg3} \leq \texttt{0xFFFFFFFF},\ s_9^1,\ \texttt{end})$$

where s_2^1 is s_0^1 with [TMP0= s_0^1.reg2 + s_0^1.reg3], and s_9^1 is the final state after executing instruction 8.

Similarly, the two symbolic paths of P^2 are

$$\pi_1^2 = (s_0^2.\texttt{reg2} + s_0^2.\texttt{reg3} + s_0^2.\texttt{reg6} > \texttt{0xFFFFFFFF},\ s_3^2,\ \texttt{address\_overflow})$$
$$\pi_2^2 = (s_0^2.\texttt{reg2} + s_0^2.\texttt{reg3} + s_0^2.\texttt{reg6} \leq \texttt{0xFFFFFFFF},\ s_{10}^2,\ \texttt{end})$$

where s_3^2 is s_0^2 with [TMP0= s_0^2.reg2+s_0^2.reg3+ s_0^2.reg6].

Applying (1) to π_2^1 and π_1^2 is easy:

$$\begin{array}{ll}
(a) & s_0^1.\texttt{reg1} = s_0^2.\texttt{reg1} \wedge \ldots \wedge s_0^1.\texttt{reg5} = s_0^2.\texttt{reg5} \wedge s_0^1.\texttt{memory} = s_0^2.\texttt{memory} \\
(b) & \wedge\ s_0^1.\texttt{pc} = 1 \wedge s_0^2.\texttt{pc} = 1 \\
(c) & \wedge\ (s_0^1.\texttt{reg2} + s_0^1.\texttt{reg3} < \texttt{0xFFFFFFFF}) \\
 & \qquad \wedge (s_0^2.\texttt{reg2} + s_0^2.\texttt{reg3} + s_0^2.\texttt{reg6} \geq \texttt{0xFFFFFFFF}) \\
(d) & \wedge\ s_0^2.\texttt{reg6} = 0 \longrightarrow \\
(e) & \qquad \texttt{end} = \texttt{address_overflow} \wedge (1 = s_0^2.\texttt{found_valid_address} \wedge \ldots)
\end{array}$$

where (a) is an instantiation of $\mathcal{O}_{\texttt{entry}}(s_0^1) = \mathcal{O}_{\texttt{entry}}(s_0^2)$, (b) of $\Theta^1(s_0^1) \wedge \Theta^2(s_0^2)$, (c) of $c^1 \wedge c^2$, (d) of $\mathcal{R}$, and (e) of $\delta^1 = \delta^2 \wedge \mathcal{O}_{\delta^1}(s^1) = \mathcal{O}_{\delta^1}(s^2)$. Note that without the restriction that $s_0^2.\texttt{reg6} = 0$, the antecedent would be satisfiable, generating a counterexample in which the two flows reach different exits. With the restriction the antecedent evaluates to FALSE, and the formula is valid.

4.2 Optimizations

Instead of verifying (1) for every pair of symbolic paths, we find it more more efficient to merge the symbolic paths of each system reaching the same exit before verification. The merged condition is simply a disjunction of all the conditions of symbolic paths reaching the door. The merged state is built by generating a case statement for each variable, with its value depending on the path condition. The verification condition has to be adapted for the merging of symbolic paths.

Symbolic pruning allows us to evaluate some of the conditional statements that appear in the target code according to the restrictions, and prunes branches of execution, thus reducing the number of non-viable paths generated, and the size of the verification conditions. In the microprograms we experimented with, as is often the case with incremental design, the new functionality is localized in some branches that are led to by tests explicitly referring to the restrictions. Indeed, empirical results establish that often symbolic pruning, combined with simple Boolean reductions, reduces prohibitively large VCs into manageable ones.

5 Results and Future Directions

The MICROFORMAL system has been under intensive research (in collaboration with academia) and development at Intel since the summer of 2003. We currently have a first

version providing core functionality. The system is fully automated and requires almost no manual intervention. As inputs it gets:

1. Files containing IRL modeling of microinstructions of current and next generation CPUs.
2. Source and target microprograms to be compared.
3. A set of restrictions under which to verify compatibility.

The output can be either *PASS*, or *FAIL*. In the latter case, a full counter-example – demonstrating the mismatch by concrete execution paths of the two microprograms being compared – is provided.

So far, MICROFORMAL has been used by microcode validators to formally verify backward compatibility for 80 microprograms. In these verification sessions, microcode of next-generation CPU is compared against that of current-generation CPU. The microprograms being verified, selected by microcode experts, are quite complex CISC flows that involve both memory interaction, and multiple sanity and permission checks that can result in exceptions. To date, MICROFORMAL has found three new unknown microcode bugs and redetected four known ones. The novel technology provided by MICROFORMAL has been recognized as one that can significantly improve the quality of microcode.

Some performance data we collected from our regression is presented in the table below.

Microprogram	Paths	Statements on longest path	Time (Seconds)
Program1	156	8497	32851
Program2	112	4323	1759
Program3	72	2346	10967
Program4	50	1988	924
Program5	39	1395	671

There are several interesting future directions that are currently being explored. The first natural extension is verifying compliance with hardware *assumptions*. Microcode and hardware interact through a complicated set of protocols. These protocols are currently expressed as a set of global microcode assumptions. A violation of these protocols by microcode could result in a multitude of undesired results - deadlocks, incorrect results, etc. Another kind of assumptions is local ones, or microcode assertions; some of the assumptions made by designers are formalized as inline assertions. These assertions are written inside the code, and are applicable only locally, at the point where they are written. Both kinds of assumptions (global and local) are validated using standard simulation, but their coverage is minimal. Most of these can be expressed as state predicates relating various environment variables. One way to formally verify assumptions is to use symbolic simulation for computing verification conditions, establishing that the required predicate holds at desired locations starting from an arbitrary initial state.

A more challenging direction is formal verification of new functionality (currently, this is validated only via standard simulation). New architectural functionality is coded both in microcode and in an architectural reference model. For formal verification of new functionality, the architects will be able to code the intended functionality of new

microprograms in IRL. MICROFORMAL will then compare the architectural specification against the actual microcode. This comparison should be able to detect discrepancies, providing complete data-path coverage and reducing the need to exercise the code in standard simulation.

Acknowledgements We wish to thank Ittai Anati for his help in clarifying some of the micro-architectural aspects that are relevant for formally modeling microcode, Amir Pnueli for ideas on modeling and verifying memory interaction, and Jim Grundy for his constructive comments on an earlier version of this paper.

References

[AHKV98] R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In *CONCUR'98*:163–178, 1998.

[BR02] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL'02*:1–3, 2002.

[CHRF00] D.W. Currie, A.J. Hu, S. Rajan, and M. Fujita. Automatic formal verification of DSP software. In *DAC'00*: 130–135, 2000.

[CKY03] E.M. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC'03*: 368–371, 2003.

[Cyr93] D. Cyrluk. Microprocessor Verification in PVS: A Methodology and Simple Example. Technical Report SRI-CSL-93-12, Menlo Park, CA, 1993.

[FH02] X. Feng and A.J. Hu. Automatic formal verification for scheduled VLIW code. In *ACM SIGPLAN Joint Conference: Languages, Compilers, and Tools for Embedded Systems, and Software and Compilers for Embedded Systems*, pages 85–92, 2002.

[HC98] S.Y. Huang and K.T. Cheng. *Formal Equivalence Checking and Design Debugging*. Kluwer, 1998.

[HJMS03] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *SPIN'03*:235–239, 2003.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, volume F-13 of *NATO ASI Series*, pages 477–498, 1985.

[HUK00] K. Hamaguchi, H. Urushihara, and T. Kashiwabara. Symbolic checking of signal-transition consistency for verifying high-level designs. In *FMCAD'00*:445–469, 2000.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[Nec00] G. Necula. Translation validation of an optimizing compiler. In *PLDI'00*: 83–94, 2000.

[PSS98] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS'98*:151–161, 1998.

[SH02] J. Sawada and W.A. Hunt. Verification of FM9801: An out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability. *J. on Formal Methods in System Design*, 20(2):187–222, 2002.

[SM96] M. Srivas and S. Miller. Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods. *J. on Formal Methods in System Design*, 8:153–188, 1996.

[Sta02] W. Stallings. *Computer Organization and Architecture, 6th ed.* Prentice Hall, 2002.

[ZPFG03] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. Voc: A translation validator for optimizing compilers. *J. of Universal Computer Science*, 9(2), 2003.