Correct-by-Construction Design of Custom Accelerator Microarchitectures

Jin Yang, Zhenkun Yang, Jeremy Casas, and Sandip Ray, *Senior Member*, IEEE

***Abstract*—Modern application-specific System-on-Chip designs include a variety of accelerator blocks that customize microcontrollers with domain-specific instruction sets and optimized microarchitectures. Unfortunately, accelerator implementations can be highly error-prone, undermining the reliability and security of the entire system. In spite of recent successes in formal methods, full verification of a complex accelerator microarchitecture is still beyond the scope of state-of-the-art formal technologies. In this paper, we address this problem through a novel methodology for incremental verification that can be tightly integrated with the design process. Our approach depends on a new foundation for microarchitecture correctness that enables viewing microarchitecture features as program transformations in a compiler design. The foundations enable designing microarchitecture features as incremental, semantics-preserving optimizations. We show how to use the foundations to develop correct-by-construction implementations of various advanced features of modern microprocessors. We demonstrate the viability of the foundations in designing correct-by-construction methodology for a superscalar microarchitectural implementation of the Versatile Tensor Accelerator.**

***Index Terms*—Correct-by-construction design, transformation-based design formal verification and composition, microarchitecture design.**

I. Introduction

Modern application-specific System-on-Chip (SoC) designs include a variety of accelerator blocks for domain-specific applications. Such accelerators often use custom microcontrollers with their own instruction sets and highly optimized microarchitecture implementation. For instance, it is common for an SoC targeted for medical imaging to have its signal processing accelerator implemented as a microcontroller with custom vector processing instructions with a microarchitecture optimized for targeted real-time requirements under aggressive power constraints. Furthermore, implementations of microarchitecture features are distributed across hardware and software (microcode) components. Obviously, bugs in these microarchitectures can undermine the entire system functionality, possibly requiring complex in-field patching, or product recalls.

One approach to ensure correctness of a microarchitecture is formal verification, which entails the use of mathematical logic to prove that a computing system meets its specification. In case of an accelerator, the specification is defined by the Instruction Set Architecture (ISA) that represents the programmer's view of the accelerator [1]. Microarchitecture verification is one of the most widely studied areas of formal verification, with research spanning over two decades [2], [3], [4], [5], [6]. However, in spite of extensive research, its adoption in industrial design flows has been limited. A key challenge is the high cost of formal verification for a practical microarchitecture design. Most modern microarchitectures include features such as pipelining, out-of-order, speculative, and superscalar execution, memory hierarchy, a variety of exceptions and interrupts, etc. While formal verification of microarchitectures with such features have been accomplished [4], [7], they have depended on significant human interaction and often taken multiple man-years to achieve. Given the aggressive time-to-market requirements of today's accelerator designs, the cost of adopting such a verification methodology is prohibitive.

One approach to address the high cost of verification is to enable integration of verification as part of the design process. In particular, design of an accelerator microarchitecture includes an iterative process of *feature addition phase*, where new features are incrementally added to improve performance and energy efficiency (see Section III). We refer to a verification methodology as *incremental* if each feature can be verified individually and the results composed to derive an overall proof of correctness. An incremental verification methodology ensures that each added feature preserves the ISA functionality of the accelerator, and consequently produces a correct-by-construction methodology for microarchitecture design. Furthermore, an incremental methodology can significantly ameliorate verification complexity reducing the abstraction gap between successive feature additions. Unfortunately, existing verification methodologies are not directly amenable to incremental verification. Indeed, the different verification methodologies and even notions of correspondence employed for various microarchitectural models have become increasingly subtle, complex, and even controversial [1], [5], precluding their use as foundations for an incremental verification methodology.

In this paper, we develop a formal foundation for microarchitecture analysis that enables incremental analysis of

Manuscript received 5 January 2023; revised 9 October 2023; accepted 16 October 2023. Date of publication 8 November 2023; date of current version 22 December 2023. This work was supported by a research gift from Intel Corporation at University of Florida. Recommended for acceptance by J. Abella. *(Corresponding author: Jin Yang.)*

Jin Yang, Zhenkun Yang, and Jeremy Casas are with the Strategic CAD and Heterogenous Platform Lab, Intel Labs, Hillsboro, OR 97124 USA (e-mail: jin.yang@intel.com; zhenkun.yang@intel.com; jeremy.casas@intel.com).

Sandip Ray is with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: sandip@ece.ufl.edu).

Digital Object Identifier 10.1109/TC.2023.3329243

different microarchitecture features. The foundations have been mechanized in Dafny [8], and the properties of the formalization discussed in the paper have been mechanically checked. We demonstrate the approach in the correct-by-construction design of a superscalar microarchitecture implementation of the Versatile Tensor Architecture (VTA).

Central to our approach is the idea to view each microarchitecture component as a *program transformer*, *i.e.*, a mapping from a stream of instructions to another "equivalent" stream of instructions. This view makes explicit the role of each microarchitecture feature in executing program instructions. We show how the formalism facilitates intuitive, incremental analysis of different microarchitecture features. Finally, a key benefit of our approach is the ability to formalize the verification of microprocessors as an extension of compiler verification, which permits re-purposing of insights and tools from compiler analysis and enabling seamless software and hardware co-design.

VTA, the accelerator microarchitecture used in this paper as a demonstration vehicle, is a deep learning accelerator for fast and efficient dense linear algebra. The microarchitecture implementation includes features like multiple issue queues, variable execution pipelines for *general matrix multiplication* and *arithmetic-logic* computations, microcode for tensor operations, etc. We show how to design the VTA microarchitecture as a sequence of transformations, and demonstrate our formalism for automatic, incremental compositional verification through symbolic simulation tools. We are not aware of other correct-by-construction flows for microarchitectures of similar complexity.

The paper makes four important contributions. *First*, we develop a new formalization of microarchitecture correctness making explicit the program transformation view. *Second*, we develop extensive foundational treatment for microarchitecture analysis based on this formalization, to support incremental, compositional verification. *Third*, we show how to mechanize our foundational analysis in Dafny. *Finally*, we demonstrate the viability of the methodology through the design correct-by-construction design of a superscalar VTA microarchitecture.

The rest of the paper is organized as follows. Section II briefly reviews related work in microarchitecture verification. We describe current practice in accelerator microarchitecture designs and explain how our approach ameliorates verification complexity in Section III. In Sections IV, V, VI, and VII, we put together the foundations of the new analysis and derive several proof rules. Section VIII shows how to formalize various advanced microarchitectural features as program transformers. Section IX discusses the mechanization of the foundations. In Section X, we discuss the use of the methodology in correct-by-construction design of the VTA microarchitecture. We conclude in Section XI. Although, the foundations are mechanized in Dafny, the paper itself does not require prior familiarity with Dafny.

II. Related Work

Analysis of microarchitectures has been a quintessential research topic in formal verification. The goal is to show that the execution of the microarchitecture model has the same behavior as the ISA. Some of the early studies have used *skewed abstraction functions* [9], [10] to map the states of the pipeline at different moments to a single ISA state. Burch and Dill [3] introduced the idea of *flushing* to reason about pipelines. Different verification strategies and notions of correspondence have been developed over the years to account for exceptions, interrupts, out-of-order and speculative executions, and many others [4], [5], [6], [11], [12], [13], [14]. Formal verification has been successfully applied on industrial microarchitectures. Goel et al. [15] used theorem proving together with automated formal tools verify the Register Transfer Level (RTL) implementation of the x86 instruction decode, translation, and execution units and the associated microcode. Kaivola et al. [16] performed formal verification of the Intel i7 processor execution engine, which involves notable manual effort in formally specifying and analyzing the intended behavior of thousands of micro-instructions. Reid et al. [17] proposed ISA-Formal, which generates an architectural model from an ARM's formal specification and then verifies an ARM implementation for equivalence against the model using bounded model checking. Similar approaches are taken in [18], [19] However, in spite of these impressive efforts, the cost of formal verification remains high: all the industrial results above reported many person-months of effort. A critical challenge in the current verification methodologies is the lack of support for interactive composition: verification technologies and even notions of correspondence are tied to specific microarchitecture features and sometimes mutually inconsistent [1], [5].

Correct-by-construction hardware design has been a focus of significant research over the last two decades. An early work applying microarchitecture transformations was by Arvind and Shen [20] that used term rewriting for microarchitecture transformations. However, this work targeted low-level architecture equivalence semantics and did not consider compositional analysis of microarchitecture features. More recently, there has been research on integrating sequential equivalence checking (SEC) with synthesis flows [21], [22], [23] and SEC between RTL and gate-level hardware designs [24]. Research has also be done on combinational equivalence checking between high-level designs in software-like languages (*e.g.*, SystemC) and RTL designs [25]. To summarize, most of the approaches follow the design-then-verify paradigm, which make the verification not scalable to the complex designs. This paper presents a correct-by-construction methodology by breaking down the design process into a sequence of small and easy-to-verify correctness-preserving steps.

To overcome these limitations, we draw inspiration from compiler design and verification, by treating a microarchitecture design as a composition of semantics-preserving program transformations. The most notable work in this area is a formally verified C compiler called CompCert [26], [27]. In addition, the Vellvm project [28] formalized LLVM's intermediate representation with a framework for reasoning about programs. Pnueli et al. proposed the notion of *translation validation* [29] for validating instances of program transformations during compilation. Necula used symbolic evaluation techniques from proof-carrying code to tackle translation validation [30].

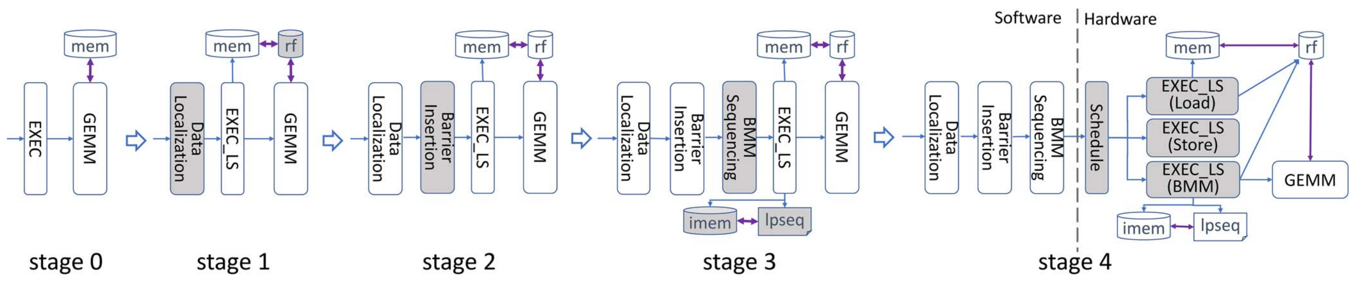


Fig. 1. Design phase 1: incremental microarchitecture feature addition for VTA.

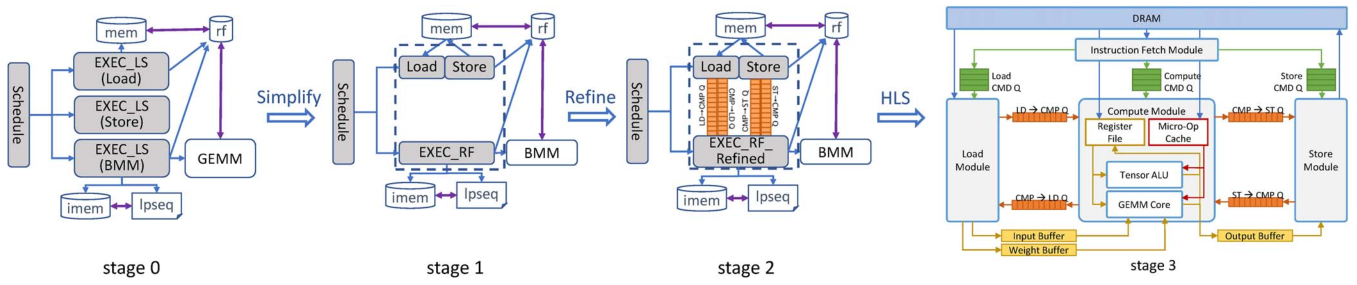


Fig. 2. Design phase 2: refinement for VTA hardware.

III. Microarchitecture Design Practice, Verification Challenges and Proposed Solution

Design of a modern accelerator microarchitecture roughly entails two distinct phases, which we refer to as (1) *feature addition* and (2) *design refinement*. Here we use the VTA accelerator design to illustrate the two phases. Note that VTA microarchitecture (Fig. 2) incorporates substantial complexity, with sophisticated optimizations including data localization, barrier insertion, mapping operations to multiple cores, etc. We use it as a motivating example to demonstrate the practical complexities involved in microarchitecture optimizations and emphasize the critical need for a verification methodology tightly integrated with design. Details of the individual optimizations will be discussed in Section X. In the first phase (Fig. 1), the microarchitecture is incrementally elaborated, starting with its architecture, with a variety of features such as data localization, barrier insertion, sequencing, and scheduling to meet its performance and energy targets. In current practice, each added feature is typically modeled in software (*e.g.*, in C or C++) and assembled into an executable model which can be simulated with target application workloads to evaluate microarchitecture decisions. Towards the end of the first phase, the features are partitioned into software and hardware components of the VTA. In the second phase, software features are refined and implemented as the compiler/runtime for VTA, while hardware features are refined and implemented as the RTL model. Fig. 2 shows examples for the hardware component, including the three `EXEC_LS` modules simplified into `Load`, `Store`, and `EXEC_RF`, and the variable-size `GEMM` (General Matrix Multiplication) module simplified to the fixed-size `BMM` (Block Matrix Multiplication) module. Currently, the hardware refinement and implementation is done through manually crafted RTL over many iterations. However, over the past decade, there have been numerous success stories in applying High-level Synthesis (HLS) to improve design efficiency while achieving optimization objectives comparable to the manual implementation [31], [32], [33].

In both phases, every decision made on a feature addition or refinement comes with an intent to incrementally improve performance or energy efficiency while preserving the functional correctness of the accelerator. Unfortunately most of these intents are poorly documented or undocumented at all, let alone formalized. This presents one of the key bottlenecks in applying formal verification to a complex design, which is tasked to show the functional equivalence between the design at one stage (e.g. stage 3 in Fig. 1) and the elaborated design at the next stage (e.g. stage 4 in Fig. 1). The reality is even worse as most of these intermediate stages are not recorded, leaving formal verification to often deal with the enormous gap between the initial design (e.g. stage 0 in Fig. 1) and the final implementation (e.g. stage 3 in Fig. 2). This precludes direct application of sequential equivalence checking techniques that rely on matching internal variables of the two designs to effectively decompose verification problems. Consequently, verification typically requires the user to hand-craft complex invariants that precisely capture the relation between the state of each microarchitecture modules and the state of each (possibly partially executed) instruction [4], [5]. The expertise and effort this requires has made it impossible for any design company to adopt formal verification to ensure the complete correctness of a product.

To overcome the barrier, we draw inspiration from the compiler world where a compiler translates a source program through a sequence of semantics-preserving program transformations (passes) to a program to be executed on a target platform. The correctness of a compiler can be established *correct-by-construction* in a scalable way by proving that all passes in the compiler are semantics-preserving and all permissible compositions of these passes are also semantics-preserving [26], [27], [28], [30]. We extend these ideas to hardware design and verification by formalizing and verifying the functionality of each feature addition in the microarchitecture as an on-the-fly program transformation on an input instruction stream (Section VI and Section VIII). The overall correctness proof is then derived using the composition theorems in Section VII. This approach also allows a seamless software and hardware co-design.

To our knowledge, this paper represents the first work to lay a solid and practical foundation for future microarchitecture design and verification in a scalar, modular and correctness-assured way. Such an approach is necessary to combat the exponentially growing cost for a new chip design with today's design practice (already exceeding \$0.5B at the $5nm$ node [34]), where verification has become the dominant factor of the cost. Companies are forced to run expensive end-to-end validation on application workloads using cloud-based simulation and emulation but still with no guarantee on complete correctness. An analysis of bugs in Intel designs and products root-causes these bugs to subtle corner-case issues in the microarchitecture features and their interactions on top of issues in manual implementations, which can be discovered and fixed much earlier with our rigorous approach. Such bugs have caused the company to lose market opportunities and incurred huge costs for the company to fix [35], [36]. We believe our approach can significantly improve design and verification efficiency and quality as well as product time-to-market, even for the current design practice by introducing feature and composition based verification collaterals.

Remark 1: As microarchitectures get increasingly complex, the distinction between different abstraction levels is getting increasingly blurred. Addition of a microarchitecture may entail successive and iterative elaborations; each elaboration breaking a complex operation into a composition of more primitive blocks. For instance, a floating-point algorithm in the execution unit of a modern microcontroller pipeline may be implemented in microcode, which entails creating a (simpler) microcontroller with its own microarchitecture elaborations (e.g., further data localization). Our methodology creates a tightly coupled iterative verification loop integrated with this iterative elaboration process. Once the elaborations are complete, RTL implementations is generated either automatically through HLS or through manually crafted RTL. Obviously, for end-to-end correctness, the generated RTL needs to be verified against the (final) feature-added microarchitecture model. We do not discuss that process here, but we note that previous work showed how to develop a corresponding correct-by-construction flow for such RTL through integrated equivalence checking [21], [22], [23]; that flow, connected with the methodology here, enables end-to-end integrated design verification.

Remark 2: Obviously, there is effort involved in the formalization and verification of individual transformations. As we discuss in Section IX, this effort can be non-trivial. However, in our experience, this cost is still significantly less than the cost of verifying an optimized microarchitecture model post facto, since the verification complexity is amortized over the different transformations. Furthermore, a transformation needs to be verified only once and can be used over and over for different microarchitecture optimizations. Finally, we extensively surveyed and analyzed a variety of industrial microarchitectures, and based on this analysis we believe that the methodology is applicable to all existing microarchitecture implementations. A key reason is that the transformer view proposed in this paper closely mimics the (informal) reasoning process employed by architects in performing the respective optimizations.

We end this section with an observation about suitability of the methodology on accelerators. The methodology itself can be applied to any microarchitecture, not only accelerators. Indeed, in our experience, verification of traditional microarchitectures can benefit from the program transformer view by enabling effective problem decomposition. However, for legacy microarchitectures, intermediate models from feature additions may not always exist and may have to be deconstructed from the implementation. On the other hand, when designing new accelerator microarchitectures, these models are designed anyhow as part of the optimization process. Furthermore, accelerator microarchitectures tend to have simpler control logic and have a lot more regular structures than general-purpose processors enabling a more straightforward comprehension of microarchitecture features.

IV. Preliminaries

We introduce formal concepts for our treatment of microarchitecture design. We use M, a finite set of variables over the set of integers $\mathbb{Z}$, to refer to the *data space* of a microarchitecture. We assume M to be partitioned into two disjoint sets M_G and M_L, *i.e.*, $M = M_G \cup M_L$. We refer to M_G and M_L as *global* and *local* memories respectively. Informally, M_G and M_L represent the architecturally and microarchitecturally visible state-holding elements respectively.

The set I of *instructions* specifies the transformation of the data space. Formally, an instruction is of the form

1) $\vec{v} := F(\vec{a})$, where $\vec{v}$ is a finite-length vector of variables over M and $\vec{a}$ is a finite-length vector of *arguments* with mixed variables and constants over $M \cup \mathbb{Z}$, called a *composite assignment* or simply *assignment*, or
2) `goto` loc `if` $C(\vec{a})$, where $loc \in \mathbb{N}$ and $C : \mathbb{N}^{|\vec{a}|} \rightarrow \mathbb{B}$, called a *branch*.[1]

We will use $\vec{d}$ to denote $\vec{a}$ if the latter does not contain any variable from M. Function Rhs returns the set of variables in $\vec{a}$ in an instruction. Function Lhs return the set of variables in $\vec{v}$ for an assignment and $[]$ for a branch. Note that composite assignments integrate both assignment and arithmetic computations. We refer to an instruction as *global* if $\vec{v}$ and $\vec{a}$ contain no variables from M_L. Intuitively, *global* instructions refer to architecturally visible machine components whereas *local* instructions can only refer to (and affect) only microarchitectural components. In practice, if a (global) instruction is implemented

[1]We treat a branch instruction separately from other instructions because it affects the control flow of the program execution which must be accounted for when reasoning about program behavior. Furthermore, separating branches enables effective comprehension of specific microarchitecture features that affect control, *e.g.*, branch prediction.

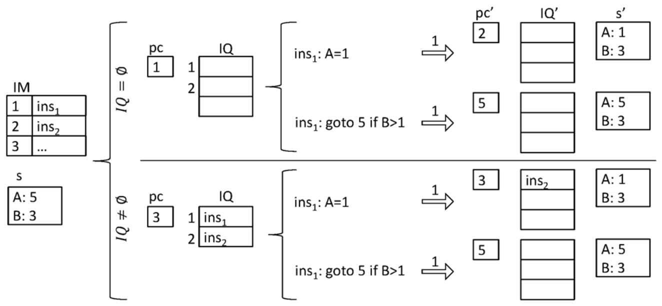


Fig. 3. Step execution example. Execute one entry (assignment or branch) from IQ or from $IM[pc]$ if IQ is empty. In terms of the microarchitecture example in Fig. 2, IM and IQ correspond to $imem$ and $Ipseq$ respectively.

by a sequence of micro-instructions, then some of the micro-instructions could be local instructions. We use $I_G \subseteq I$ to refer to the subset of global instructions in I.

A *machine program* (or simply, *program*) P is a triple $P \triangleq \langle IM, pc, IQ\rangle$ where $IM : \mathbb{N} \to I_G$, $pc \in \mathbb{N}$, and IQ is a sequence $[e_1, \ldots, e_k]$ where each $e_i \in I$. For simplicity of exposition, we assume that IM is a constant formalizing a read-only instruction memory. However, the approach can be easily generalized to account for a modifiable instruction memory. When there is no ambiguity, we will use a concise definition of a machine program $P \triangleq \langle pc, IQ\rangle$ throughout the paper.

Remark 3: The notion of "machine program" above as well as the idea of program transformations introduced below are analogous to, but different from, the traditional definition of programs used in program analysis or compiler design communities. Traditionally, one views a program as a static, finite sequence of instructions. Our formalization augments this static view with a view of a trace of a program as a dynamic stream of instructions. In particular, it is convenient to think of the instructions in the IM to constitute the static instruction sequence forming a program, while the instructions in IQ can be viewed as a stream of instructions already "fetched" from IM.

We use $|IQ|$, $IQ[i]$, and $IQ[i..j]$ to denote the length, the i-th instruction, and the subsequence of i-th to j-th instructions of IQ respectively. For the latter, We will omit i or j if $i = 1$ or $j = |IQ|$. We use $IQ_1 + IQ_2$ to denote the concatenation of sequences IQ_1 and IQ_2. The first instruction in the program, denoted as $hd(P)$, is defined as $hd(P) = \begin{cases} IQ[1] & IQ \neq [] \\ IM[pc] & otherwise \end{cases}$

V. Formal Foundation of Microarchitecture Design

We start with a few definitions to make explicit the notion of program transformation. We then define semantics preservation that forms the basis of our microarchitecture analysis.

We now formalize the step semantics of a program. The definitions below are formalization of standard interpreter semantics [37] adapted to our notation of data space and program. Here, *step execution* returns the transformed machine program after execution of one instruction. Fig. 3 illustrates an example of a step depending on the state of the IQ and the corresponding state update. If the instruction is a branch instruction and the condition evaluates to *true*, then IQ is emptied to avoid speculative commit. Note that this formalization does *not* preclude reasoning out-of-order or speculative executions. In Section VIII we show how to formalize these operations. However, it enforces the requirement that architecturally visible commits are performed in program order.

Definition 1: (State). A state s of a microarchitecture is a mapping $s : M \to \mathbb{Z}^{|M|}$. Let $s[v]$, $s[\vec{a}]$ denote the value of variable $v \in M$, the vector of values for vector $\vec{a}$ in s. A state transition is defined to occur from s to s' through instruction ins, denoted by $\{s\}ins\{s'\}$, if the following hold:

- Case 1: ins is an assignment $\vec{v} := F(\vec{a})$. Then $s' = s[\vec{v} := F(s[\vec{a}])]$, where ":=" operator means substitution. This can be also represented as $s' = eval(ins, s)$, where $eval(ins, s)$ means evaluation of ins in state s.
- Case 2: ins is a branch instruction goto loc if $C(\vec{a})$. Then $s' = s$.

Remark 4: The definition of state models architectural/microarchitectural components that are not tied to the program control. In particular, the formalization does not include pc and IQ as part of the state definition but as part of definition of the program. Doing so simplifies reasoning about programs and program equivalences by treating the program itself and the components involved in its control as one formal object. Note that the objective of the foundation is to view microarchitecture features as equivalence-preserving transformers of program, and the definition of state is targeted to enable establishing of the equivalence relation.

Definition 2: (Step Execution). A step-execution of program $P \triangleq \langle pc, IQ\rangle$ in state s is program $P' \triangleq \langle pc', IQ'\rangle$ and state s', denoted by $\langle P', s'\rangle = step(P, s)$, such that $\{s\}hd(P)\{s'\}$, and

- Case 1 $hd(P)$ is `goto` loc `if` $C(\vec{a})$ and $C(s[\vec{a}]) = true$, then $pc' = loc$, $IQ' = []$.
- Case 2: Otherwise, $pc' = pc$, $IQ' = IQ[2..]$ when $IQ \neq []$ and $pc' = pc + 1$, $IQ' = []$ when $IQ = []$.

Definition 3: (Prefix Execution). A k-step prefix execution of program $P \triangleq \langle pc, IQ\rangle$ in state s, denoted by $\langle P', s'\rangle = pstep^k(P, s)$, is recursively defined as

- $\langle P', s'\rangle = \langle P, s\rangle$, if $k = 0$ or $IQ = []$
- $\langle P', s'\rangle = pstep^{k-1}(P'', s'')$ and $\langle P'', s''\rangle = step(P, s)$ for some program P'' and state s'', otherwise.

Remark 5: Definition 3 is formalized to return the state after execution of k (microarchitectural) steps. The condition $IQ = []$ is included in Definition 3 to account for the following two cases: (1) $k > |IQ|$ or (2) $IQ[i]$ is a branch instruction `goto` loc `if` $C(\vec{a})$ for some $i < k$ and $C(s_i[\vec{a}])$ is true where s_i is the state at step i. In the latter case, $IQ' = []$ and $pc' = loc$.

Fig. 4 shows an example of a K-step execution. Informally, a K-step execution is like partially flushing the IQ where at most k entries are evaluated and stops when $IQ = []$. No instruction is executed from IM. The example shows both cases where all $j \leq k$ instructions are evaluated assuming no other instruction from 2 to $j - 1$ updates A and B, or there are $> k$ entries in IQ resulting in entries remaining in the IQ after evaluating k instructions.

Finally, we define program equivalence.

Definition 4: (Program Equivalence). Program P_1 in state s_1 and program P_2 in state s_2 are equivalent, denoted by $\langle P_1, s_1\rangle \equiv \langle P_2, s_2\rangle$, if

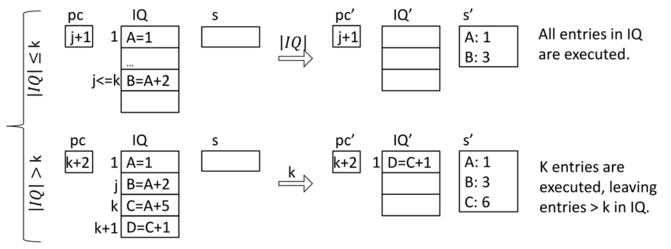
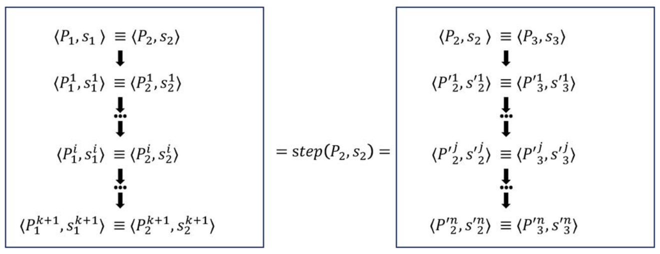


Fig. 4. K-step prefix execution example. Executes $min(|IQ|, k)$ entries from the IQ.



$\langle P_1, s_1 \rangle \equiv \langle P_2, s_2 \rangle$
$\langle P_1^1, s_1^1 \rangle \equiv \langle P_2^1, s_2^1 \rangle$
$\langle P_1^i, s_1^i \rangle \equiv \langle P_2^i, s_2^i \rangle$
$\langle P_1^{k+1}, s_1^{k+1} \rangle \equiv \langle P_2^{k+1}, s_2^{k+1} \rangle$
$= step(P_2, s_2) =$
$\langle P_2, s_2 \rangle \equiv \langle P_3, s_3 \rangle$
$\langle P'^1_2, s'^1_2 \rangle \equiv \langle P'^1_3, s'^1_3 \rangle$
$\langle P'^j_2, s'^j_2 \rangle \equiv \langle P'^j_3, s'^j_3 \rangle$
$\langle P'^n_2, s'^n_2 \rangle \equiv \langle P'^n_3, s'^n_3 \rangle$

Fig. 6. Steps in the inductive proof of transitivity.

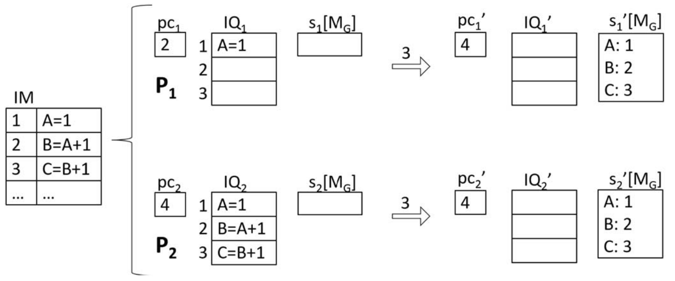


Fig. 5. Execution equivalence example. P_1 and P_2 are equivalent after stepping 3 times.

1) If $IQ_1 = IQ_2 = []$, then $pc_1 = pc_2, s_1[M_G] = s_2[M_G]$,
2) If $IQ_1 \neq []$ or $IQ_2 \neq []$, then $s_1[M_G] = s_2[M_G]$ and one of the following must be true:
   a) (P_1 advances) $IQ_1 \neq [] \wedge \langle P_1', s_1' \rangle \equiv \langle P_2, s_2 \rangle$,
   b) (P_2 advances) $IQ_2 \neq [] \wedge \langle P_1, s_1 \rangle \equiv \langle P_2', s_2' \rangle$,
   c) (Lockstep advance) $\langle P_1', s_1' \rangle \equiv \langle P_2', s_2' \rangle$

   where $\langle P_1', s_1' \rangle = step(P_1, s_1)$, $\langle P_2', s_2' \rangle = step(P_2, s_2)$.

Remark 6: Case 1 states that P_1 and P_2 are the same program, and they start from the same architectural state. Case 2 reduces two programs P_1 and P_2 to case 1 by executing the instructions from the two programs such that the sequences of architecture changes from the execution are the same under stuttering (one of them may execute additional instructions from the memory but not both). Fig. 5 shows an example of two programs P_1 and P_2 where P_2 more aggressively prefetches instructions into the IQ are equivalent after stepping three times.

Remark 7: For the reader familiar with specification and verification of distributed reactive systems, the notion above is analogous to bisimulation equivalence under stuttering [38], [39]. Bisimulation is a notion of correspondence where we consider two systems equivalent when they produce the same observable behavior at each transition. Stuttering relaxes the bisimulation correspondence by requiring that the behaviors match only on specific transitions (referred to as "commitment steps") while other transitions produce no change in the observable components. Our notion is similar to stuttering bisimulation where the commitment steps correspond to updates to architectural components of the states.

The following theorem is an easy consequence of the definition.

Theorem 1: (Transitivity). If $\langle P_1, s_1 \rangle \equiv \langle P_2, s_2 \rangle$ and $\langle P_2, s_2 \rangle \equiv \langle P_3, s_3 \rangle$ where $P_i \triangleq \langle pc_i, IQ_i \rangle$ for $i = 1, 2, 3$, then $\langle P_1, s_1 \rangle \equiv \langle P_3, s_3 \rangle$.

Proof: We prove by induction on the numbers of steps k and m as defined in Definition 4 to establish $\langle P_1, s_1 \rangle \equiv \langle P_2, s_2 \rangle$ and $\langle P_2, s_2 \rangle \equiv \langle P_3, s_3 \rangle$. □

Base case: $k = 0$, $m = 0$. By the first case of Definition 4, we have $IQ_1 = IQ_2 = IQ_3 = []$, $pc_1 = pc_2 = pc_3$, and $s_1[M_G] = s_2[M_G] = s_3[M_G]$. Therefore, by Definition 4, $\langle \langle IM, pc_1, [] \rangle, s_1 \rangle \equiv \langle \langle IM, pc_3, [] \rangle, s_3 \rangle$

Inductive case: Assume for all programs P_1, P_2, P_3 that require $\le k$ steps and $\le m$ steps to establish equivalences $\langle P_1, s_1 \rangle \equiv \langle P_2, s_2 \rangle$ and $\langle P_2, s_2 \rangle \equiv \langle P_3, s_3 \rangle$, we have $\langle P_1, s_1 \rangle \equiv \langle P_3, s_3 \rangle$. There are three subcases to consider:

- $k+1$ steps and $n \le m$ steps to establish $\langle P_1, s_1 \rangle \equiv \langle P_2, s_2 \rangle$ and $\langle P_2, s_2 \rangle \equiv \langle P_3, s_3 \rangle$.
- $n \le k$ steps and $m+1$ steps to establish $\langle P_1, s_1 \rangle \equiv \langle P_2, s_2 \rangle$ and $\langle P_2, s_2 \rangle \equiv \langle P_3, s_3 \rangle$.
- $k+1$ steps and $m+1$ steps to establish $\langle P_1, s_1 \rangle \equiv \langle P_2, s_2 \rangle$ and $\langle P_2, s_2 \rangle \equiv \langle P_3, s_3 \rangle$.

We will prove the first sub-case. The other two can be proved similarly. Consider the two sequences of advances in Fig. 6 that establish $\langle P_1, s_1 \rangle \equiv \langle P_2, s_2 \rangle$ (left) and $\langle P_2, s_2 \rangle \equiv \langle P_3, s_3 \rangle$ (right). Let $i, j > 0$ be the steps where $\langle P_2^i, s_2^i \rangle = step(P_2, s_2) = \langle P_2'^j, s_2'^j \rangle$ (middle). Since the two sub-sequences starting from $\langle P_1^i, s_1^i \rangle \equiv \langle P_2^i, s_2^i \rangle$ and $\langle P_2'^j, s_2'^j \rangle \equiv \langle P_3'^j, s_3'^j \rangle$ are of length $\le k$ and $\le m$ respectively, by the induction hypothesis, $\langle P_1^i, s_1^i \rangle \equiv \langle P_3'^j, s_3'^j \rangle$.

We now construct the sequence starting from pair $\langle P_1, s_1 \rangle$ and $\langle P_3, s_3 \rangle$ that first step-executes $\langle P_1, s_1 \rangle$ until $\langle P_1^i, s_1^i \rangle$ but keep $\langle P_3, s_3 \rangle$ unchanged, and then step-executes $\langle P_3, s_3 \rangle$ until $\langle P_3'^j, s_3'^j \rangle$ but keep $\langle P_1^i, s_1^i \rangle$ unchanged. Then we have $s_1^n[M_G] = s_3'^n[M_G]$ for every pair $\langle P_1^n, s_1^n \rangle$ and $\langle P_3'^n s_3'^n \rangle$ in this sequence. Therefore, by case 2 of Definition 4, $\langle P_1, s_1 \rangle \equiv \langle P_3, s_3 \rangle$.

The following two lemmas will be useful in verifying many of the microarchitecture properties in Section VIII.

Lemma 1: (Common Prefix). Let $P_1 \triangleq \langle pc_1, IQ_1 \rangle$ *and* $P_2 \triangleq \langle pc_2, IQ_2 \rangle$ such that for some $k \le min(|IQ_1|, |IQ_2|)$, $IQ_1[i] = IQ_2[i]$ for all $i \le k$. Then $\langle P_1, s \rangle \equiv \langle P_2, s \rangle$ if $pstep^k(P_1, s) \equiv pstep^k(P_2, s)$ for all state s.

Proof: The k-step prefix executions of both programs from s will end up with the same end state following the lockstep advances in Definition 4. Therefore the correctness can be established by a simple induction on k. □

Lemma 2: (Common Sub-State). Let s_1 and s_2 be two states such that $s_1[v] = s_2[v]$ for all $v \in V$ where $M_G \subseteq V \subseteq M$. Let $P \triangleq \langle pc, IQ \rangle$ be a program such that $Rhs(ins) \subseteq V$ for every ins in IQ. Then $\langle P, s_1 \rangle \equiv \langle P, s_2 \rangle$.

Algo. 1: FETCH($P \triangleq \langle pc, IQ\rangle, s$)

```
1 $ins \leftarrow IM[pc]$
2 if $ins$.MATCH($goto\, loc\, if\, C(\vec{a})$) $\wedge\, BP(ins)$ then
3     $ins \leftarrow goto\, pc+1\, if\, not\, C(\vec{a})$
4     $IQ' \leftarrow IQ + [ins]$
5     $pc' \leftarrow loc$
6 else
7     $IQ' \leftarrow IQ + [ins]$
8     $pc' \leftarrow pc + 1$
9 return $\langle pc', IQ'\rangle$
```

Proof: We prove by induction on $|IQ|$. □

Base case: $IQ = []$. It follows case 1 of Definition 4.

Inductive case: assume the claim holds for all instruction queues of length $\leq k$. Consider an IQ with $k+1$ instructions. Let s_1' and s_2' be the two states after step-executing P in s_1 and s_2. It is easy to see that $s_1'[v] = s_2'[v]$ for all $v \in V$. Following the hypothesis and case 2 (c) of Definition 4, the lemma holds for IQ.

Definition 5: (Program Transformer). Let $\wp$ be the set of all programs and S be the set of all states. Let $T : S \rightarrow \wp \rightarrow \wp$ be a function that translates a program to another program for a given state. T is a program transformer, if for all $s \in S$, $p \in \wp$, $\langle P, s\rangle \equiv \langle T(s, P), s\rangle$.

VI. Basic Microarchitecture Features as Program Transformers

A key contribution of our work is the observation that microarchitectural features can be appropriately viewed as incremental optimizations in processor design via program transformation. The features covered in this section are ”basic” microarchitectural features such as instruction fetch and data forwarding. We will discuss more advanced features after formalizing compositional rules. Obviously, given the number of sophisticated microarchitecture features available in a modern microprocessor it is impossible to provide explicit analysis of the entire spectrum of features.[2] Instead we focus illustrating the approach through a selection of representative features.

A. Instruction Fetch

We view instruction fetch as a program transformation that updates IQ with the prefix of the program in IM pointed to by pc. Algorithm 1 defines the corresponding microarchitecture operation.[3] Here BP is a non-deterministic predicate that predicts the branch will be taken or not. Note that the non-determinism enables compositional verification of any branch prediction algorithm independently from the Fetch functionality.

[2]For simplicity of presentation, the formalization we show here is somewhat simplified from the actual transformer implementations deployed in our case studies. However, the proofs discussed here are generalizable to our more elaborate transformation implementations.

[3]Throughout the paper, we use standard algorithmic constructs to define the transformations, and show them as pseudocode. The pseudocode shown here is an abstract representation of the transformer implementation.

Algo. 2: DATAFORWARD($P \triangleq \langle pc, IQ\rangle, s$)

```
1 $i \leftarrow$ any number in $[1, |IQ|]$
2 $ins \leftarrow IQ[i]$
3 foreach $v \in Rhs(ins)$ do
4     if $v \in M_L \wedge \forall j.\, j < i \implies v \notin Lhs(IQ[j])$ then
          $ins \leftarrow$ REPLACERHS($ins, v, s[v]$)
5 $IQ' \leftarrow IQ[..(i-1)] + [ins] + IQ[(i+1)..]$
6 return $\langle pc, IQ'\rangle$
```

Theorem 2: (Fetch is a program transformer).

Proof: Let $\langle\langle pc_e, []\rangle, s_e\rangle = pstep^{|IQ|}(\langle pc, IQ\rangle, s)$ and $\langle\langle pc_e', IQ_e'\rangle, s_e\rangle = pstep^{|IQ|}(\langle pc', IQ'\rangle, s)$. Since $IQ' = IQ + [ins]$, we have $s_e = s_e'$. By Lemma 1 we only need to prove $\langle\langle pc_e, []\rangle, s_e\rangle \equiv \langle\langle pc_e', IQ_e'\rangle, s_e\rangle$.

The case is straightforward if a branch is taken in IQ, as $IQ_e' = []$, $pc_e = pc_e'$. Otherwise, by Definition 2, $pc_e = pc$, $pc_e' = pc'$. There are two possibilities:

- $pc_e' = pc + 1$ and $IQ_e' = [IM[pc]]$, or
- $pc_e' = loc$, $IM[pc] =$ `goto loc if` $C(\vec{a})$ and $IQ_e' =$ [`goto pc+1 if not` $C(\vec{a})$].

For both cases, $step(\langle pc_e, []\rangle, s_e) \equiv step(\langle pc_e', IQ_e'\rangle, s_e)$. □

B. Data Forwarding

The Fetch example, albeit illustrative, is rather simple. To see the power of the program transfer view, it is illustrative to consider data forwarding which reduces or eliminates the need for instructions to stall for the output of previous instructions to update the architecturally visible state components; instead, output from the previous computation is fed directly as operand for the subsequent instruction. Analysis of pipelines with data forwarding involves subtle invariants since it must account for a variety of data hazards, *e.g.*, a significant part of Sawada's proof [4] involves defining invariants characterizing hazard-free data forwarding. However, from the perspective of program transformation, we can simply view it as an operation for constant forwarding. Algorithm 2 defines the data-forwarding operation.

The algorithm selects an instruction ins from IQ and replaces any local variable v (i.e. $v \in M_L$) from $Rhs(ins)$ with $s[v]$ if v is not updated by any preceding assignment.

Theorem 3: (DataForward is a program transformer).

Proof: The proof is trivial if a branch is taken in step-executing instructions in $IQ[..(i-1)]$. Otherwise, let s' be the state after step-executing these instructions. by Lemma 1. we need to prove $\langle\langle pc, [ins] + IQ_e\rangle, s'\rangle \equiv \langle\langle pc, [ins'] + IQ_e\rangle, s'\rangle$ where $IQ_e = IQ[(i+1)..]$. Since steps $3-4$ in the algorithm guarantee the step-executions of ins and ins' in s' resulting in the same state s'', we need to prove $\langle\langle pc, IQ_e\rangle, s''\rangle \equiv \langle\langle pc, IQ_e\rangle, s''\rangle$ which is trivial. □

VII. Composition of Program Transformers

A key property of program transformers is their compositionality. Theorem 4 defines a set of composition rules. These rules enable us to derive correctness of advanced microarchitecture features by viewing them as a composition of simpler transformers.

Theorem 4: Let $T_1 : S \to \wp \to \wp$ and $T_2 : S \to \wp \to \wp$ be two program transformers. Then the following are also program transformers.

- Non-deterministic composition $T = T_1|T_2$;
- Sequential composition $T = T_1; T_2$;
- Star composition $T = T_1^*$;
- Conditional composition $T = if\ C(V)$ then T_1 else T_2 for any condition $C(V)$; and
- Parallel composition $T = T_1||T_2$, if range(T_1) and range(T_2) are non-overlapping where range(T_i) is the range of IQ for T_i.

Proof: Let $P = \langle IM, pc, IQ\rangle$ and s be a state.

Non-deterministic composition: Let $P' = T(s, P)$. By definition, either $P' = T_1(s, P)$ or $P' = T_2(s, P)$. Since T_1 and T_2 are transformers, $\langle P, s\rangle \equiv \langle P', s\rangle$. Therefore, $T = T_1|T_2$ is a transformer.

Sequential composition: Let $P' = T_1(s, P)$ and $P'' = T_2(s, P')$. Then $\langle P, s\rangle \equiv \langle P', s\rangle$ and $\langle P', s\rangle \equiv \langle P'', s\rangle$. By Theorem 1, $\langle P, s\rangle \equiv \langle P'', s\rangle$. Therefore, $T = T_1; T_2$.

Star composition: Obviously T_1^0 is a transformer as $\langle P, s\rangle \equiv \langle P, s\rangle$. Assume T_1^k is a transformer, then the sequential composition $T_1^{k+1} = T_1^k; T_1$ is a transformation.

Conditional composition: The proof is similar to non-deterministic composition.

Parallel composition: Since $range(T_1)$ and $range(T_2)$ are non-overlapping and since transformers do not change global variables, $T_1||T_2$ will be equivalent to $T_1; T_2$. □

Most composition rules have analogues in bisimulation relation. One unique rule is parallel composition, which is crucial for compositional analysis of microarchitecture with components for performing overlapped concurrent executions. As an example, consider the composition `Fetch` || `Rename` || `OOO` (Out-Of-Order execution). Most modern pipeline functionality includes this composition. In Section VIII we will prove that `Rename` and `OOO` are program transformers. Since `Fetch` appends new global instructions from IM to IQ, `Rename` only works on global instructions in IQ, the parallel composition rule enables us to derive this composition as a transformer as long as `OOO` does not update any global instructions.

To understand the role of program transformations to enable microarchitecture designs as incremental optimizations, consider a microarchitecture design in Algorithm 3 that defines the computation for a single "clock cycle". It starts with a sequence of program transformations through a composed program transformer in steps 1 - 7 and ends with executing the first N_4 instructions from IQ in steps 9 - 17.

We can view these fragments in the algorithm as primitive functionalities or building blocks for the microarchitecture. For an actual microarchitecture design, high performance is achieved by making these sequentially composed blocks *parallel* through (1) segmenting (pipelining) IQ into non-overlapping pipe stages, separated by *latches*, and assigning blocks to operate on specific pipe stages, and then (2) implementing the blocks for each pipe stage using one combinational logic block between latches. Furthermore, all heavy computations will be performed as a part of transformations in steps 1 - 8. Steps 9 - 17 only execute instructions that are simple

Algo. 3: uArchDesign($P \triangleq \langle pc, IQ\rangle, s$)

```
1  $P' \leftarrow P$;
2  repeat $N_1$ times
3      $P' \leftarrow$ Fetch$(P', s)$
4  repeat $N_2$ times
5      $P' \leftarrow$ Rename$(P', s)$
6  repeat $N_3$ times
7      $P' \leftarrow$ Ooo$(P', s)$
8  $s' \leftarrow s$
9  repeat $N_4$ times
10     if $P'.IQ \neq []$ then
11         $ins \leftarrow P'.IQ[1]$
12         $P' \leftarrow \langle P'.pc, P'.IQ[2..]\rangle$
13         if $ins$.Match$(\vec{v} := F(\vec{a}))$ then
14             $s' \leftarrow eval(ins, s')$
15         else if $ins$.Match$(goto\ loc\ if\ C(\vec{a}))$ then
16             if $C(s'[\vec{a}])$ then
17                 $P' \leftarrow \langle loc, []\rangle$
18 return $\langle P', s'\rangle$
```

assignments of form $\vec{v} = \vec{a}$, and thus are called the *retirement stage*.

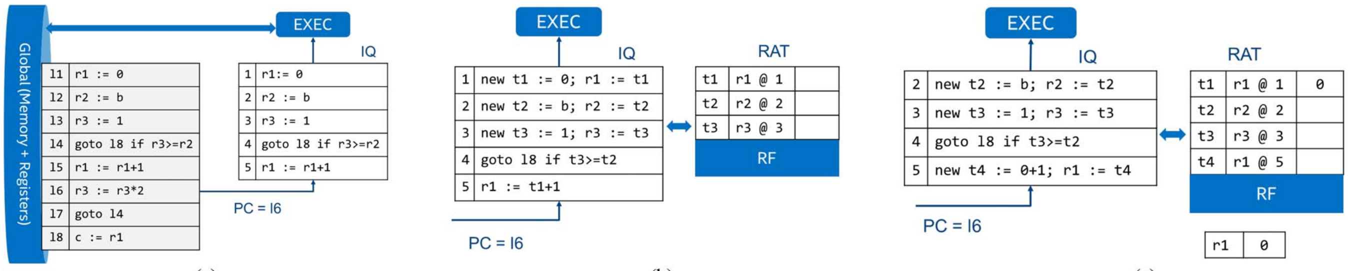
Finally, note that the correctness holds independent of the values of the parameters in the primitive operations (*e.g.*, N_1 through N_4) or the specifics of the decisions for branch prediction and instruction selection. These characteristics enable a scalable modular design and verification of a complex microarchitecture. Unlike traditional verification approaches, the only proof obligation in our framework entails showing the correctness of the primitive application and the application of composition rules; the rest is derived from the foundations through composition rules.

VIII. Advanced Microarchitecture Features

The program transformation view provides a powerful tool for analyzing microarchitectures. In this section we will show how to reconcile various advanced microarchitecture features with this foundation. Note that these features are "advanced" in the sense that they are more complex than the basic features discussed in Section VI and formal verification of microprocessors with these features have traditionally been difficult (see Section II); however, they are well-established features and have been deployed in processor architectures for several decades.

A. Data Localization

Data Localization is the approach to hide memory latency by moving the data close to the computation ahead of time. There are many ways to implement data localization. A common form of data localization in OOO microprocessors is *Register Allocation and Renaming*. Architecturally, when an instruction refers to the data at a remote location, the processor transposes the reference to a specific physical register on the fly [40]. We incorporate this view as a program transformation by allocating



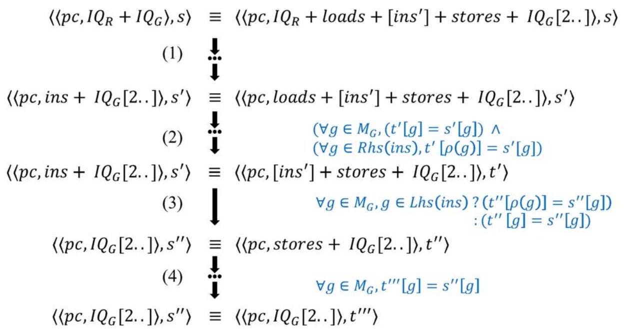
```
(a)
11 r1 := 0
12 r2 := b
13 r3 := 1
14 goto 18 if r3>=r2
15 r1 := r1+1
16 r3 := r3*2
17 goto 14
18 c := r1

IQ
1 r1:= 0
2 r2 := b
3 r3 := 1
4 goto 18 if r3>=r2
5 r1 := r1+1

(b)
IQ
1 new t1 := 0; r1 := t1
2 new t2 := b; r2 := t2
3 new t3 := 1; r3 := t3
4 goto 18 if t3>=t2
5 r1 := t1+1

RAT
t1 r1 @ 1
t2 r2 @ 2
t3 r3 @ 3

(c)
IQ
2 new t2 := b; r2 := t2
3 new t3 := 1; r3 := t3
4 goto 18 if t3>=t2
5 new t4 := 0+1; r1 := t4

RAT
t1 r1 @ 1 0
t2 r2 @ 2
t3 r3 @ 3
t4 r1 @ 5

r1 0
```

Fig. 7. Register renaming example: (a) program for $pow2(b)$ with an unrolled prefix in IQ; (b) program after register renaming for instructions 1,2,3 in IQ; (c) program after executing instruction 1, data forwarding and register renaming to instruction 5.

```
Algo. 4: DATALOCALIZE($P \triangleq \langle pc, IQ_R + IQ_G\rangle, s$)
1  if $IQ_G = []$ then return $P$
2  $ins \leftarrow IQ_G[1]$
3  $\rho \leftarrow \emptyset$                          ▷ global to local mapping
4  foreach $i \in [1, |IQ_R|]$ do
5      if $IQ_R[i]$.MATCH$(l := g) \wedge l \in M_L \wedge g \in M_G$ then
           $\rho[g] \leftarrow l$
6  $\langle loads, ins', stores\rangle \leftarrow \langle \emptyset, ins, \emptyset\rangle$
7  foreach $g \in$ RHS$(ins)$ do
8      if $g \notin domain(\rho)$ then
9          $l \leftarrow$ any $v \in M_L : v \notin range(\rho)$   ▷ allocation
10         $\rho[g] \leftarrow l$
11         $loads \leftarrow loads + [l := g]$
12     $ins' \leftarrow$ REPLACERHS$(ins', g, \rho[g])$
13 foreach $g \in$ LHS$(ins)$ do
14     $l \leftarrow$ any $v \in M_L : v \notin range(\rho)$       ▷ allocation
15     $ins' \leftarrow$ REPLACELHS$(ins', g, l)$
16     $stores \leftarrow stores + [g := l]$
17     $\rho[g] \leftarrow l$
18 $IQ'_R \leftarrow IQ_R + loads + [ins'] + stores$
19 return $\langle pc, IQ'_R + IQ_G[2..]\rangle$
```

a temporary variable from local memory M_L to locally reference a member of M_G. *Scratch Buffers* constitute another form of data localization where the data for computations are first loaded from memory to scratch buffers where subsequent computations can load data and store (intermediate) results more efficiently. The final results are then stored back to memory. Finally, the entire *Hierarchical Cache System* constitute a form of data localization albeit at a much larger scale. The invariant maintained by the cache system ensures that the computation can get data from any cache line in the system as long as it maintains the same data in the corresponding memory location. Obviously it is much more complex to manage the invariant and to prevent data access during transient phases through the implementation of a protocol such as cache coherence.

Remark 8: Establishing data localization as a transformer allows its use as a building block to construct the cache system in a correct-by-composition way. Furthermore, it enables the separation of the concerns in design and refinement. In the case of cache coherence, for example, our foundation provides a generic formalization of the specification of cache coherence



$\langle\langle pc, IQ_R + IQ_G\rangle, s\rangle \equiv \langle\langle pc, IQ_R + loads + [ins'] + stores + IQ_G[2..]\rangle, s\rangle$
(1)
$\langle\langle pc, ins + IQ_G[2..]\rangle, s'\rangle \equiv \langle\langle pc, loads + [ins'] + stores + IQ_G[2..]\rangle, s'\rangle$
(2)
$(\forall g \in M_G, (t'[g] = s'[g]) \wedge$
$(\forall g \in Rhs(ins), t'[\rho(g)] = s'[g])$
$\langle\langle pc, ins + IQ_G[2..]\rangle, s'\rangle \equiv \langle\langle pc, [ins'] + stores + IQ_G[2..]\rangle, t'\rangle$
(3)
$\forall g \in M_G, g \in Lhs(ins)\ ?\ (t''[\rho(g)] = s''[g])$
$: (t''[g] = s''[g])$
$\langle\langle pc, IQ_G[2..]\rangle, s''\rangle \equiv \langle\langle pc, stores + IQ_G[2..]\rangle, t''\rangle$
(4)
$\forall g \in M_G, t'''[g] = s''[g]$
$\langle\langle pc, IQ_G[2..]\rangle, s''\rangle \equiv \langle\langle pc, IQ_G[2..]\rangle, t'''\rangle$

Fig. 8. Steps in the localization proof.

protocol as necessary to enable composition with other microarchitecture features for overall microarchitecture correctness.

Algorithm 4 defines the micro-architecture operation for the data localization. It assumes that IQ is partitioned into two parts $IQ_R + IQ_G$ where the tail part IQ_G before renaming contains only global instructions, and the head part IQ_R after renaming contains no global instructions.

Steps 3-5 build the most recent renaming ρ from a global variable to a local variable. This is necessary as a global variable may get renamed multiple times during its lifetime in IQ. Steps 7-12 rename every global variable g in $RHS(ins)$ to its corresponding local variable $\rho[g]$ and allocate one l if no such local variable exists. In the latter case, the data stored in g will be pre-loaded to l. Similarly, Steps 13-18 rename every global variable g in $LHS(ins)$. Doing so will make all computations depend only on local variables.

Remark 9: To optimize searches in steps 4 - 5 of Algorithm 4 as well as in step 4 of Algorithm 2, a data structure called RAT (Register Aliasing Table) is created to keep track of the most recent renaming of a global variable and the pool of free local variables. RAT is updated during this algorithm and the step executions in Algorithm 3.

Theorem 5: (DataLocalization is a program transformer).

Proof: Consider the four stages of advances in Fig. 8. In stage 1, the instructions in IQ_R are executed in both programs from state s, resulting in the same state s'. In stage 2, the instructions in $load$ in the transformed program are executed from s', resulting in state t'. For every global variable g, $s'[g] = t'[g]$. Further, every local variable l associated with a global variable g in $Rhs(ins)$, $t'[l] = t'[g]$. In stage 3, ins and ins' are executed from s' and t' respectively. For every global variable g, if it appears in $Lhs(ins)$, $t''[l] = s''[g]$

Algo. 5: OOO($P \triangleq \langle pc, IQ\rangle, s$)

```
1 $i \leftarrow$ FINDANY($IQ, \lambda.\, ins \Rightarrow$ RHS($ins$) $= \emptyset$)
2 if $i = nil$ then return $P$
3 $ins' \leftarrow$ REPLACERHS($IQ[i]$, EVALRHS($IQ[i], s$))
4 $IQ' \leftarrow IQ[..(i-1)] + ins' + IQ[(i+1)..]$    ▷ replace instruction
5 return $\langle pc, IQ'\rangle$
```

where l is the local variable associated with g. Otherwise, $t''[g] = s''[g]$. In stage 4, the instructions in *store* are executed, resulting in the same global memory content in s'' and t'''. By Lemma 2, $\langle\langle pc, IQ_G[2..]\rangle, s''\rangle \equiv \langle\langle pc, IQ_G[2..]\rangle, t'''\rangle$. By Definition 4, the original program and transformed program are equivalent. □

Fig. 7 shows how this works with the example program to compute the power of 2. The data localization transformation creates a temporary local variable (in this case `t1, t2, t3, t4`) for each global variable (`r1, r2, r3, r4` respectively) and modify each reference to a global variable in subsequent instructions in IQ to its associated local variable (e.g., replace `r2` and `r3` in instruction 4 with `t2` and `t3` in (b)).

B. Out-of-Order Execution

Out-of-Order (OOO) execution is one of the most complex features in modern microarchitectures. It enables execution of instructions in a sequence different from program order based on the availability of input operands and execution unit resources. From the perspective of verification, it complicates notions of correspondence and proof strategies, *e.g.*, Burch and Dill notions are not applicable for OOO microarchitectures. However, the program transformation view accounts for OOO in a straightforward manner by viewing it as an optimization for evaluating instructions whenever an instruction is "ready", *i.e.*, when its operands have been evaluated to a constant. For instance, in Fig. 7(c), instruction 5 is ready and can be simplified through OOO transformation. Algorithm 5 provides a formal description of the transformation. When all variables in the right-hand-side of an instruction are constants, then this instruction is "ready". Its right-hand-side will be replaced with the evaluation of the operations. In real world implementations, the ready instructions will normally be put into a ready queue, scheduled by the scheduler, and replaced once the simplification is done.

Theorem 6: (OOO is a program transformer).

The proof of the theorem is straightforward as the only difference between IQ and IQ' is the instruction $IQ[i]$ is replaced by ins' where the right-hand-side function with only constants is evaluated and replaced by the result.

Remark 10: In practice, we partition this algorithm into two parts: (1) we schedule the instructions into a 'ready' queue; and (2) we pick the first instruction from the queue, and evaluate/execute the instruction. The proof of this more advanced algorithm would be similar, since any instruction in the 'ready' queue will have its right-hand-side variables being constants.

```
 1 function find_max(a: seq<int>): (max: int)
 2   decreases |a|
 3   requires |a| > 0                          // precondition
 4   ensures max in a                          // postcondition
 5   ensures ∀ x: int · x in a ⇒ max ≥ x      // postcondition
 6 {
 7   if |a| == 1 then a[0]
 8   else
 9     var k ← find_max(a[1..]);
10     assert a == [a[0]] + a[1..];
11     if a[0] ≥ k then a[0] else k
12 }
```

Fig. 9. An example function in Dafny to find the maximum from a given non-empty sequence of integers.

Remark 11: The OOO transformer can be extended to support superscalar architecture with multiple execution units. Instead of a single 'ready' queue, the idea is for each unit to maintain a linear sequence of instructions in its local instruction memory with its own program counter. An instruction is sent to a unit based on the type of instructions the unit supports and its availability, if the instruction is ready or it is only data-dependent to an instruction that is already sent to the unit; an instruction can also be sent to a different unit as long as a barrier instruction is inserted to ensure that all the instructions it depends on are executed before its execution. Correspondingly, OOO can also be extended to co-processors and accelerators where more complex computations are offloaded.

IX. MECHANIZATION IN DAFNY

We have mechanized the foundations in Dafny [8]. In this section, we provide an overview of the mechanization. The goal is not to explain the mechanization in elaborate detail but rather to give the reader familiar with formal semantics and proof mechanization a flavor of how to mechanize the foundations in a typical proof assistant. The reader interested in the foundations but not in the mechanization can skip this section without loss of continuity.

Dafny is a verification-aware programming language with built-in constructs for specification. It allows users to write specification, implementation and proofs of programs. Fig. 9 shows an example function that finds the maximum from a given a non-empty sequences of integers. Line 2 hints Dafny to prove the termination with this recursive function. Line 3 shows the precondition, which needs to be satisfied when calling this function. Line 4-5 shows the postconditions to hold upon the completion of this function. Dafny will verify if the function body (Line 7-11) indeed correctly implements the specification. The Dafny static program verifier uses Z3 Satisfiability Modulo Theories (SMT) solver to verify the correctness of the programs. Dafny is also able to generate executable (in C++, Go, C#, etc.) models after the verification, which is helpful for simulating our processor design with concrete instructions.

Fig. 10 shows the formalization of the foundational concepts. We define memory as a *map* from addresses to memory element `MemElem`. An address can be either *local* or *global*, defined as `LAddr` or `GAddr`, respectively. An *instruction* can be an arithmetic instruction `Arith`, a `Move`, an unconditional jump `Jmp`, conditional jump instructions `JZ` and `JNZ`, and a `NOP`. A program *state* consists of a local memory and a global memory.

```
type MemElem(!new) // Opaque Type, non-heap based
type Mem = m: imap<nat, MemElem> | ∀ a: nat · a in m

datatype Operator = Operator(op: string)
datatype Addr = LAddr(addr: nat) | GAddr(addr: nat)
datatype Operand = Addr(addr: Addr) | Imm(imm: MemElem)

datatype Instr =
  | Arith(dst: Addr, op: Operator, srcs: seq<Operand>)
  | Move(src: Operand, dst: Addr)
  | Jmp(loc: nat)
  | JZ(loc: nat,  cond: Addr)
  | JNZ(loc: nat, cond: Addr)
  | NOP

type LMem = m: imap<nat, MemElem> | ∃ a: nat · a !in m

datatype State = State(lmem: LMem, gmem: Mem)

datatype Program_t = Program_t(
  im: imap<nat, Instr>, pc: nat, iq: seq<Instr>)
{
  predicate well_formed() {
    ∧ (∀ pc: nat · pc in im)
    ∧ (∀ ins · ins in im.Values ∧ is_gmem_instr(ins))
  }
}
type Program = p: Program_t | p.well_formed() witness *
```

(a)

```
function step(p: Program, s: State): (Program, State, Instr) {
  var (ins, iq') ← if |p.iq| ≠ 0 then (p.iq[0], p.iq[1..])
                   else (p.im[p.pc], []);
  if ins.Jmp? then
    (p.(pc ← ins.loc, iq ← []), s, ins)
  else if ins.JZ? ∨ ins.JNZ? then
    // taking the branch?
    if ∨ (ins.JZ? ∧ is_memelem_zero(s.get_mem(ins.cond)))
       ∨ (ins.JNZ? ∧ !is_memelem_zero(s.get_mem(ins.cond)))
    then
      (p.(pc ← ins.loc, iq ← []), s, ins) // empty iq
    else
      var pc' ← if |p.iq| ≠ 0 then p.pc else p.pc + 1;
      (p.(pc ← pc', iq ← iq'), s, ins)
  else
    var pc' ← if |p.iq| ≠ 0 then p.pc else p.pc + 1;
    (p.(pc ← pc', iq ← iq'), eval_instr(ins, s), ins)
}

function steps(p: Program, s: State, k: nat)
  : (Program, State, seq<Instr>)
  decreases k {
  if k == 0 ∨ |p.iq| == 0 then (p, s, [])
  else
    var (p', s', ins) ← step(p, s);
    var (p'', s'', instrs) ← steps(p', s', k - 1);
    (p'', s'', [ins] + instrs)
}
```

(b)

Fig. 10. Formalization of program semantics foundation in Dafny. (a) Foundational definitions formalizing abstract machine. (b) Formalization of `step` and prefix execution `steps`.

```
function fetch(p: Program, s: State): (p': Program)
  ensures |p'.iq| == |p.iq| + 1
  ∧ p.iq < p'.iq       // prefix
  ∧ p'.iq[|p.iq|] == p.im[p.pc]
  ∧ p'.pc == p.pc + 1
{
  var instr ← p.im[p.pc];
  var iq' ← p.iq + [instr];
  Program_t(p.im, p.pc + 1, iq')
}

lemma fetch_is_transformer()
  ensures valid_transformer(fetch)
{
  ∀ p: Program, s: State
    ensures prog_equiv(p, s, fetch(p, s), s)
  {
    var pf ← fetch(p, s);
    var k ← |p.iq|;
    assert prog_equiv(p, s, pf, s) by {
      var (p', s', instrs)    ← steps(p, s, k);
      var (pf', sf', instrs') ← steps(pf, s, k);
      assert instrs == instrs' ∧ s' == sf' by {
        lemma_common_prefix(p, pf, k, s);
      }
      assert |p'.iq| == 0;
      assert |pf'.iq| == 0 ∨ |pf'.iq| == 1;
      if |pf'.iq| == 0 {         // a branch taken
        assert p'.pc == pf'.pc;
      } else {
        assert p'.pc == p.pc ∧ pf'.pc == pf.pc by {
          lemma_common_prefix(p, pf, k, s);
        }
        var (p'', s'', ins) ← step(p', s');
        var (pf'', sf'', insf) ← step(pf', sf');
        assert prog_equiv(p'', s'', pf'', sf'');
      }
      lemma_common_prefix(p, pf, k, s);
    }
  }
}
```

Fig. 11. Mechanization of the proof of Theorem 2 in Dafny.

And finally, as defined in previous sections, a *program* has instruction memory, a program counter, and a sequence of instruction that serves as `iq`. Note that the predicate `well_formed` ensures that `pc` is valid in the instruction memory, and all the instruction memory only contains global instructions.

It is instructive to see how to mechanize program transformer proofs. In Fig. 11, theorem `fetch_is_transformer` proves that function `fetch` is a valid transformer (Theorem 2). A key lemma is `lemma_common_prefix` which is the Dafny formulation of Lemma 1. The proof proceeds as follows. Since fetch only appends an instruction to the IQ, p and pf share the common prefix (of size k, where k is the size of the IQ in p) in their IQs. We "flush" k instructions in their IQs with *steps*. By lemma `lemma_common_prefix`, they should produce the same state ($s' = sf'$). After that, the IQ in p' should be empty, and the IQ in pf' is either empty (due to a branch taken) or of size 1. If it is empty, its pc must be the same as the pc in p', since they must have executed the same branch instruction. Otherwise there is no taken branch, therefore the pc is unchanged. We further "step" both programs, noting that the instruction being executed must be $p.IM[p.pc]$, therefore, $\langle p'', s'' \rangle \equiv \langle pf'', sf'' \rangle$. According to the definition of `prog_equiv`, $\langle p', s' \rangle \equiv \langle pf', sf' \rangle$. Finally, using lemma `lemma_common_prefix` we conclude $\langle p, s \rangle \equiv \langle pf, s \rangle$.

X. Case Study: Correct-by-Construction Superscalar VTA Microarchitecture

In this section, we applied the program transformation foundations to develop a correct-by-construction superscalar microarchitecture implementation of the Versatile Tensor Accelerator (VTA). The focus is not to cover the formalization in full detail but to give a flavor of the microarchitecture so that the reader can appreciate the subtleties involved and the role of our new foundations to conquer them.

A. Microarchitecture Feature Addition

We will elaborate the four stages in Fig. 1 in Section III. Note that VTA accelerates a sequence of two-dimensional arbitrary-sized tensor computations. The ISA of the accelerator supports tensor operations such as GEMM, which performs matrix-matrix multiplication over an input tensor and an weight tensor.

Algo. 6: Gemm$(itensor, wtensor, otensor, m, n, k)$

```
1 $r \leftarrow \textsc{ReduceAxis}(0, k)$
2 $ot \leftarrow \textsc{Compute}((m, n), \lambda\, x, y:$
        $\textsc{Sum}(itensor[x, r] \cdot wtensor[r, y], r))$
3 $otensor \leftarrow \textsc{Copy}(ot, m, n)$
```

Algo. 7: Lgemm(i, w, o, m, n, k)

```
1 $it \leftarrow Copy(ITensorBuf[i], m, k)$
2 $wt \leftarrow Copy(WTensorBuf[w], k, n)$
3 $r \leftarrow \textsc{ReduceAxis}(0, k)$
4 $ot \leftarrow \textsc{Compute}((m, n), \lambda\, x, y: \textsc{Sum}(it[x, r] \cdot wt[r, y], r))$
5 $OTensorBuf[o] \leftarrow \textsc{Copy}(ot, m, n)$
```

Stage 0: ISA Abstraction Algorithm 6 shows the formal semantics of the GEMM instruction in the VTA ISA as it is executed by the $EXEC$ core.[4] Here itensor, wtensor, otensor are the memory addresses of the input, weight and output tensors of sizes $m \cdot k$, $k \cdot n$ and $m \cdot n$, respectively.

Stage 1: Sequential VTA with Fetch and Data Localization (SFL) This feature introduces local buffers to store instructions and data and hide memory latency. For instructions, we apply the Fetch and Data Localization transformers. For data, we introduce three local buffers ITensorBuf of size I, WTensorBuf of size W and OTensorBuf of size O to the local storage rf to store arrays of input, weight and output tensors respectively. The instruction LGEMM is similar to GEMM but performs the multiplication over an input tensor and weight tensor from the local buffers, as shown in Algorithm 7. Supporting LGEMM requires three additional instructions ILOAD, WLOAD, and STORE. The operational model of these instructions is obvious, *e.g.*, ILOAD itensor, i, m, k loads an input tensor *itensor* from the main memory into entry i of ITensorBuf.

Stage 2: Sequential VTA with Barrier Insertion (SBI) To enable downstream concurrent executions, this feature inserts barrier/release instructions: a LG_BARRIER before an LGEMM and corresponding LG_RELEASE after the last LOAD that loads the data for LGEMM. It also inserts barrier/release instructions: a GS_BARRIER before an STORE and corresponding GS_RELEASE after the LGEMM that produces the result for the *STORE*.

Stage 3: Sequential VTA with BMM Sequencing (SBS) Since only fixed-size block matrix multiplication (BMM) can be implemented in hardware, this feature introduces a BMM-Sequence module (Algorithm 8) to decompose a GEMM computation into a semantically equivalent sequence of BMM and ACC (ACCumulate) computations, based on block matrix multiplication framework. Let the sizes of the fixed-size tensors for BMM be fm, fn, and fk. The instruction BRESET sets the (s, z)-block of entry $ins.o$ in $OTensorBuf$ to 0. The instruction BMMACC performs a BMM operation on the (x, y)-block of entry $ins.i$ in ITensorBuf and (y, z)-block of entry $ins.w$ in WTensorBuf, and then adds the result to the (s, z)-block of entry $ins.o$ in OTensorBuf.

[4]We use the term $EXEC$ core to mean a processing element that performs an arithmetic-logic functionality.

Algo. 8: BMMSequence(IQ, uIQ)

```
1 $ins \leftarrow IQ.deq()$
2 for $x \leftarrow 0 \ldots ins.m/fm$ do
3     for $z \leftarrow 0 \ldots ins.n/fn$ do
4         $uIQ.enq(\textsc{Instr}($BRESET ins.o, x, z$))$
5         for $y \leftarrow 0 \ldots ins.k/fk$ do
6             $uIQ.enq(\textsc{Instr}($BMMACC, ins.i, x, y, ins.w, y, z,
                ins.o, x, z$))$
7         end
8     end
9 end
```

Algo. 9: Decode(IQ, uIQ)

```
1  $ins \leftarrow IQ.deq()$
2  for $x \leftarrow 0 \ldots ins.m/fm$ do
3      for $z \leftarrow 0 \ldots ins.n/fn$ do
4          $uIQ.enq(\textsc{Instr}(\text{BRESET}))$
5          for $y \leftarrow 0 \ldots ins.k/fk$ do
6              $uIQ.enq(\textsc{Instr}(\text{IBLOAD}, ins.i, ins.x, ins.y))$
7              $uIQ.enq(\textsc{Instr}(\text{WBLOAD}, ins.w, ins.y, ins.z))$
8              $uIQ.enq(\textsc{Instr}(\text{BGEMMACC}))$
9              $uIQ.enq(\textsc{Instr}(\text{BSTORE}, ins.o, ins.x, ins.y))$
10         end
11     end
12 end
```

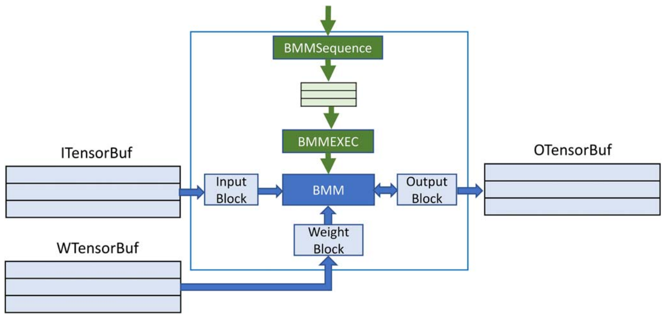


Fig. 12. BMM module refinement.

Stage 4: Concurrent VTA with Multiple Cores (CMC) Here, the EXEC_LS core is replaced with a Schedule module and three concurrent EXEC_LS cores, responsible for LOAD, STORE and GEMM operations. The Schedule module receives a stream of instructions and issues them to corresponding cores. The barrier/release instructions synchronize executions on cores. Since the cores run dedicated functions and GEMM only performs fixed-size BMM computations, they can be significantly simplified (Fig. 2 stage 1).

B. Refinement

Each feature can be further optimized for software or hardware implementation. For example, the BMM sequencing and computation in Algorithm 8 (steps 4 and 6) is refined to Algorithm 9 (steps 4 and 6-9) by introducing fixed-size blocks Input Block of size $fm \cdot fk$, Weight Block of size $fk \cdot fn$, and Output Block of size $fm \cdot fn$ between the local tensor buffers, and the BMM module (Fig. 12) and five new instructions IBLOAD, WBLOAD, BSTORE, BRESET and BGEMMACC operate on these blocks, *e.g.*, IBLOAD i, x, y loads a block (x, y) of size (fm, fk) from tensor i in *ITensorBuf* to *Input Block*.

C. Design, Implementation and Verification

Both phases of the VTA design were done using HeteroCL [41]. HeteroCL is a programming infrastructure composed of a Python-based domain-specific language (DSL) and a compilation flow. HeteroCL supports two back-ends: the generation of the LLVM code from the feature modules to be implemented as the compiler for the accelerator, and the generation of synthesizable C from the feature modules to be implemented as the hardware through HLS. Our approach enables rapid end-to-end design exploration and seamless software and hardware co-design.

Ensuring correctness of the design entails showing that the operations introduced in each step are indeed program transformers. The composition rules from Section VI then complete the proof. No additional hand-crafted invariant is necessary. We are also developing an automated sequential equivalence checking flow for discharging program equivalence proof obligations [42].

XI. Conclusion and Future Work

We have developed a foundation for microarchitecture verification based on program transformations, that permits incremental analysis of microarchitecture components. We showed how various microarchitecture features can be viewed as program transformers, and developed compositional rules to enable correct-by-construction implementations. In addition to facilitating intuitive specifications, the approach enables verification of microarchitectures with advanced features through automated tools, while obviating the need for hand-crafted invariants. We discussed how to use the approach for automated verification of a superscalar implementation of VTA. We are aware of no other frameworks for analysis of microarchitectures of such complexity without extensive manual intervention. The foundations foster correct-by-construction design by progressively optimizing and incrementally verifying individual features.

We are applying the same foundations for provably-correct design of a Fully Homomorphic Encryption accelerator. In future work, we plan to design a superscalar RISC-V processor and extend the foundations for SoC designs.

References

[1] M. Aagaard, B. Cook, N. Day, and R. B. Jones, “A framework for microprocessor correctness statements,” in *Proc. 11th Int. Conf. Correct Hardware Des. Verification Methods (CHARME)*, LNCS, T. Margaria and T. F. Melham, Eds., vol. 2144. Scotland, U.K.: Springer-Verlag, 2001, pp. 443–448.

[2] W. A. Hunt Jr. and B. Brock, “A formal HDL and its use in the FM9001 verification,” in *Mechanized Reasoning and Hardware Design*, Prentice-Hall International Series in Computer Science, C. A. R. Hoare and M. J. C. Gordon, Eds., Englewood Cliffs, NJ, USA: Prentice-Hall, 1992, pp. 35–48.

[3] J. R. Burch and D. L. Dill, “Automatic verification of pipelined microprocessor control,” in *Proc. 6th Int. Conf. Comput.-Aided Verification (CAV)*, LNCS, D. L. Dill, Ed., vol. 818. Berlin, Germany: Springer-Verlag, 1994, pp. 68–80.

[4] J. Sawada and W. A. Hunt Jr., “Verification of FM9801: An out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability,” *Formal Methods Syst. Des.*, vol. 20, no. 2, pp. 187–222, 2002.

[5] P. Manolios, “Correctness of pipelined machines,” in *Proc. 3rd Int. Conf. Formal Methods Comput.-Aided Des. (FMCAD)*, LNCS, W. A. Hunt Jr. and S. D. Johnson, Eds., vol. 1954. Austin, TX, USA: Springer-Verlag, 2000, pp. 161–178.

[6] R. Hosabettu, G. Gopalakrishnan, and M. Srivas, “Verifying advanced microarchitectures that support speculation and exceptions,” in *Proc. 12th Int. Conf. Comput.-Aided Verification (CAV)*, LNCS, E. A. Emerson and A. P. Sistla, Eds., vol. 1855. Berlin, Germany: Springer-Verlag, Jul. 2000.

[7] B. Brock and W. A. Hunt Jr., “Formal analysis of the motorola CAP DSP,” in *Industrial-Strength Formal Methods*. Berlin, Germany: Springer-Verlag, 1999.

[8] K. R. M. Leino, “Dafny: An automatic program verifier for functional correctness,” in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370.

[9] M. Srivas and M. Bickford, “Formal verification of a pipelined microprocessor,” *IEEE Softw.*, vol. 7, no. 5, pp. 52–64, Sep. 1990.

[10] A. Bronstein and T. L. Talcott, “Formal verification of pipelines based on string-functional semantics,” in *Formal VLSI Correctness Verification, VLSI Design Methods II*, L. J. M. Claesen, Ed., Amsterdam, The Netherlands: North-Holland, 1990, pp. 349–366.

[11] S. Ray and W. A. Hunt Jr., “Deductive verification of pipelined machines using first-order quantification,” in *Proc. 16th Int. Conf. Comput.-Aided Verification (CAV)*, LNCS, R. Alur and D. A. Peled, Eds., vol. 3114. Boston, MA, USA: Springer-Verlag, Jul. 2004, pp. 31–43.

[12] R. Jhala and K. McMillan, “Microarchitecture verification by compositional model checking,” in *Proc. 12th Int. Conf. Comput.-Aided Verification (CAV)*, LNCS, G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102. Berlin, Germany: Springer-Verlag, 2001.

[13] R. E. Bryant, S. German, and M. N. Velev, “Exploiting positive equality in a logic of equality with uninterpreted functions,” in *Proc. 11th Int. Conf. Comput.-Aided Verification (CAV)*, LNCS, N. Halbwachs and D. Peled, Eds., vol. 1633. Berlin, Germany: Springer-Verlag, 1999, pp. 470–482.

[14] S. K. Lahiri and R. E. Bryant, “Deductive verification of advanced out-of-order microprocessors,” in *Proc. 15th Int. Conf. Comput.-Aided Verification (CAV)*, LNCS, W. A. Hunt Jr. and F. Somenzi, Eds., vol. 2275. Berlin, Germany: Springer-Verlag, Jul. 2003, pp. 341–354.

[15] S. Goel and S. Ray, “Microarchitecture assurance and the role of theorem proving,” in *Handbook of Computer Architecture*, A. Chattopadhyay, Ed. Springer, 2022.

[16] R. Kaivola et al., “Replacing testing with formal verification in Intel CoreTM i7 processor execution engine validation,” in *Proc. Int. Conf. Comput. Aided Verification*, Berlin, Germany: Springer, 2009, pp. 414–429.

[17] A. Reid et al., “End-to-end verification of arm processors with Isa-formal,” in *Proc. Int. Conf. Comput. Aided Verification (CAV’16)*, Lecture Notes in Computer Science, S. Chaudhuri and A. Farzan, Eds., vol. 9780, no. 9780. Berlin, Germany: Springer Verlag, Jul. 2016, pp. 42–58.

[18] D. Gao and T. Melham, “End-to-end formal verification of a RISC-V processor extended with capability pointers,” in *Proc. Formal Methods Comput. Aided Des. (FMCAD)*, Piscataway, NJ, USA: IEEE, 2021, pp. 24–33.

[19] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, “Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SoC) verification,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 1, pp 1–24, Dec. 2018, doi: 10.1145/3282444.

[20] Arvind and X. Shen, “Using term rewriting systems to design and verify processors,” *IEEE Micro*, vol. 19, no. 3, pp. 36–46, May/Jun. 1999.

[21] S. Ray, K. Hao, F. Xie, and J. Yang, “Formal verification for high-assurance behavioral synthesis,” in *Proc. 7th Int. Symp. Automated Technol. Verification Anal. (ATVA)*, LNCS, Z. Liu and A. P. Ravn, Eds., vol. 5799. Macao SAR, China: Springer, Oct. 2009, pp. 337–351.

[22] K. Hao, S. Ray, F. Xie, and J. Yang, “Optimizing equivalence checking for behavioral synthesis,” in *Proc. Des. Automat. Test Europe (DATE)*, 2012, pp. 1500–1505.

[23] Z. Yang, K. Hao, S. Ray, and F. Xie, “Equivalence checking for compiler transformations in behavioral synthesis,” in *Proc. 31st IEEE Int. Conf. Comput. Des. (ICCD)*, G. Byrd, K. Schneider, N. Chang, and S. Ozev, Eds. 2013, pp. 491–494.

[24] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen, "Scalable sequential equivalence checking across arbitrary design transformations," in *Proc. Int. Conf. Comput. Des.*, 2006, pp. 259–266.

[25] A. J. Hu, "High-level vs. RTL combinational equivalence: An introduction," in *Proc. Int. Conf. Comput. Des.*, 2006, pp. 274–279.

[26] X. Leroy, "Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant," in *Proc. 33rd Symp. Princ. Program. Lang. (POPL)*, New York, NY, USA: ACM, 2006, pp. 42–54.

[27] D. Kästner et al., "CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler," in *Proc. Embedded Real Time Softw. Syst. (ERTS)*, SEE, Jan. 2018. [Online]. Available: http://xavierleroy.org/publi/erts2018_compcert.pdf

[28] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the LLVM intermediate representation for verified program transformations," *SIGPLAN Notices*, vol. 47, pp. 427–440, Jan. 2012.

[29] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Proc. TACAS*, 1998, pp. 151–166.

[30] G. C. Necula, "Translation validation for an optimizing compiler," in *Proc. ACM SIGPLAN PLDI*, 2000, pp. 83–94.

[31] "Stratus high-level synthesis." Cadence. Accessed: 2023. [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html

[32] "Catapult high-level synthesis." Mentor Graphics. Accessed: 2023. [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls

[33] R. Venkatesan et al., "A 0.11 pJ/Op, 0.32-128 TOPS, scalable multi-chip-module-based deep neural network Accelerator designed with a high-productivity VLSI methodology," in *Proc. Hot Chips, Symp. High Perform. Chips*, Cupertino, CA, USA: Springer-Verlag, 2019, pp. 1–24.

[34] W. Chappell, "The intertwined history of DARPA and Moore's law, DARPA 2018," *Defense Media Network*, 2018. [Online]. Available: https://www.defensemedianetwork.com/stories/the-intertwined-history-of-darpa-and-moores-law/

[35] "Sandy bridge bug 2x costly as pentium math bug," *Tom's Hardware*, 2011. [Online]. Available: https://www.tomshardware.com/news/sandy-bridge-sata-error-pentium-fdiv-bug,12115.html

[36] "Intel's sapphire rapids had 500 bugs, launch window moves further," *Tom's Hardware*, 2022. [Online]. Available: https://www.tomshardware.com/news/intel-sapphire-rapids-had-500-bugs-launch-window-moves-further

[37] J. McCarthy, "A basis for a mathematical theory of computation," in *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, Eds. Amsterdam, The Netherlands: North Holland, 1963.

[38] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theor. Computer Sci.*, vol. 82, no. 2, pp. 253–284, May 1991.

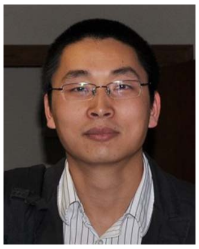
[39] L. Lamport, "Specifying concurrent program modules," *ACM Trans. Program. Lang. Syst. (ACM TOPLAS)*, vol. 5, no. 2, pp. 190–222, Apr. 1983.

[40] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2011.

[41] Y.-H. Lai et al., "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, FPGA, New York, NY, USA: ACM, 2019, pp. 242–251.

[42] Y. Wang and F. Xie, "Extending tensor virtual machine to support deep-learning accelerators with convolution cores," in *Proc. 26th Int. Conf. Eng. Complex Comput. Syst. (ICECCS)*, 2022, pp. 189–194.

Jin Yang received the Ph.D. degree in CS from the University of Texas at Austin. He is a Senior Principal Research Scientist with Intel Labs, responsible for applied formal methods research in the specification, design, and verification of system software, hardware, and security. He joined Intel, in 1997, and has led many high-impact research projects and corporate-wide initiatives at Intel. He is active in the CAD and formal methods research communities with over 50 publications.



Zhenkun Yang received the Ph.D. degree in computer science from Portland State University. He is a Research Scientist with Intel Labs from 2015. His research mainly focuses on formal verification of software, firmware and hardware systems, with techniques include theorem proving, model checking, symbolic execution, and fuzzing.

Jeremy Casas received the M.S. degree in CS from the Oregon Graduate Institute. He is a Research Scientist with Intel Labs, currently focused on R&D of new design and verification paradigms and development environments for modern computer designs. Prior to joining Intel Labs, he was a Post-Silicon Validation Architect with Intel's client product development group. He has extensive experience on development and application of dynamic and formal verification systems, micro-architecture design and verification, and post-Silicon validation/debug tools and methodologies. He has publications in conferences/journals and several patents.

Sandip Ray (Senior Member, IEEE) received the Ph.D. degree from the University of Texas at Austin. He is a Professor at the Warren B. Nelms Institute for the Connected World affiliated with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA. Prior to that, he worked with NXP Semiconductors and Intel Strategic CAD Laboratories where, he led industrial research and R&D projects in pre-silicon and post-silicon validation of security and functional correctness of SoC designs, design-for-security and design-for-debug architectures, and security validation for automotive and the Internet-of-Things applications. His current research targets correct, dependable, secure, and trustworthy computing. He is the Author of three books and over 100 publications in international journals and conferences. He has also served as a TPC Member of over 50 international conferences and as Guest Editor for several journals.