Model Checking Reconfigurable Processor Configurations for Safety Properties*

John Cochran, Deepak Kapur, and Darko Stefanović

University of New Mexico, Albuquerque NM 87131, USA,
cochran@cs.unm.edu,
WWW home page: http://www.cs.unm.edu/~slugboy/index.html

Abstract. Reconfigurable processors pose unique problems for program safety because of their use of computational approaches that are difficult to integrate into traditional program analyses. The combination of proof-carrying code for verification of standard processor machine code and model-checking for array configurations is explored. This combination extends proof-carrying code to provide a context for model checking, but uses standard model checking technology. This approach is shown to be useful in verifying safety properties including the synchronization of memory access by the reconfigurable array and memory access bounds checking.

1 Introduction

Reconfigurable computing is a rapidly evolving technology that has great potential for improved processing efficiency for a variety of important computational tasks. This improvement, however, comes at the expense of increased risk of problems from faulty or malicious programming. Although methods of mitigating this risk have not been explored deeply in the literature, this will be necessary for the success of the technology.

We are exploring model checking combined with proof-carrying code as a potential method of ensuring safety of reconfigurable processor programs. This exploration will be in the context of the Garp reconfigurable processor. We show that significant safety properties can be *efficiently* and *automatically* verified by model checking. However, we also report that model-checking some other properties of interest is not efficient, and we consider possible directions for addressing this shortcoming.

2 Safety for Reconfigurable Computing

There are several reasons that cause reconfigurable processors to pose an increased safety risk: reconfigurable processors entail greater consequences of unsafe operation because reconfigurable fabrics can be destroyed by some improper uses; reconfigurable processors have more complex functionality; the large variability of computations and fine-grained concurrency make programming reconfigurable fabrics more difficult; reconfigurable configurations are also less accessible and less modifiable; and finally, the

* Partially supported by the NSF Grants nos. CCR-9996150 and ITR-CCR-0113611.

computation on a reconfigurable fabric cannot in general be augmented by instrumenting code because of the lack of space and tight timing issues.

Safety properties of programs are assertions that some bad event never happens during a computation. In contrast, correctness or liveness properties state that a computation produces correct results or that some event eventually happens during a computation. What the bad events are will be left to a safety policy which will depend on the context of program usage. Safety properties are generally easier to verify than correctness or liveness properties, which makes them a better candidate for formal methods.

Safety properties for reconfigurable processors can be decomposed into properties of the standard part of the processor and properties of the reconfigurable array. This decomposition assumes that the two sections of the processor are relatively independent. This is true of the Garp and similar processors but not true of reconfigurable processors that implement specialized machine instructions in the reconfigurable array.

Each of these types of safety property is further subdivided into generic properties that must be true for all instances (programs and array configurations) and specific properties that must be true of individual instances and that are derived from a safety policy. Generic safety properties include such properties as not running code outside of the program code segment and only calling functions at code locations where functions reside (for the standard part of the processor), and avoiding memory bus conflicts and undefined configurations (for the reconfigurable array).

Unfortunately, there is no obvious method of verifying both the standard and reconfigurable parts of the program in a single framework. Proof-carrying code works well for sequential instructions that can be symbolically evaluated to produce a verification condition, but does not address the concurrent, instructionless computation performed by the reconfigurable array. On the other hand, model checking works well for finite state, concurrent systems such as the reconfigurable array, but has problems dealing with infinite state systems such as standard program code.

The novel approach explored here uses model checking to verify safety properties of the reconfigurable array, and proof-carrying code to verify safety properties of the standard machine code. In addition, proof-carrying code provides a context for model checking the reconfigurable array by providing preconditions, postconditions, a memory partition to prevent memory conflicts between the standard processor executions and array executions while the array is running, and local safety properties. This is achieved by extended Proof-carrying code's context mechanism for function calls to reconfigurable array executions.

The context is needed for model checking the array configurations because simply checking the configuration for safety properties without any external context would restrict the class of such properties greatly. For example, any property that relies on the correct initialization of array registers cannot be verified unless there is some external assurance that the registers are in fact correctly initialized.

3 Proof-Carrying Code

Proof-carrying code (PCC) [NL97,Nec97,Nec98] is a method of ensuring the safety of untrusted machine code. It relies on the code producer to produce a verification of safety and transmit evidence of the verification to the code consumer.

3.1 Overview

The first component of PCC is a *safety policy*, which is specified by the developer of the system and which can vary widely over a large space of possible policies. There is no notion of a universally acceptable safety policy for all systems in PCC, although there is a common infrastructure that is part of the final safety policy specification. Each system must define a safety policy (or a set of such policies) using the common infrastructure and advertise it to developers of programs for the system.

The program developer must provide a *proof of compliance* with the advertised safety policy. This proof, the second component of PCC, must be in a specific format to allow automated checking. This also means that the proof must refer directly to the machine code and not an abstracted version of the program. How the proof is produced does not matter for the PCC system.

The third component of PCC is a *proof checker* for the client. When the client receives code from an untrusted source it examines the code along with the attached proof to make sure that the code obeys the safety policy. This checking must be automated, fast, and highly reliable. It must be the case that if the proof checks then it is safe to run the code, according to the advertised policy. Thus it does not matter if the code or the proof is tampered with or garbled. If the proof still checks, the code will be safe to run even if it does not necessarily perform its expected function.

The main advantage of PCC is that the hard work—compiling the code and producing the proof—is in the hands of the code producer, while only the easy work, checking the proof and running the program, is required of the code consumer. This asymmetry allows the hard work to be done once and used many times. This is opposed to analyzing the code every time a new client uses it. This alternative is not only wasteful of processor time for the client, but much harder than having the producer give a proof, as the client has less information about the code than the producer—no source code, no formal or informal specifications, etc.

3.2 Extensions

We have extended the PCC system for a reconfigurable array. The semantics of safe program execution and the symbolic evaluator that produces the safety predicate have been extended to treat array executions as a type of function call. The extension includes new instructions for accessing array registers, loading configurations, and starting array execution. The semantics and symbolic execution of old instructions are updated to take into account the reconfigurable array execution state. These extensions are necessary to provide the correct semantics of instructions reading array registers. If these instructions are called during array execution, the value read is indeterminate, otherwise it is available to the symbolic evaluator. The extensions are documented in [Coc02].

A big difference between regular function calls and reconfigurable array executions is that the computation calling a function waits for it to return while the computation initiating an array execution may continue a concurrent computation. We model this in the safety semantics by the use of two variants of Hilbert's ε-calculus [Lei69].

One version models values in the reconfigurable array that are inaccessible by the standard part of the program with the ε constructor. This constructor follows the rules:

$$\exists v.Fv \supset F(\varepsilon_\alpha v.Fv) \tag{1}$$

$$\forall v(Fv \equiv Gv) \supset \varepsilon_\alpha v.Fv = \varepsilon_\alpha v.Gv \tag{2}$$

This states that for any locations α, if it can contain a value for which F holds, then F holds of the value $\varepsilon_\alpha v.Fv$, and any such constructed value is unique. ε-terms can occur in the symbolic evaluation of the program, but not the final property to be proved.

The other version models values computed by the reconfigurable array that are non-deterministic to the standard part of the processor. This version is called η-calculus[1], and follows the rule:

$$\exists v.Fv \supset F(\eta_\alpha v.Fv) \tag{3}$$

This states that for any locations α, if it can contain a value for which F holds, then F holds of the value $\eta_\alpha v.Fv$. There is no guarantee of uniqueness for η-terms. η-terms only occur in the semantics and are used to prove soundness.

In addition to the property to prove for the standard section of the program, the extension provides preconditions and postconditions for array execution, partitions the memory between the standard processor and reconfigurable array, and provides a context for the safety policy that allows a concrete realization of safety properties. The preconditions state what values can be expected to be placed into the reconfigurable array registers at the start of array execution, as well as the number of cycles the array will run. The postconditions state what values can be expected to be left in the reconfigurable array registers following array execution. The memory partition simply states which addresses the reconfigurable array can access without interference from the rest of the processor. The safety properties to be checked are evaluations of the safety policy in the state reached by the symbolic evaluator at the time of array execution.

4 Model Checking

Model checking can be easily summarized as a simple problem dealing with machines and temporal propositions [CGP99]. Assume that there is some finite state machine $M = \{S, S', T, L\}$, where S is the set of states, $S' \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is a total transition relation, and $L : S \to 2^A$ is an injective function from states to sets of state variables A labelling those variables that are true in the state. The model-checking problem is then to determine, for any temporal formula f, those states $S_t \subseteq S$ for which f is true.

Model checking is used to verify safety properties of the reconfigurable array. Model checking is a formal verification method that automatically examines finite models of concurrent systems to determine the properties of the models. These verification properties can be expressed in any one of a variety of logics but propositional temporal logics give a good balance between expressibility and efficiency of checking. In this work we use RTCTL [Cam96], a branching-time propositional temporal logic [Pnu77]. RTCTL uses bounded temporal quantifiers to implement bounded model checking.

RTCTL formulas express ways in which propositions can be true over a branching model of time. For example,

$$\mathbf{ABG}\ 0..1023(x < y + 2048) \tag{4}$$

[1] η-calculus is called κ-calculus in [Coc02], but was later found to be equivalent to η-calculus as discussed in [Lei69].

states that on all paths from the current time, at all times from now until 1023 time steps later, x is less than y plus 2048, and

$$\mathbf{AF}(x = 0) \tag{5}$$

states that on all paths, at some time from now on, x is equal to 0. The first of these examples is a safety property over a bounded time period, while the second is a liveness property over all time.

Model checking does not have to be extended for our purposes. Off-the-shelf model checkers provide rich enough description languages to describe the system to be modeled, and specification languages to specify the properties to be checked. In particular, we have used the NuSMV system [CR98] for the verification of array configurations. NuSMV uses the language SMV to describe models. This language is similar to hardware description languages or process algebras. As specification language, NuSMV supports RTCTL. For RTCTL model checking, NuSMV uses ordered binary decision diagrams (OBDDs) as a symbolic representation of the model and specification [Bry92]. This use of symbolic representations greatly improves the efficiency of model-checking for array configurations.

4.1 Building Models for Configurations

Before model checking a configuration, it is necessary to build a model for it. The usual practice in model checking systems is to build a model in a description language before implementing the system. In our case, the code consumer does not have any information about configurations except their bit level encoding and the machine code where they occur. Thus we build the model by translating the bit level encoding to the SMV language. This uses only the bit level encoding; any information from the surrounding machine code that is needed to verify safety must be used in the safety properties for the configuration. Details can be found in [Coc02].

5 The Garp Processor

The reconfigurable processor which we use as our example is the Garp processor designed at Berkeley [Hau00,HW97,Hau97]. This processor has not been physically implemented but it has been thoroughly specified and documented, which is critically important for proving safety properties. The documentation and the ability of the reconfigurable array to independently access memory were the deciding factors when we chose Garp. The basic feature of the Garp design that will be considered is the reconfigurable logic array.

The Garp logic array is an array of blocks arranged into 24 columns. The leftmost block in each row is a control block and all the others are logic blocks. There is an unspecified number of rows in the array (but there cannot be more than 32 because of the constraints of the configuration file format).

Each logic block has two 2-bit clocked registers. There are four 2-bit inputs to the logic blocks that can come from any wire pair accessible to the block, from latched registers, or from the binary constants 00 or 10. The outputs can be the registers, the block function output, or one of the block inputs.

The **table mode** takes the four 2-bit main inputs and outputs two bits depending on a lookup table. The **split table mode** is similar but has two separate tables for the first and second bits.

The **select mode** implements a multiplexer which uses one input as control and selects between the other inputs. The inputs can be shifted to the left or inverted. The **partial select mode** is identical to select mode with the exception of which inputs are available.

The **carry chain mode** and the **triple add mode** are more complicated modes that can be used to implement functions requiring fast carry chains or arithmetic on three inputs.

Other functions of the logic array are initiated by the control blocks. Of primary interest, memory loads and stores can be initiated by the control blocks. This is independent of the standard processor, but accesses go through the same memory hierarchy as for the standard processor. There is only one memory address bus, so only one access can be initiated at a time. A control block can load or store the registers in its row over one of four memory data busses. More detailed information on the array features can be obtained from the Garp Architecture Manual [Hau97].

6 Safety Problems with Garp

A safety problem with the Garp reconfigurable array is that only one memory access may be initiated per cycle although more than one control block could be triggered to access memory depending on the state of the array. Thus it is possible for a perfectly legal configuration to attempt to perform an illegal action. A similar problem is that two memory accesses initiated at different times could illegally schedule the transfer of data for the same cycle. These actions could lead to unexpected behavior because there is no specification of what should happen in these circumstances. Because they cannot be checked syntactically, these control problems must be checked in some other manner. These problems are addressed by generic safety properties as discussed in Section 7.

Accessing and writing of data at addresses forbidden by a safety policy is another safety problem. Such accesses could lead to the unwanted access to private information, buffer overruns [CWP+00], stack smashing [One96], and similar problems. These can occur in perfectly legal configurations and programs but are very troublesome. Common solutions to these problems in standard processors, boundary checks by the language, programmer, or analysis tool, do not work for the reconfigurable array. There is no suitable method to make the hardware handle these problems either. A method of preventing them in untrusted code is critical to ensure safety, therefore our work focuses on these problems. These problems are addressed by specific safety properties as discussed in Section 7.

Another class of safety problems is not addressed here. These include illegal block configurations in the encoding, multiply driven wires, and clock skew issues. Fortunately this type of safety problem has been investigated by Hauser and can be detected by hardware validation of configurations before they are loaded [Hau00]. Hauser also mentions dynamically checking for illegal actions by legal configurations in the hardware, but here we show that it is possible to check for these problems statically.

7 Properties to be Model Checked

Once a configuration is translated into SMV it can be checked for many properties. These properties fall into three classes:

- Generic memory control properties.
- Context dependent memory access properties.
- Context dependent postcondition properties.

Each of these properties can rely on the preconditions for the configuration, so the specifications are implications with the preconditions as an antecedent and the safety specifications as the consequent. Typically, the preconditions reflect array register and counter initialization, while the postcondition reflects assumptions that can be made about values in the array registers following execution.

The generic properties which will be checked for *all* configurations include:

- At most one memory access is initialized per clock cycle.
- At most one memory access is scheduled to use the bus for each cycle.
- There is a memory item ready to read when a row reads one.
- There is a row initiating a memory write when a row transfers to memory.

These properties all deal with synchronizing memory accesses so that they are defined by the semantics of Garp's reconfigurable array. If any of these properties is false, then there can be undefined behavior from the array. From a list of which control blocks can initiate memory accesses, it is possible to deduce these specifications.

The context dependent safety properties for a *particular* configuration include:

- All memory accesses respect the memory partition.
- All memory accesses respect the memory access safety properties.
- The postcondition is true after array execution.

These properties all rely on information from the safety policy and the symbolic evaluation of the program at the point where the array configuration is executed. Because the array uses standard binary representations for addresses, the configuration cannot be too devious in its memory accesses. The preconditions and postconditions must be in a standard representation because they are used in the verification of the standard code as well as the array. This allows a coherent translation of the properties expressed with standard program types such as integers and characters as primitives, and the model checking version which is expressed with bits as primitives.

Either of these types of properties can also use information about the initial count for the array execution. This count gives the number of cycles that the array executes if it does not halt itself first. The count is used in the bounds for temporal quantifiers such as $0..5$ in Equation 4.

8 Performance of NuSMV on Translated Input Files

The performance of the model checker, NuSMV 2.0 running on an AMD Athlon at 1900 MHz with 1024 M of memory under Debian Gnu/Linux 2.2, is the main factor to be explored for performance. Four example configurations were checked for six properties, the first of which is a generic property and the rest of which are specific properties:

– Memory control safety (MC).
– Memory alignment (MA).
– Lower bounds for memory reads (LBR).
– Upper bounds for memory reads (UBR).
– Lower bounds for memory writes (LBW).
– Upper bounds for memory writes (UBW).

The four configurations include three that perform the same function, but have different control paths and preconditions. The application is to read 200 word-sized pixels from an array, lighten each color component, and write the results back to the array. This application was chosen to reflect a common but problematic use of the reconfigurable array, reading a writing memory with little interaction with the standard processor.

The first configuration (IM1) has the precondition that the register in the fifth row of the reconfigurable array is loaded with a value equal to the value loaded in the first row of the reconfigurable array minus 14. This is because the first row reads the pixel array, the fifth row writes the result back, each of them is incremented by 2 on each cycle, and it takes seven cycles for the computation. The second configuration (IM2) has the precondition that the values loaded into the first and fifth row registers are equal. This is because the fifth row does not start incrementing until it is signaled on cycle 7 by the control path. The third configuration (IM3) does not have any corresponding precondition because it passes address from the first row to the fifth alongside the computation so that the value in the fifth row is always correct.

The fourth configuration (HASH) is a hash table. It is included to check the effect of *computed addresses* on the model checking. It simply reads values from an array, computes a 10-bit offset by repeated shifts and exclusive ors, and writes the value to an address plus the offset.

Each of the the configurations has preconditions to ensure that the control path is correctly initialized, and that the access locations fit into memory without any wraparound. The correct initialization preconditions are simply assertions that certain registers have certain values, while the memory fit preconditions are disjunctions of assertions that certain register locations have the value 0. All of the configurations have the trivial postcondition *true* because they do not leave any values in the array registers for later use. They only affect memory.

The results are presented in Table 1 for unbounded properties, and Table 2 for bounded properties. The memory write properties for IM1, IM3, and HASH do not appear in Table 2 because they did not finish model checking in under four days. Further information is available in [**?**].

The reasons for the poor performance on the write boundary properties (LBW, UBW) are varied. For IM1, the precondition requires a very long bit level specification that makes even trivial specifications practically uncheckable. The other properties did not rely on this precondition, so they could be checked without it. IM3 and HASH both have a write address that is dependent on several rows of registers and cycles. This seems to be more than the model checker can handle efficiently. Even for IM2 it took a significantly longer time to check the write boundary properties as they depend on more of the array than the rest of the properties.

IM3 and HASH took longer to check on the other properties because they are about two and three times as large as the other configurations. Thus the longer times are not unexpected.

This investigation of performance is far from complete, but it shows that common safety properties are efficiently checkable for certain types of configurations. Unfortunately, there is not a wider base of applications that could be used to produce realistic data, since Garp is not implemented.

	Configuration			
Property	IM1	IM2	IM3	HASH
MC	6.710	6.140	6.810	17.400
MA	6.740	13.690	14.140	33.670

Table 1. Results of Model Checking Example Configurations for Unbounded Properties. Model checking times are given in seconds.

	Time Step Bound						
Property	400	1000	1024	4000	4096	16000	16384
IM1 LBR	15.23	16.84	16.92	24.10	24.42	54.00	54.76
IM1 UBR	16.97	20.66	20.26	38.78	36.95	106.72	98.70
IM2 LBR	15.13	16.55	16.65	23.20	23.12	49.38	50.04
IM2 UBR	16.59	19.98	19.99	36.57	36.57	102.92	94.32
IM2 LBW	64.00	76.11	70.52	141.71	88.37	161.54	414.43
IM2 UBW	69.64	87.61	76.30	109.13	197.60	240.14	239.10
IM3 LBR	16.12	18.63	18.91	30.70	31.22	80.50	81.77
IM3 UBR	18.00	23.73	23.22	48.70	47.40	147.10	142.53
HASH LBR	27.27	29.30	29.20	38.84	39.29	78.56	79.93
HASH UBR	28.42	32.99	32.65	58.93	57.52	168.92	169.41

Table 2. Results of Model Checking Example Configurations for Bounded Properties. Model checking times are given in seconds.

9 Conclusions and Further Work

The main result of this work is that model checking reconfigurable processor configurations is a viable verification method for important safety properties. Although some properties have been found to be too complex for efficient checking, there may be methods to mitigate this in many cases. As the examples show, equivalent computations with different control strategies can have very different behavior when model checked. This could be taken into account in a compiler designed to produce efficiently checkable configurations.

Unfortunately, some properties seem to be uncheckable for practical purposes. There is no way to implement the hash table application without computing the write address, but even trivial computed address such as in IM3 do not check in a reasonable time.

Integrating model checking into a synoptic system of program verification in order to solve these problems is being explored. In particular, replacing proof-carrying code based on first-order logic with proof-carrying code based on temporal logic [BL02] is expected to provide a greater range of safety properties that can be checked. This may allow model checking of easy-to-check properties, which are then integrated into the proof of safety attached to the program as a whole.

In the hash table example, the model could be decomposed into small pieces involving only a single cycle. The pieces could then be model checked efficiently and the checked properties combined to yield a proof of safety. This would be possible because the proof-carrying code infrastructure would be able to check the correctness of temporal logic proofs.

References

[BL02] A. Bernard and P. Lee. Temporal logic for proof-carrying code. Technical Report CMU-CS-02-130, School of Computer Science, Carniege Mellon University, Pittsburgh, PA, 2002.

[Bry92] R.E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[Cam96] S.V. Campos. *A Quantitative Approach to the Formal Verification of Real-Time Systems*. PhD thesis, Carnegie Mellon University, 1996.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[Coc02] J. Cochran. Towards provably safe reconfigurable processor code: A model checking and proof-carrying code approach. Master's thesis, University of New Mexico, 2002.

[CR98] A. Cimatti and M. Roveri. *NuSMV 1.1 User Manual*. ITC-IRST and CMU, 1998.

[CWP+00] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 119–129, January 2000.

[Hau97] J. Hauser. *The Garp Architecture*. University of California at Berkeley, Department of Electrical Engineering and Computer Science, Computer Science Division, Oct 1997.

[Hau00] J.R. Hauser. *Augmenting a Microprocessor with Reconfigurable Hardware*. PhD thesis, University of California, Berkeley, 2000.

[HW97] J.R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[Lei69] A.C. Leisenring. *Mathematical Logic and Hilbert's ε-symbol*. MacDonald and Co., London, 1969.

[Nec97] G.C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[Nec98] G.C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.

[NL97] G.C. Necula and P. Lee. Safe untrusted agents using proof-carrying code. In *Mobile Agents and Security*, pages 61–89, 1997.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundation of Computer Science*. IEEE Computer Society Press, 1977.