

## 6 Experimental Evaluation

In this section, we experimentally evaluate the theory of skipping refinement. Our goals are to evaluate the specification costs and benefits of using skipping refinement as a notion of correctness and to determine the impact that the use of skipping refinement has on state-of-the-art verification tools in terms of capacity and verification times.

We consider three case studies. The first is a hardware implementation of a JVM-inspired stack machine with an instruction buffer. The second is a memory controller with an optimization that eliminates redundant writes to memory. The third is a compiler transformation that vectorizes a list of scalar instructions.

For each case study, we compare two approaches for verifying correctness. The approaches require that we have both a specification system and an implementation system. The first approach is to prove input-output equivalence, *i.e.*, we show that if the specification and the implementation systems start in equivalent initial states and are given the same inputs, then if both systems terminate, the final states of the two systems are also equivalent. The second approach is to prove skipping refinement.

The first two case studies were developed using the BAT specification language [18], compiled to sequential AIGs, and then analyzed using model-checkers. We performed experiments with four model-checkers: TIP, IIMC, BLIMC, and SUPER\_PROVE [1] using a machine with an Intel Xeon X5677 with 16 cores running at 3.4GHz and 96GB main memory. We chose SUPER\_PROVE and IIMC as they are the top 2 model-checkers in the single safety property track [1]. TIP and BLIMC are chosen to cover temporal decomposition and bounded model-checking based tools. The timeout limit for model-checker runs is set to 900 seconds. While SUPER\_PROVE and IIMC are multi-threaded, BLIMC and TIP use only single core of the machine.

The last case study of compiler transformation involves systems whose state space is infinite. This allowed us to evaluate the use of skipping refinement for infinite state systems. Since model checkers cannot be used to verify such systems, we used the ACL2s interactive theorem prover [7].

The BAT files, corresponding AIGs, ACL2s models, and ACL2s proof scripts are publicly available [2].

Our results show that when correctness is based on input-output equivalence, model-checkers quickly start timing out as the complexity (size) of the systems increases. In contrast, when skipping refinement is used, much larger systems can be automatically verified. For the infinite state case study, interactive theorem proving was used and the issue there is really about the amount of human guidance needed for proofs. The skipping refinement proofs were much simpler.

### 6.1 JVM-inspired Stack Machine

In this case study we verify a stack machine inspired by the Java Virtual Machine (JVM). Java processors have been proposed as an alternative to just-in-time compilers to improve the performance of Java programs. Java processors

such as JME [10] fetch bytecodes from an instruction memory and store them in an instruction buffer. The bytecodes in the buffer are analyzed to perform instruction-level optimizations *e.g.*, instruction folding. In this case study, we verify BSTK, a simple hardware implementation of part of the JVM. BSTK is an incomplete and inaccurate model of JVM that only models an instruction memory, an instruction buffer and a stack. Only a small subset of JVM instructions are supported (*push*, *pop*, *top*, *nop*). However, even such a simple model is sufficient to exhibit the applicability of skipping simulation and the limitations of current hardware model-checking tools.

STK is the high-level specification with respect to which we verify the correctness of BSTK, the implementation. Both STK and BSTK are parameterized systems with parameters  $s$  (width of stack),  $k$  (depth of stack), and  $n$  (number of instructions in *imem*). Their behaviors are defined using abstract transition systems. The syntax and operational semantics are shown in Fig. 3. Adding element  $e$  to the beginning or end of list (or array)  $l$  is denoted by  $e::l$  and  $l::e$ , respectively. Each transition consists of a state:condition pair above a line, followed by the next state below the line. If a concrete state matches the state in a transition and satisfies all elements of the condition, then the state can transition to the state below the line. Every condition implicitly contains the condition  $pc < n$ .

The state of STK consists of an instruction memory, a program counter, and a stack. STK fetches an instruction from the instruction memory, executes it, increases the program counter and possibly modifies the stack, as outlined in Fig. 3. If none of the transition rules match (which happens when  $pc = n$ ), then STK stutters (this is not shown as a transition rule in Fig. 3).

The state of BSTK is similar to STK, except that it also includes an instruction buffer (with capacity  $k$ ). BSTK fetches an instruction from the instruction memory and as long as it is not *top* and the instruction buffer is not full, it queues it to the end of the instruction buffer and increments the program counter. If the fetched instruction is *top* the machine executes all buffered instructions in the order they were enqueued, thereby draining the instruction buffer and obtaining a new stack. Otherwise, if the instruction buffer is full, then the machine again executes all buffered instructions in the order they were enqueued, thereby draining the instruction buffer and obtaining a new stack, and it also updates the instruction buffer so that it only contains just the current fetched instruction. If none of the transition rules match, then BSTK drains the instruction buffer (if it is not empty) and updates the stack accordingly (this is not shown in Fig. 3). Note that both machines are deterministic.

Let  $\mathcal{M}_A = \langle S_A, \xrightarrow{A}, L_A \rangle$  and  $\mathcal{M}_C = \langle S_C, \xrightarrow{C}, L_C \rangle$  be the transition systems for STK and BSTK as specified in Fig. 3. With input-output equivalence as the correctness condition, we say that  $\mathcal{M}_C$  correctly implements  $\mathcal{M}_A$  under a refinement map  $r$  if for every  $\mathcal{M}_C$  state  $s_0$  that has an empty instruction buffer, if we run  $\mathcal{M}_C$  to completion (which occurs when  $pc = n$ ) to reach state  $s_C$ , then starting  $\mathcal{M}_A$  in state  $r(s_0)$  and running  $\mathcal{M}_A$  to completion yields a state  $w_a$

$stk := [v_1, \dots, v_s]$	(Stack)
$inst := \langle push\ v \rangle \mid \langle pop \rangle \mid \langle top \rangle \mid \langle nop \rangle$	(Instruction)
$imem := [inst_1, \dots, inst_n]$	(Program)
$pc := 0 \mid 1 \mid \dots \mid n$	(Program Counter)
$ibuf := [inst_1, \dots, inst_k]$	(Instruction Buffer)
$sstate := \langle imem, pc, stk \rangle$	(STK State)
$istate := \langle imem, pc, ibuf, stk \rangle$	(BSTK State)

<p>STK (<math>\xrightarrow{A}</math>) where <math>s = \text{capacity of } stk, t =  stk </math></p> $\frac{\langle imem, pc, stk \rangle : imem[pc] = \langle push\ v \rangle, t < s}{\langle imem, pc + 1, v :: stk \rangle}$ $\frac{\langle imem, pc, stk \rangle : imem[pc] = \langle push\ v \rangle, t = s}{\langle imem, pc + 1, stk \rangle}$ $\frac{\langle imem, pc, [] \rangle : imem[pc] = \langle pop \rangle}{\langle imem, pc + 1, [] \rangle}$ $\frac{\langle imem, pc, v :: stk \rangle : imem[pc] = \langle pop \rangle}{\langle imem, pc + 1, stk \rangle}$ $\frac{\langle imem, pc, stk \rangle : imem[pc] = \langle top \rangle}{\langle imem, pc + 1, stk \rangle}$ $\frac{\langle imem, pc, stk \rangle : imem[pc] = \langle nop \rangle}{\langle imem, pc + 1, stk \rangle}$
<p>BSTK (<math>\xrightarrow{C}</math>) where <math>k = \text{capacity of } ibuf, m =  ibuf </math></p> $\frac{\langle imem, pc, ibuf, stk \rangle : m < k, imem[pc] \neq top}{\langle imem, pc + 1, ibuf :: imem[pc], stk \rangle}$ $\frac{\langle imem, pc, ibuf, stk \rangle : imem[pc] = top, \langle ibuf, 0, stk \rangle \xrightarrow{A}^m \langle ibuf, m, stk' \rangle}{\langle imem, pc + 1, [], stk' \rangle}$ $\frac{\langle imem, pc, ibuf, stk \rangle : m = k, \langle ibuf, 0, stk \rangle \xrightarrow{A}^m \langle ibuf, m, stk' \rangle}{\langle imem, pc + 1, [imem[pc]], stk' \rangle}$

Fig. 3: Syntax and Semantics of Stack and Buffered Stack Machine

such that  $r(s_c) = w_a$ . The refinement map we use is defined as follows.

$$r(\langle imem, pc, ibuf, stk \rangle) = \langle imem, pc, stk \rangle$$

Before we describe skipping refinement, we first discuss why a simpler notion of refinement such as stuttering refinement will not suffice. If the BSTK takes a step which requires it to drain its instruction buffer, then the stack will be updated to reflect the execution of a potentially large number of instructions, something that is neither a stuttering step nor a single transition of the STK system. Therefore, it is not possible to prove that BSTK refines STK, using stuttering refinement and a refinement map that does not transform the *stack* (such as  $r$ , defined above).

We now formulate the correctness of BSTK based on skipping refinement. We show  $\mathcal{M}_C \lesssim_r \mathcal{M}_A$ , using Definition 5. First observe that the largest number of STK steps BSTK can make in one transition arises when the instruction buffer is full, in which case the BSTK machine executes all instructions in the instruction buffer. Thus condition WFSK2d in Definition 5, that requires us to reason about reachability, can easily be reduced to bounded reachability. Hence we set  $j = k + 1$ , where  $k$  is the capacity of the instruction buffer and condition WFSK2d is  $\langle \exists v : w \rightarrow^{<j} v : uBv \rangle$ . Next we define the refinement map, but first we note that we do not have to consider all syntactically well-formed STK states. We only have to consider states whose instruction buffer is consistent with the contents of the instruction memory, so-called *good* states based on commitment refinement maps [16]. One way of defining a good state is as follows: state  $s$  is good iff  $pc \geq |ibuf|$  and stepping BSTK from  $\langle imem, pc - |ibuf|, [], stk \rangle$  state for  $|ibuf|$  steps yields state  $s$ , where  $|ibuf|$  is number of instructions in state  $s$ 's instruction buffer. As part of the skipping refinement proof, we define the notion of a good state and show that good states are closed under the transition relation of BSTK.

Let  $S'_C$  is the set of good states. The refinement map  $R : S'_C \rightarrow S_A$  is defined as follows:  $R.s = \langle imem, pc - |ibuf|, stk \rangle$ . Given  $R$ , we define  $B = \{ \langle s, R.s \rangle \mid s \in S'_C \}$ .

Since STK and BSTK are deterministic machines and STK does not stutter, we only need to define one rank function:  $rank : S'_C \rightarrow \omega$  where  $rank(s) = k - |ibuf|$  (recall that  $k$  is capacity of *ibuf*). We can also simplify WFSK2 (Definition 5) as follows.

For all  $s, u \in S'_C$  such that  $s \xrightarrow{C} u$ :

$$R.s \xrightarrow{A}^{<k+1} R.u \vee [R.u = R.s \wedge rank(u) < rank(s)] \quad (2)$$

Notice that since BSTK is deterministic,  $u$  is a function of  $s$ , so we can remove  $u$  from the above formula, if we wish. Since  $k + 1$  is a constant, we can expand out  $\xrightarrow{A}^{<k+1}$  using only  $\xrightarrow{A}$  instead.

To evaluate the computational benefits of skipping refinement, we created a benchmark suite parameterized by the size of *imem*, *ibuf*, and *stk* (Table 1). We defined the STK and the BSTK systems and encoded correctness conditions

based on input-output equivalence and skipping refinement using the BAT specification language and tool. Using BAT, we compiled these problems to sequential AIGs and used hardware model-checkers to perform the verification.

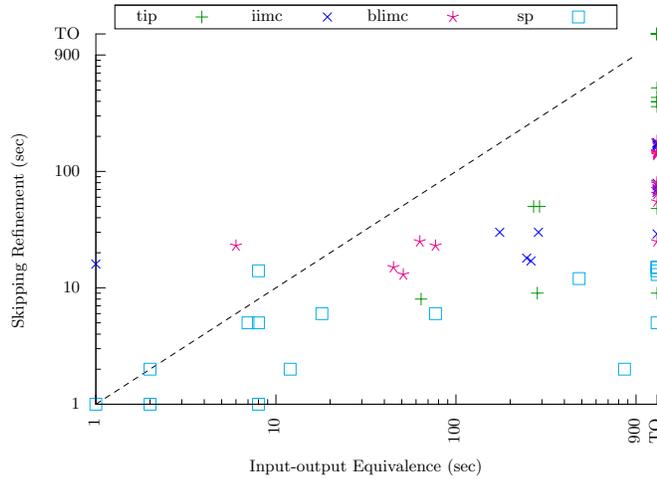


Fig. 4: Running time (log scale) of model-checkers for stack machine and memory controller

In Fig. 4, we plot the running times (in seconds) for four model-checkers: SUPER\_PROVE (SP), TIP, IIMC and BLIMC. The x-axis represents the running time of model-checkers using input-output equivalence and y-axis represents the running time using skipping refinement. A point with  $x = 900$  s indicates that the model-checker timed out for input-output equivalence while  $y = 900$  s indicates that the model-checker timed out for skipping refinement. Results show that model-checkers timeout for most of the configurations when using input-output equivalence while all model-checkers except TIP can solve all the configurations using skipping refinement. Furthermore, there is an improvement of several orders of magnitude in the running time when using skipping refinement. Notice that the number of latches for the skipping refinement tables (for this and all other case studies) is 0. This is because we just unrolled Formula 2 as described above. We believe that if we encoded the problem as a sequential AIG, then performance would tend to improve, based on our experience with these problems. Given that performance was already so much better than the performance on the input-output equivalence problems, we did not try to further simplify the skipping proof (but notice that we did use sequential AIGs for the input-output equivalence proofs because the combinatorial AIGs would have been too large).

stkdepth	stkword	ibuf	imem	inputs	latches(io)	gates(io)	gates(sks)
8	8	8	16	227	552	24460	120598
8	8	16	64	707	1597	51019	307185
8	8	32	128	1347	3040	114380	859186
8	8	32	256	2627	5602	134112	897243
8	8	32	512	5186	10724	173557	935298
16	16	16	32	836	1981	255683	863083
16	16	16	64	1412	3135	264472	990759
16	16	32	128	2564	5730	539129	2080698
16	16	32	256	4868	10340	574221	2118753
16	16	32	512	9476	19558	644385	2152063

Table 1: Configurations of BSTK and STK machines and size of AIGs for input-output equivalence(io) and skipping refinement(sks).

## 6.2 Memory Controller

Modern microprocessors operate at a higher clock frequency than their main memories. Thus it is essential for a memory controller, the interface between the CPU and main memory, to buffer requests and responses and synchronize communication between the CPU and memory. Moreover, current memory controllers implement optimizations to maximize available memory bandwidth utilization. In this case study, we model such a memory controller, OptMEMC. OptMEMC fetches a memory request from location  $pt$  in a queue of CPU requests,  $reqs$ . It enqueues the fetched request in the request buffer,  $rbuf$  and increments  $pt$  to point to the next CPU request in  $reqs$ . If the fetched request is a *read* or the request buffer is full (the capacity of  $rbuf$  is  $k$ , a fixed positive integer), then before enqueueing the request into  $rbuf$ , OptMEMC first analyzes the request buffer for consecutive write requests to the same address in the memory ( $mem$ ). If such a pair of writes exists in the buffer, it marks the older write requests in the request buffer as redundant. Then it executes all the requests in the request buffer except the marked (redundant) ones. Requests in the buffer are executed in the order they were enqueued.

To reason about the correctness of OptMEMC using refinement, we define an abstract, high-level specification system, MEMC, that describes the “legal” behavior of any memory controller implementation. It fetches memory requests from the CPU one at a time and immediately executes the *read* or *write* request. The syntax and the semantics of MEMC and OPTMEMC are given in Fig. 5, using the same conventions as described previously in stack machine section. We define parameterized systems using parameters that determine the size of  $req$ ,  $rbuf$  and  $mem$ , and encode the input-output equivalence between MEMC and OptMEMC using the BAT specification language. We then compile the descriptions to sequential AIGs and use model-checkers to verify correctness.

We now formulate the correctness of OptMEMC based on the notion of skipping refinement. Let  $\mathcal{M}_A = \langle S_A, \xrightarrow{A}, L_A \rangle$  and  $\mathcal{M}_C = \langle S_C, \xrightarrow{C}, L_C \rangle$  be transition systems for MEMC and OptMEMC respectively. Given a refinement map  $r : S_C \rightarrow S_A$ , we use Definition 5 to show that  $\mathcal{M}_C \lesssim_r \mathcal{M}_A$ . As was the case

$mem := [v_1, \dots, v_n]$	(Memory)
$req := \langle write\ addr\ v \rangle \mid \langle read\ addr \rangle$ $\quad \mid \langle refresh \rangle, addr < n$	(Request)
$reqs := [req_1, \dots, req_n]$	(Requests)
$rbuf := [req_1, \dots, req_k]$	(Request Buffer)
$sstate := \langle reqs, pt, mem \rangle$	(MEMC State)
$istate := \langle reqs, pt, rbuf, mem \rangle$	(OptMEMC State)

<p>MEMC (<math>\xrightarrow{A}</math>)</p> $\frac{\langle reqs, pt, mem \rangle, reqs[pt] = \langle write\ addr\ v \rangle}{\langle reqs, pt + 1, mem[addr] \leftarrow v \rangle}$ $\frac{\langle reqs, pt, mem \rangle, reqs[pt] = \langle read\ addr \rangle}{\langle reqs, pt + 1, mem \rangle}$ $\frac{\langle reqs, pt, mem \rangle, reqs[pt] = \langle refresh \rangle}{\langle reqs, pt + 1, mem \rangle}$
<p>OptMEMC (<math>\xrightarrow{C}</math>)</p> <p>Let <math> rbuf  = j</math></p> $\frac{\langle reqs, pt, rbuf, mem \rangle, \quad j < k, \quad req \neq top}{\langle reqs, pt, rbuf :: reqs[pt], mem \rangle}$ $\frac{\langle reqs, pt, rbuf, mem \rangle, \quad reqs[pt] = \langle read\ addr \rangle, \quad \langle rbuf, 0, mem \rangle \xrightarrow{A}^j \langle rbuf, j, mem' \rangle}{\langle reqs, pt, nil, mem' \rangle}$ $\frac{\langle reqs, pt, rbuf, mem \rangle, \quad j = k, \quad \langle rbuf, 0, mem \rangle \xrightarrow{A}^j \langle rbuf, k, mem' \rangle}{\langle reqs, pt, rbuf :: reqs[pt], mem' \rangle}$

Fig. 5: Syntax and Semantics of MEMC and OptMEMC

with the previous case study, OptMEMC and MEMC are deterministic machines and MEMC does not stutter. WFSK2 (Definition 5) can again be simplified to Formula 2. We encode Formula 2 for the systems in Table 2 using BAT and compile the correctness condition to an AIG.

The model checking running times are shown in Fig. 4. While model-checkers timeout for many configurations (points with  $x = 900s$  in Fig. 4) when using input-output equivalence, all except TIP solve all the configurations when using skipping refinement. Furthermore, the running times with skipping refinement show improvement of several orders of magnitude. Also, the performance of SUPER\_PROVE when using skipping refinement is much more robust with respect to the size of the system than when using input-output equivalence.

msize	mword	rbuf	reqs	input	latches(io)	gates(io)	gates(sks)
8	8	8	32	480	1082	26429	119075
8	8	8	64	896	1916	32817	117454
8	8	16	32	480	1187	48610	293098
8	8	16	64	896	2021	54998	302956
16	16	32	64	1664	4054	534746	2057352
16	16	32	128	3072	6872	556142	2097336
16	16	32	256	5888	12506	598914	2135409
16	16	32	512	11520	23772	684438	2173481

Table 2: Configurations of OPTMEMC and MEMC machines and size of AIGs for input-output equivalence(io) and skipping refinement(sks).

### 6.3 Superword Level Parallelism with SIMD instructions

An effective way to improve the performance of multimedia programs running on modern commodity architectures is to exploit Single-Instruction Multiple-Data (SIMD) instructions (*e.g.*, the SSE/AVX instructions in x86 microprocessors). Compilers analyze programs for *superword level parallelism* and when possible replace multiple scalar instructions with a compact SIMD instruction that concurrently operates on multiple data [12]. In this case study, we illustrate the applicability of skipping refinement to verify the correctness of such a compiler transformation using ACL2s, an interactive theorem prover. The source language consists of scalar instructions and the target language consists of both scalar and vector instructions. We model the transformation as an function that takes as input a program in the source language and outputs a program in the target language. Instead of proving that the compiler is correct, we prove the equivalence of the source program with the generated target program, *i.e.*, we use the translation validation approach to compiler correctness [4].

For presentation purposes, we make some simplifying assumptions: the state of the source and target programs (modeled as transition systems) is a three-tuple consisting of a sequence of instructions, a program counter and a store.

$$\begin{array}{l}
a = b + c \\
d = e + f
\end{array}
\quad \rightarrow \quad
\begin{array}{|c|} \hline a \\ \hline d \\ \hline \end{array}
=
\begin{array}{|c|} \hline b \\ \hline e \\ \hline \end{array}
+_{SIMD}
\begin{array}{|c|} \hline c \\ \hline f \\ \hline \end{array}$$

$$\begin{array}{l}
u = v \times w \\
x = y \times z
\end{array}
\quad \rightarrow \quad
\begin{array}{|c|} \hline u \\ \hline x \\ \hline \end{array}
=
\begin{array}{|c|} \hline v \\ \hline y \\ \hline \end{array}
\times_{SIMD}
\begin{array}{|c|} \hline w \\ \hline z \\ \hline \end{array}$$

Fig. 6: Superword Parallelism

We also assume that a SIMD instruction operates on *two* sets of data operands simultaneously and that the transformation identifies parallelism at the basic block level. Therefore, we do not model any control flow instruction. Note that we do *not* reorder instructions in the source program. Fig. 6 shows how two add and two multiply scalar instructions are transformed into corresponding SIMD instructions.

$loc := \{x, y, z, a, b, c, \dots\}$	(Variables)
$sop := add \mid sub \mid mul \mid and \mid or \mid nop$	(Scalar Ops)
$vop := vadd \mid vsub \mid vmul \mid vand \mid vor \mid vnop$	(Vector Ops)
$sinst := sop\langle z \ x \ y \rangle$	(Scalar Inst)
$vinst := vop\langle c \ a \ b \rangle\langle f \ d \ e \rangle$	(Vector Inst)
$sprg := [] \mid sinst :: sprg$	(Scalar Program)
$vprg := [] \mid (sinst \mid vinst) :: vprg$	(Vector Program)
$store := [] \mid \langle x, v_x \rangle :: store$	(Registers)

<div style="border-bottom: 1px solid black; padding-bottom: 10px;"> <p>Scalar Machine (<math>\xrightarrow{A}</math>)</p> <math display="block"> \frac{\langle sprg, pc, store \rangle, \{ \langle x, v_x \rangle, \langle y, v_y \rangle \} \subseteq store, \quad sprg[pc] = sop\langle z \ x \ y \rangle, \quad v_z = \llbracket (sop \ v_x \ v_y) \rrbracket}{\langle sprg, pc + 1, store _{z:=v_z} \rangle} </math> </div> <div style="padding-top: 10px;"> <p>Vector Machine (<math>\xrightarrow{C}</math>)</p> <math display="block"> \frac{\langle vprg, pc, store \rangle, \{ \langle x, v_x \rangle, \langle y, v_y \rangle \} \subseteq store, \quad sprg[pc] = sop\langle z \ x \ y \rangle, \quad v_z = \llbracket (sop \ v_x \ v_y) \rrbracket}{\langle vprg, pc + 1, store _{z:=v_z} \rangle} </math> <math display="block"> \frac{\langle vprg, pc, store \rangle, \quad vprg[pc] = vop\langle c \ a \ b \rangle\langle f \ d \ e \rangle, \quad \{ \langle a, v_a \rangle, \langle b, v_b \rangle, \langle d, v_d \rangle, \langle e, v_e \rangle \} \subseteq store, \quad \langle v_c, v_f \rangle = \llbracket (vop \ \langle v_a \ v_b \rangle \langle v_d \ v_e \rangle) \rrbracket}{\langle vprg, pc + 1, store _{c:=v_c, f:=v_f} \rangle} </math> </div>
---

Fig. 7: Syntax and Semantics of Scalar and Vector Program

The syntax and operational semantics of the scalar and vector machines are given in Fig. 7, using the same conventions as described previously in the stack

machine section. We denote that  $x, \dots, y$  are variables with values  $v_x, \dots, v_y$  in *store* by  $\{\langle x, v_x \rangle, \dots, \langle y, v_y \rangle\} \subseteq \text{store}$ .  $\llbracket (sop\ v_x\ v_y) \rrbracket$  denotes the result of the scalar operation *sop* and  $\llbracket (vop\ \langle v_a\ v_b \rangle \langle v_d\ v_e \rangle) \rrbracket$  denotes the result of the vector operation *vop*. Finally, we use  $\text{store}|_{x:=v_x, \dots, y:=v_y}$  to denote that variables  $x, \dots, y$  are updated (or added) to *store* with values  $v_x, \dots, v_y$ .

As was the case with the stack machine in subsection 6.1, it can be shown that the notion of stuttering simulation is too rigid to relate the scalar and the vector machines on scalar and vector programs produced by the compiler, no matter what refinement map we use. To see this, note that the vector machine might run exactly twice as fast as the scalar machine and during each step the scalar machine might be modifying the memory. Since both the machines do not stutter, in order to use stuttering refinement, the length of the vector machine run has to be equal to the run of the scalar machine.

Let  $\mathcal{M}_A$  and  $\mathcal{M}_C$  be transition systems of the scalar and vector machines, respectively. The vector machine is correct iff  $\mathcal{M}_C$  refines  $\mathcal{M}_A$ . We show  $\mathcal{M}_C \lesssim_r \mathcal{M}_A$ , using Definition 5. Determining  $j$ , an upper-bound on skipping that reduces condition WFSK2d in Definition 5 to bounded reachability is simple because the vector machine can perform at most 2 steps of the scalar machine at once; therefore  $j = 3$  suffices.

We next define the refinement map. Let *sprg* be the source program and *vprg* be the compiled vector program. We use the compiler to generate a lookup table *pcT* that maps values of the vector machine's program counter to the corresponding values of the scalar machine's program counter. The refinement map  $R : S_C \rightarrow S_A$  is defined as follows.

$$R(\langle vsprg, pc, store \rangle) = \langle sprg, pcT(pc), store \rangle$$

Note that *pcT*(*pc*) can also be determined using a history variable and this is preferable from a verification efficiency perspective. Given  $R$ , we define  $B = \{ \langle s, R.s \rangle \mid s \in S_C \}$ .

Since the machines do not stutter, WFSK2 (Definition 5) can be simplified as follows. For all  $s, u \in S_C$  such that  $s \xrightarrow{C} u$ :

$$R.s \xrightarrow{A}^{<3} R.u \tag{3}$$

Since the vector machine is deterministic,  $u$  is a function of  $s$ , so we can remove  $u$  from the above formula, if we wish. Also, we can expand out  $\xrightarrow{A}^{<3}$  to obtain a formula using only  $\xrightarrow{A}$  instead.

For this case study we used deductive verification methodology to prove correctness. The scalar and vector machines are defined using the data-definition framework in ACL2s [7,6]. We formalized the operational semantics of the scalar and vector machines using standard methods. The sizes of the program and store are unbounded and thus the state space of the machines is infinite. Once the definitions were in place, proving skipping refinement with ACL2s was straightforward. Proving input-output equivalence would have required theorem proving expertise and insight to come up with the right invariants, something we avoided. The proof scripts are publicly available [2].