Trace Semantics and Algebraic Laws for MCA ARMv8 Architecture Based on UTP

Lili Xiao and Huibiao Zhu(✉)

East China Normal University, Shanghai, China
hbzhu@sei.ecnu.edu.cn

Abstract. Hardware architectures like x86 and ARM provide relaxed memory models for efficiency reasons. The revised ARMv8 architecture is multi-copy atomic (MCA), which brings relaxed-memory effects through thread-local out-of-order, speculative execution and thread-local buffering. In this paper, we investigate the trace semantics for the MCA ARMv8 architecture, acting in the denotational semantics style based on Unifying Theories of Programming (UTP). In order to present all the valid execution results including reorderings of any program under ARMv8, a trace expressed as a sequence of snapshots is introduced, and it relies heavily on various dependencies. The snapshots record the change of variables of different types of actions. We also study the algebraic laws for MCA ARMv8, including a set of sequential and parallel expansion laws. The concept of head normal form is explored for each program, and every program is described in the form of guarded choice which can model the execution of a program with reorderings. Therefore, the linearizability for ARMv8 is supported.

Keywords: Relaxed memory model · MCA ARMv8 architecture · Unifying Theories of Programming (UTP) · Trace semantics · Algebraic laws

1 Introduction

ARMv7 and early ARMv8 architectures defined a relaxed memory model used to improve the performance of concurrent programs. This model is non multi-copy atomic (non MCA). However, the complexity of implementation, verification and reasoning produced by allowing non MCA behaviors does not bring in sufficient performance benefits [1]. Then the revised ARMv8 architecture is shift to the model under multi-copy atomic (MCA) semantics [2], which illustrates that when a write is visible to some other thread, it becomes visible to all other threads. Therefore, it simplifies the allowed behaviors of every program.

The MCA ARMv8 architecture maintains the buffer of each thread, throwing away the redundant buffers in [3], shown in Fig. 1. Always, a memory write is split into two steps, committing the write to buffer and propagating it to memory later. A read from location x demands to first check the private buffer to

S. Qin et al. (Eds.): SETTA 2021, LNCS 13071, pp. 81–101, 2021.
https://doi.org/10.1007/978-3-030-91265-9_5

see whether it contains such a write to the same location. If yes, the read operation terminates. Otherwise, the shared memory will be explored. TSO [4] and ARMv8 are both MCA models [5], and TSO only omits store-load constraint. However, ARMv8 releases store-store, store-load, load-store and load-load constraints, if a variety of dependencies (explained in the following section) do not exist. In addition, ARMv8 supports speculative execution, which describes that the instructions in a branch may execute before the evaluation of the branching condition has completed. The cfence instruction is used to prohibit it.

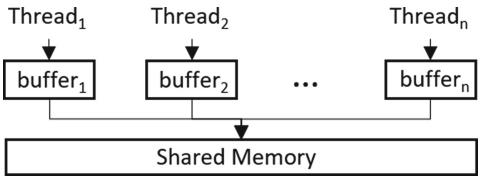

Fig. 1. The MCA ARMv8 architecture.

To demonstrate how ARMv8 exhibits reorderings, consider the parallel program $(x := 1; y := 1)||(a := y; b := x)$. Since the statements $x := 1$ and $y := 1$ do not depend on each other, $x := 1$ and $y := 1$ can be reordered. If $y := 1$ is scheduled firstly and then the reads from y and x happen, the variables a and b can obtain 1 and 0 in the same execution.

Unifying Theories of Programming (UTP) [6] was developed by Hoare and He in 1998. It aims at proposing a convincing unified framework to combine and link operational semantics [7], denotational semantics [8] and algebraic semantics [9]. In this paper, we consider the denotational semantics of the MCA ARMv8 architecture, where our approach is based on UTP and the trace structure is applied. In our semantic model, a trace is in the form of the sequence of snapshots, and the snapshots record the changes on registers, buffers and memory contributed by different types of actions. With the dependencies among those actions, all the valid execution traces can be achieved. We also explore the algebraic laws for MCA ARMv8, including a set of sequential and parallel expansion laws. On the basis of the laws, we can see that every program can be converted into a guarded choice.

The operational and axiomatic models of MCA ARMv8 are introduced in [1,10], while our investigation for it can not only support the linearizability [11,12] of this architecture, but also support to deduce some interesting algebraic properties of programs.

The remainder of this paper is organized as follows. We investigate the trace semantics of the MCA ARMv8 architecture in Sect. 2. Section 3 presents a set of algebraic laws including sequential and parallel expansion laws. Section 4 concludes the paper and discusses the future work. We leave some technical definitions and analyses in the appendix.

2 Trace Semantics

2.1 The Syntax of ARMv8

In this section, we give the description of the programs under ARMv8 with a simple imperative language, which is adapted and extended from [13]. In the following syntax, e ranges over arithmetic expressions on real numbers, h over Boolean expressions and p over programs. Particularly, a Fence instruction is used to guarantee the absolute order of the memory accesses separated by it, while speculative execution can be prevented by the control fence (cfence) instructions. The program illustrated in the previous section is one quick example.

$$
\begin{aligned}
&v ::= ..., -2, -1, 0, 1, 2, ... \\
&e ::= v \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid ... \\
&h ::= true \mid false \mid e_1 = e_2 \mid \neg h \mid h_1 \vee h_2 \mid h_1 \wedge h_2 \mid ... \\
&p ::= x := e \mid \text{Fence} \mid \text{cfence} \mid p_1; p_2 \mid \text{if } h \text{ then } p_1 \text{ else } p_2 \mid \text{while } h \text{ do } p \mid p_1 || p_2
\end{aligned}
$$

2.2 The Semantic Model

This section investigates the denotational semantic model for the MCA ARMv8 architecture, with the application of the trace structure. We illustrate the behaviors of a process by a trace of snapshots, which records the sequence of actions.

A snapshot in a trace can be expressed as a triple $(cont, oflag, eflag)$, where:

1. Generally, *cont* is composed of two elements *var* and *val*, denoting the data state of one variable at a given moment. However, it can also be illustrated as a branching condition h or Fence or cfence.
2. $oflag$ works on distinguishing different types of operations, and Table 1 gives a brief description of it.
   (a) If *cont* is in the form of (var, val), $oflag$ can be divided into three categories. When *var* is a global variable, committing to the buffer leads to that $oflag$ is 1, and propagating to the whole memory results in that $oflag$ is 2. When writing to a local variable, $oflag$ is set to be 3.
   (b) Otherwise, the corresponding $oflag$ to a branching condition h or Fence or cfence is 0 or -1 or -2.

Table 1. Different types of operations divided by the parameter $oflag$.

$oflag$	Values					
	1	2	3	0	-1	-2
Types	Committing	Propagating	Register write	Branching condition	Fence	cfence

3. For a process, in order to include its environment's behaviors, we introduce the parameter $eflag$. Once the process does the action, $eflag$ is set to be 1. If the operation is performed by its environment, $eflag$ is equal to 0.

The projection function $\pi_i (i \in \{1,2,3\})$ is defined to get the i-th element of a snapshot, e.g., $\pi_3(cont, oflag, eflag) = eflag$. Then, if $cont$ is in the form of (var, val), we use the function $\pi_i (i \in \{1,2\})$ to obtain the relevant variable and value, i.e., $\pi_1(\pi_1(cont, oflag, eflag)) = var$, $\pi_2(\pi_1(cont, oflag, eflag)) = val$.

We use the notation $traces(P)$ to stand for all the valid execution results. Two simple examples are shown below to provide an intuitive illustration of it.

Example 1.1. Consider the program $a := 1; b := 1$, where a and b are both local. Because $a := 1$ and $b := 1$ do not have dependency, either $a := 1$ or $b := 1$ can be chosen to execute first. Then, two traces can be generated.

$$traces(a := 1; b := 1) = \left\{ \boxed{\langle((a,1),3,1),\ ((b,1),3,1)\rangle}, \boxed{\langle((b,1),3,1),\ ((a,1),3,1)\rangle} \right\}$$

Example 1.2. Given a program $P||Q$, where $P =_{df}\ a := 1$, $Q =_{df}\ b := 1$, and a and b are local, $\langle \mathbf{((a,1),3,1)},\ ((b,1),3,0)\rangle$ is one of $traces(P)$. Since the former and latter are contributed by P and P's environment (i.e., Q), the third elements are 1 and 0 respectively. Meanwhile, $\langle((a,1),3,0),\ \mathbf{((b,1),3,1)}\rangle$ is one of $traces(Q)$. Hence, $P||Q$ can produce one trace $\langle \mathbf{((a,1),3,1)},\ \mathbf{((b,1),3,1)}\rangle$, reflected in the trace semantics of parallel construct. □

2.3 Trace Semantics

In the following, we present the trace semantics $traces(P)$ for each program P under the MCA ARMv8 architecture.

Local Assignment. Local variables are written to the private registers in every thread directly. Here, it is denoted by the second parameter 3 in the snapshot.

$$traces(a := e) =_{df} \{s \,^\wedge\, \langle((a, r(e)), 3, 1)\rangle\} \text{ where, } \pi_3^*(s) \in 0^*$$

Here, the expression $\pi_3^*(s) \in 0^*$ informs that $eflag$ in every snapshot of the sequence s is 0, i.e., s is contributed by the environment. On the basis of the introduction to the projection function π_3, the notation $\pi_3^*(s)$ denotes the repeated execution of the function π_3 on each snapshot in the trace s. Then, with the application of this approach, a process can include its environment's behaviors. The notation $=_{df}$ refers to definitions, whereas $s^\wedge t$ stands for the concatenation of traces s and t. Further, $s^\wedge T =_{df} \{s^\wedge t \mid t \in T\}$ and $S^\wedge T =_{df} \{s^\wedge t \mid s \in S \wedge t \in T\}$.

In addition, we introduce a read function named r to get the concrete value of a variable, and the detailed definition of it is given in Appendix A (page 20). Note that $r(e)$ requires us to execute the read function of every variable which appears in the expression e. For instance, $r(x + y)$ is expressed as $r(x) + r(y)$. After getting the values of those variables, the value of the expression can be calculated.

The definitions for $traces(Fence)$ and $traces(cfence)$ are similar.

$$traces(Fence) =_{df} \{s \,^\wedge\, \langle(Fence, -1, 1)\rangle\} \text{ where, } \pi_3^*(s) \in 0^*$$
$$traces(cfence) =_{df} \{s \,^\wedge\, \langle(cfence, -2, 1)\rangle\} \text{ where, } \pi_3^*(s) \in 0^*$$

Global Assignment. We split the global assignment into two steps: (1) committing the write to the store buffer; (2) propagating it to the shared memory. The two steps cannot be swapped.

$$traces(x := e) =_{df} \{u^\frown\langle((x, r(e)), 1, 1)\rangle^\frown v^\frown\langle((x, r(e)), 2, 1)\rangle\}$$
$$\text{where, } \pi_3^*(u) \in 0^* \text{ and } \pi_3^*(v) \in 0^*$$

Similar to the explanation of local assignment, the environment can perform any number of operations before each step of global assignment. Thus, two subtraces u and v are inserted, which are contributed by the environment. In the above trace, $(x, r(e))$ denotes that the value of x is changed to $r(e)$. The second parameter being 1 or 2 says that the effect of x's change has been brought to buffer or memory. The assignment is done by the thread itself, i.e., $eflag$ is 1.

Conditional and Iteration

Example 2. Consider the execution of conditional in P_1, where the variables x, y and z are global, and a and b are local.

if $(x == 1)$ {	if $(x == 1)$ {	if $(x == 1)$ {
$a := y$;	$y := 1$;	cfence;
} else {	}	$a := y$;
$b := z$;		}
}		
(P_1)	(P_2)	(P_3)

Now, we introduce the speculative execution [14] in conditional. Speculative execution is that the instructions in a branch can be executed before the branching condition is evaluated to increase performance. Because the speculative execution is allowed by the specification of the MCA ARMv8 architecture, the branching condition $x == 1$, $a := y$ in one branch and $b := z$ in another have the same possibility to be performed firstly. The middle layer in Fig. 2 depicts these three situations, and each framed part is done first.

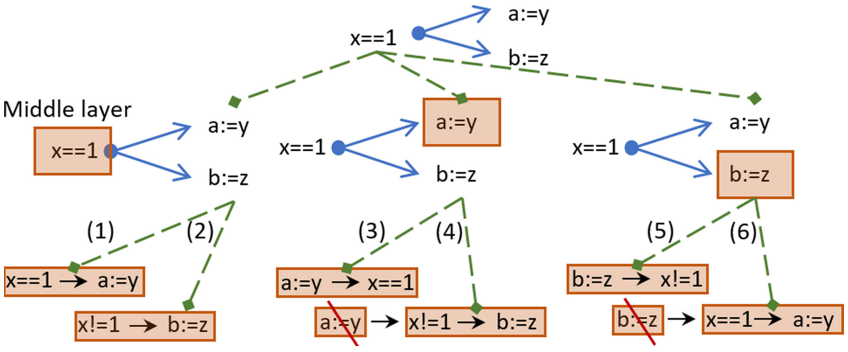

Fig. 2. The illustration of if structure.

– When the evaluation $x == 1$ is scheduled, the conditional will behave the same as $a := y$ if the judgment is true, otherwise behave as $b := z$, shown as the situations (1) and (2) in Fig. 2. The traces $\langle (x == 1, 0, 1),\ ((a, r(y)), 3, 1)\rangle$ and $\langle (x! = 1, 0, 1),\ ((b, r(z)), 3, 1)\rangle$ are related to these two situations.
– The conditional executes the load $a := y$ first, and then evaluates the branching condition $x == 1$. If true, the process terminates successfully and produces the trace $\langle ((a, r(x)), 3, 1),\ (x == 1, 0, 1)\rangle$. Otherwise, the result caused by $a := y$ is discarded. The conditional continues to carry out the instruction $b := z$, and then generates the trace $\langle (x! = 1, 0, 1),\ ((b, r(z)), 3, 1)\rangle$. They are described by the situations (3) and (4) in Fig. 2. The analysis of executing $b := z$ first is similar and presented in cases (5) and (6). □

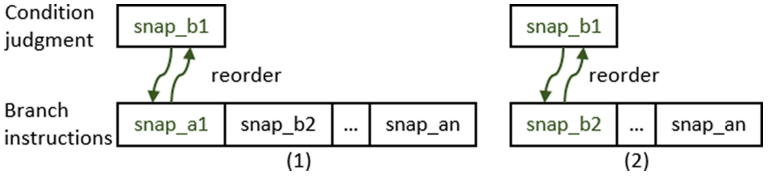

Fig. 3. The dependency in if structure.

Now, we study the trace semantics of conditional. Firstly, to judge whether a common statement can be speculatively executed, shown in Fig. 3(1), we introduce the function $NoDepd_1(snap\_b, snap\_a)$. It defines the requirements that $snap\_b$ and $snap\_a$ should achieve if there is no dependency between them:

(1) The assigned variable in $snap\_a$ is not global, because a thread cannot discard the result once it makes some changes in any location in the memory.
(2) $dom(\pi_1(snap\_b))$ records the set of all the variables in the branching condition. The written variable in $snap\_a$ cannot appear in the mentioned set.
(3) The variables read by $snap\_a$ and those read by $snap\_b$ do not contain the same global variables. The former ones are denoted by $dom(\pi_2(\pi_1(snap\_a)))$ and the latter ones are represented as $dom(\pi_1(snap\_b))$.

Here, $snap\_a$ is one snapshot of an assignment. The snapshot of a condition judgment h is denoted by $snap\_b$, which is in the form of $(h, 0, 1)$.

$NoDepd_1(snap\_b, snap\_a)$ can be formalized as below. Here, we use $Globals$ to denote the set of all the global variables, and $dom(\_)$ stands for the variables appearing in the argument. Note that the three formulas below correspond to the three items above.

$$NoDepd_1(snap\_b, snap\_a)$$
$$=_{df} \begin{pmatrix} (\pi_1(\pi_1(snap\_a)) \notin Globals) \ \wedge & ...(2.3.1) \\ (\pi_1(\pi_1(snap\_a)) \notin dom(\pi_1(snap\_b))) \ \wedge & ...(2.3.2) \\ \big(\big(dom(\pi_2(\pi_1(snap\_a))) \cap dom(\pi_1(snap\_b))\big)\big) \cap Globals = \varnothing\big) & ...(2.3.3) \end{pmatrix}$$

Secondly, for nested conditional, in order to investigate whether two branching conditions can be reordered, which is illustrated in Fig. 3(2), we give the

definition of the function $Nodepd_2(snap\_b_1, snap\_b_2)$. If two branching conditions do not depend on each other, the following condition that $snap\_b_1$ and $snap\_b_2$ may not refer to the same global variables should be satisfied, which is defined as $Nodepd_2(snap\_b_1, snap\_b_2)$.

$$Nodepd_2(snap\_b_1, snap\_b_2) =_{df} (dom(\pi_1(snap\_b_1)) \cap dom(\pi_1(snap\_b_2))) \cap Globals = \varnothing$$

For a condition judgment h, $traces(h) =_{df} \{s^\frown\langle snap\_b\rangle\}$, where $\pi_3^*(s) \in 0^*$, and $snap\_b = (h, 0, 1)$. It means that the environment is allowed to do any number of operations before h, denoted by the sequence s.

Then, given a snapshot $snap\_b$ of branching condition h and a trace t of all the instructions in a branch, we interleave $s^\frown\langle snap\_b\rangle$ and t which is formalized as $addCond(s^\frown\langle snap\_b\rangle, t)$ to produce all the possible execution results.

$$\begin{aligned}
&addCond(s^\frown\langle snap_b\rangle, t)\\
=_{df}\ &hd(s^\frown\langle snap_b\rangle)^\frown addCond(tl(s^\frown\langle snap_b\rangle), t)\\
&\cup \left(\begin{array}{l}(hd(t)^\frown addCond(s^\frown\langle snap_b\rangle, tl(t)))\\ \quad \lhd \left(\begin{array}{c}\pi_3(hd(t)) = 0\\ \vee(\pi_2(hd(t)) \in \{1,2,3\} \wedge NoDepd_1(snap_b, hd(t)))\quad ...(2.3.4)\\ \vee\ (\pi_2(hd(t)) = 0 \wedge NoDepd_2(snap_b, hd(t)))\quad ...(2.3.5)\end{array}\right) \rhd\\ \phi\end{array}\right)
\end{aligned}$$

where, $addCond(\langle\rangle, \langle\rangle) = \{\langle\rangle\}$

$addCond(s^\frown\langle snap\_b\rangle, \langle\rangle) = \{s^\frown\langle snap\_b\rangle\}, addCond(\langle\rangle, t) = \{t\}$,

During the process of interleaving, we skip all the environment behaviors included in s and t. When meeting a snapshot in t which has dependency with $snap\_b$ (i.e., none of $NoDepd_1$ or $NoDepd_2$ can be satisfied shown as the formulas (2.3.4) and (2.3.5)), only the element in $s^\frown\langle snap\_b\rangle$ can be scheduled. The calculation of t will be explained in the later paragraph.

The notation $hd(s)$ is used to denote the first snapshot of the trace s and $tail(s)$ stands for the result of removing the first snapshot in the trace s.

Therefore, we give the definition of conditional by applying $addCond$.

$$traces(\text{if } h \text{ then } P \text{ else } Q) =_{df} \bigcup_{c_1} addCond(s_1, t_1) \lhd\ h\ \rhd \bigcup_{c_2} addCond(s_2, t_2)$$

where, $c_1 = s_1 \in traces(h) \wedge t_1 \in traces(P),\ c_2 = s_2 \in traces(\neg h) \wedge t_2 \in traces(Q)$

Example 2: Continuation. Now, we give different scenarios to help understand conditional better.

Case 1: As analyzed in Fig. 2, the traces of P_1 are produced as below.

$$traces(P_1) = \left\{\begin{array}{l}\langle(x == 1, 0, 1),\ ((a, r(y)), 3, 1)\rangle,\ \langle((a, r(x)), 3, 1),\ (x == 1, 0, 1)\rangle,\\ \langle(x! = 1, 0, 1),\ ((b, r(z)), 3, 1)\rangle,\ \langle((b, r(z)), 3, 1),\ (x! = 1, 0, 1)\rangle\end{array}\right\}$$

Case 2: Assume x and y in P_2 are global variables. Then the instruction $y := 1$ cannot be executed before the branching condition $x == 1$.

$$traces(P_2) = \{\ \langle(x == 1, 0, 1),\ ((y, 1), 1, 1), ((y, 1), 2, 1)\rangle,\ \langle(x! = 1, 0, 1)\rangle\ \}$$

Case 3: Consider the program P_3. Although a is a local variable, the load $a := y$ cannot be performed before $x == 1$ since the special instruction cfence exists.

$$traces(P_3) = \{\, \langle (x == 1, 0, 1),\ (\text{cfence}, -2, 1),\ ((a, r(y)), 3, 1)\rangle,\ \langle (x! = 1, 0, 1)\rangle \,\} \quad \square$$

The trace semantics of *Iteration* is discussed based on that of *Conditional* and least fixed point concept [15,16]. For while h do P, we consider it as if h then $(P;$while h do $P)$ else Π. Then, the trace semantics of it can be achieved.

$$\begin{aligned}
&traces(\text{while } h \text{ do } P) =_{df} \bigcup_{n=0}^{\infty} traces\{F^n(\text{STOP})\},\\
&\text{where, } F(X) =_{df} \text{if } h \text{ then } (P; X) \text{ else } \Pi,\\
&\qquad F^0(X) =_{df} X,\\
&\qquad F^{n+1}(X) =_{df} F(F^n(X))\\
&\qquad\qquad = \underbrace{F(...(F}_{n \text{ times}}(F(X)))...)\\
&\qquad traces(\Pi) =_{df} \{\varepsilon\} \text{ and } traces(\text{STOP}) =_{df} \{\}
\end{aligned}$$

Sequential Composition. To facilitate making sequential composition between two traces s and t, we continue to introduce two more functions firstly.

If $x := e$ and $y := f$, which are represented by two snapshots $snap\_a_1$ and $snap\_a_2$ under the formal model, do not have dependency, four constraints should hold [17]. Here, x and y may be global or local, and e and f are expressions.

(1) The variables assigned in $snap\_a_1$ and $snap\_a_2$ are distinct, and they can be extracted from these snapshots through $\pi_1(\pi_1(snap\_a_1))$ and $\pi_1(\pi_1(snap\_a_2))$.
(2) y should not be referred to in e. In other words, the assigned variable in $snap\_a_2$ cannot be free in the variables read by $snap\_a_1$ represented as $dom(\pi_2(\pi_1(snap\_a_1)))$.
(3) The variables read by $snap\_a_2$ which we use $dom(\pi_2(\pi_1(snap\_a_2)))$ to denote should not contain the assigned variable in $snap\_a_1$.
(4) The variables read by $snap\_a_1$ and those by $snap\_a_2$ can have the same variables, but those variables must be local.

And we use the four lines below (i.e., (2.4.1), (2.4.2), (2.4.3) and (2.4.4)) in the function $NoDepd_3(snap\_a_1, snap\_a_2)$ to outline the mentioned four conditions.

$$\begin{aligned}
&NoDepd_3(snap_a_1, snap_a_2)\\
=_{df} &\left(\begin{array}{l}
\left(\begin{array}{ll}
(\pi_1(\pi_1(snap_a_1)) \neq \pi_1(\pi_1(snap_a_2))) \ \wedge & ...(2.4.1)\\
(\pi_1(\pi_1(snap_a_2)) \notin dom(\pi_2(\pi_1(snap_a_1)))) \ \wedge & ...(2.4.2)\\
(\pi_1(\pi_1(snap_a_1)) \notin dom(\pi_2(\pi_1(snap_a_2)))) \ \wedge & ...(2.4.3)\\
((dom(\pi_2(\pi_1(snap_a_1))) \cap dom(\pi_2(\pi_1(snap_a_2)))) \cap Globals = \varnothing) & ...(2.4.4)
\end{array}\right)\\
\vee \left(\begin{array}{l}
(\pi_2(snap_a_1) = 2 \wedge \pi_2(snap_a_2)! = 2) \vee\\
(\pi_2(snap_a_1) = \pi_2(snap_a_2) = 2 \wedge \pi_1(\pi_1(snap_a_1)) \neq \pi_1(\pi_1(snap_a_2)))
\end{array}\right)
\end{array}\right)
\end{aligned}$$

In particular, the term *forwarding*, which has the equivalent effect with *bypassing* [18] under TSO memory model, is illustrated by the last two lines in the formula above. It says that the operation propagating to the shared memory does not depend on the load action later. However, if the load is also related to a write to one location, two propagation actions should follow the principle named modify order of the same location.

Example 3. Consider the sequential program $x := 1; a := x$, where a is local and x is global. As explained above, the sub-traces $\langle((x,1),2,1),\ ((a,r(x)),3,1)\rangle$ and $\langle((a,r(x)),3,1),\ ((x,1),2,1)\rangle$ are both valid.

$$traces(x := 1; a := x) = \left\{ \begin{array}{l} \langle((x,1),1,1),\ ((x,1),2,1),\ ((a,r(x)),3,1)\rangle, \\ \langle((x,1),1,1),\ ((a,r(x)),3,1),\ ((x,1),2,1)\rangle \end{array} \right\}$$

Here, the environment operations are not exhibited. We also ignore how to make composition of these snapshots, and the technique of it is given later. □

There is an assignment $x := e$ and a branching condition h, and they conform to program order. *snap_a* is one snapshot of $x := e$, while *snap_b* is the snapshot of h. If the snapshots can be reordered, two requirements should be met, defined by $NoDepd_4(snap\_a, snap\_b)$. One is that both of them cannot load the same global variables, modeled as the former conjunct in the formula (2.4.5). Informally, the other requirement is that x does not appear free in h. Hence, the variables which *snap_b* reads do not contain the variable which *snap_a* writes.

Specially, if *snap_a* is the snapshot of propagation, it and *snap_b* do not have dependency without any constraint according to *forwarding*, denoted by the last line in the formula.

$$\begin{array}{l} NoDepd_4(snap\_a, snap\_b) \\ =_{df} \left(\begin{array}{l} \left(\begin{array}{l} \big(dom(\pi_2(\pi_1(snap\_a))) \cap dom(\pi_1(snap\_b)) \big) \cap Globals = \varnothing \\ \wedge\ (\pi_1(\pi_1(snap\_a)) \notin dom(\pi_1(snap\_b))) \end{array} \right) \quad ...(2.4.5) \\ \vee\ \pi_2(snap\_a) = 2 \end{array} \right) \end{array}$$

Then, we give a detailed introduction to the function $seqcom(s,t)$ whose target is to interleave two traces s and t. The result of interleaving two empty traces is still empty. If one of them is empty and the other is nonempty, the result follows the nonempty one.

$$\begin{array}{l} seqcom(s,t) \\ =_{df} \left(\begin{array}{l} hd(s)^\frown seqcom(tl(s),t) \\ \cup \left(\begin{array}{l} (hd(t)^\frown seqcom(s,tl(t))) \\ \qquad \triangleleft\ \pi_3(hd(t)) = 0 \vee \bigvee\limits_{i \in \{1,2,3,4,5\}} case_i(s,t) \triangleright \\ \phi \end{array} \right) \end{array} \right) \end{array}$$

where, $seqcom(s, \langle\rangle) = \{s\},\ seqcom(\langle\rangle, t) = \{t\},\ seqcom(\langle\rangle, \langle\rangle) = \{\langle\rangle\}$

The first snapshot in s can always be scheduled. However, if the first in the next trace t wants to be triggered, it should satisfy the conditions that it is

contributed by the environment, or it is done by the thread itself but meets one of the following five requirements. The requirements are expressed by $case_i$ where $i \in \{1, 2, 3, 4, 5\}$. Table 2 gives a brief introduction to $case_i$. It is worth noting that, the mentioned conditions lead to the difference between this interleaving introduced here and traditional interleaving [16].

Table 2. The description of $Case_i$.

Cases	Description
$case_1(s,t)$	If the first in the latter trace t is the snapshot of a Fence instruction, how to make it be the head of the interleaving of s and t
$case_2(s,t)$	The snapshot of a cfence instruction is at the head of t
$case_3(s,t)$	One snapshot of a global assignment takes the lead in t
$case_4(s,t)$	A local assignment's snapshot comes first in the trace t
$case_5(s,t)$	The branching condition is scheduled first in t

Now, we give the detailed formalization and illustration of those cases as below. $case_1$ is that the first in t is the snapshot of a Fence instruction, and it wants to become the head of the interleaving of s and t. Then all the snapshots in s, which are not done by the environment (The same applies to the following cases), should only be related with local assignments. And those assignments cannot read any global variables. The reason for these constraints is that for a Fence instruction, all the po-previous memory access instructions, conditional branch instructions and barriers are finished.

$$case_1(s,t) =_{df} \left(\begin{array}{c} \pi_1(hd(t)) = \text{Fence} \wedge \pi_3(hd(t)) = 1 \\ \wedge\ \forall a' \in s \bullet \left(\pi_3(a') = 1 \rightarrow \left(\begin{array}{c} \pi_2(a') = 3 \\ \wedge\ \forall x \in dom(\pi_2(\pi_1(a'))) \bullet x \notin Globals \end{array} \right) \right) \end{array} \right)$$

The snapshot of a cfence instruction at the beginning of the next trace t would like to be scheduled first. It requires that any snapshot related to a barrier or a branching condition, does not occur in the trace s, which is formalized as $case_2$.

$$case_2(s,t) =_{df} \left(\begin{array}{c} \pi_1(hd(t)) = \text{cfence} \wedge \pi_3(hd(t)) = 1 \\ \wedge\ \forall a' \in s \bullet \left(\pi_3(a') = 1 \rightarrow \left(\begin{array}{c} \pi_2(a')! = 0 \\ \wedge \pi_2(a')! = -1 \\ \wedge \pi_2(a')! = -2 \end{array} \right) \right) \end{array} \right)$$

Provided that the first snapshot $hd(t)$ in t is resulted from committing or propagating a memory write, it is impossible for the trace s to include the snapshots of the Fence and cfence instructions, and branching conditions (Taking no account of any environment operation). In other words, s is the sequence of the snapshots of global and local assignments contributed by the thread itself,

as well as some environment actions. Therefore, for each snapshot a' in s, once $eflag$ is 1, $NoDepd_3$ holds between the snapshots a' and $hd(t)$. This case is modeled as below.

$$case_3(s,t) =_{df} \left(\begin{array}{c} \pi_1(\pi_1(hd(t))) \in Globals \wedge \pi_3(hd(t)) = 1 \\ \wedge\ \forall a' \in s \bullet \left(\pi_3(a') = 1 \rightarrow \left(\begin{array}{c} \pi_2(a')! = -1 \wedge \pi_2(a')! = -2 \\ \wedge\ \pi_2(a')! = 0 \wedge NoDepd_3(a', hd(t)) \end{array} \right) \right) \end{array} \right)$$

If the head in t, which is the snapshot of a local assignment, wants to be executed first, there are mainly two cases. And $case_4$ modeled as $case_4(s,t) =_{df} case_{4\_1}(s,t) \vee case_{4\_2}(s,t)$ presents the both cases.

Now, we define the case $case_{4\_1}$ that the register write reg_write is demanded to read some global variables. Then, all the instructions, which are po-previous to the write, may be branching conditions and assignments. If the previous is a condition judgment, $NoDepd_1$ is supposed to be satisfied between the snapshots of it and reg_write. Otherwise, $NoDepd_3$ should hold between the snapshots of reg_write and the po-previous assignment.

$$\begin{array}{l} case_{4\_1}(s,t) \\ =_{df} \left(\begin{array}{c} \pi_1(\pi_1(hd(t))) \in Locals \wedge \pi_3(hd(t)) = 1 \\ \wedge\ \exists x \in domain(\pi_2(\pi_1(hd(t)))) \bullet x \in Globals \\ \wedge\ \forall a' \in s \bullet \left(\pi_3(a') = 1 \rightarrow \left(\begin{array}{c} (\pi_2(a') = 0 \wedge NoDepd_1(a', hd(t))) \vee \\ (\pi_2(a') = 1, 2, 3 \wedge NoDepd_3(a', hd(t))) \end{array} \right) \right) \end{array} \right) \end{array}$$

Here, we use $Locals$ to denote the set of all the local variables.

We start to give a brief introduction to $case_{4\_2}$. The difference from $case_{4\_1}$ is that in this case, the trace s can have the snapshot of Fence.

$$\begin{array}{l} case_{4\_2}(s,t) \\ =_{df} \left(\begin{array}{c} \pi_1(\pi_1(hd(t))) \in Locals \wedge \pi_3(hd(t)) = 1 \\ \wedge\ \forall x \in domain(\pi_2(\pi_1(hd(t)))) \bullet x \notin Globals \\ \wedge\ \forall a' \in s \bullet \left(\pi_3(a') = 1 \rightarrow \left(\begin{array}{c} \pi_2(a') = -1 \vee \\ (\pi_2(a') = 0 \wedge NoDepd_1(a', hd(t))) \vee \\ (\pi_2(a') = 1, 2, 3 \wedge NoDepd_3(a', hd(t))) \end{array} \right) \right) \end{array} \right) \end{array}$$

The analysis of a branching condition and that of a local assignment are similar. Hence, we ignore the detailed definition, which is denoted by $case_5$.

Finally, we give the definition of sequential composition.

$$traces(P;Q) = \bigcup_{c} seqcom(s,t), \text{ where, } c = s \in traces(P) \wedge t \in traces(Q)$$

Example 4. Consider the example $P;Q$, where $P =_{df} x := 1$, $Q =_{df} y := 1$, x and y are global variables. $P;Q$ is activated with $x = y = 0$. Figure 4 gives a description of the trace of P (i.e., s) and Q (i.e., t) respectively. tr is one trace of $P;Q$, which is interleaved from P and Q.

For simplicity, we do not exhibit the environment operations. Although there are many executing cases for $P;Q$, we only analyze one scenario shown above.

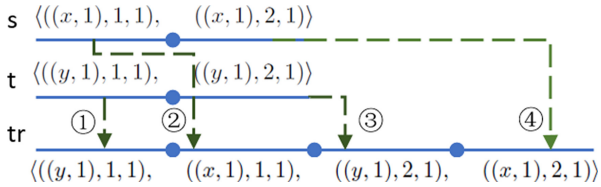

Fig. 4. The illustration of sequential composition.

1. The head $((y, 1), 1, 1)$ in t has no dependency with every snapshot in s, in consequence, it can be fetched firstly.
2. As the first element in s, $((x, 1), 1, 1)$ can be scheduled at any time, and here it is triggered in the second step.
3. We put the snapshot $((y, 1), 1, 1)$ in the third position of the trace tr of $P; Q$. Then, $((x, 1), 2, 1)$ can only be placed in the forth of tr. □

Parallel Construct. In this section, we discuss the trace semantics of parallel construct, which is formed by the merging of contributed components' traces.

Example 5. We use the example $P||Q$, where $P =_{df} x := 1$ and $Q =_{df} a := 1; b := x$, to illustrate how the trace semantics of parallel composition can be constructed. Here, the variable a and b are local, and x is a global variable.

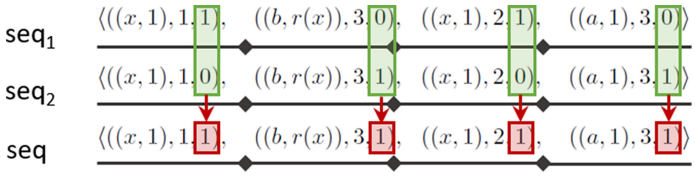

Fig. 5. The illustration of merging.

Here, we consider one scenario for the execution of $P||Q$. The operation committing the write to x is performed first. Then Q carries out the read from the location x. Finally, both processes complete their rest actions in proper order.

Then, the process P can produce the following sequence seq_1 shown in Fig. 5. The first and third snapshots are made by P itself, hence the last elements of them are both 1. The remaining snapshots in seq_1 with $eflag$ being 0 are contributed its environment Q. And Q yields the sequence seq_2 of snapshots.

Regardless of the fact that one action is done by the process P or Q, it is contributed by the parallel program $P||Q$. Hence, their merge gives a trace of $P||Q$ which is illustrated by seq in the above figure.

Note that, the thread Q carries out the read function $r(x)$ when the sequential composition just completes, because Q cannot classify the private and shared information if the parallel composition starts to execute. As a consequence, the value of $r(x)$ in Fig. 5 is 0. □

The sequence seq_1 of process P and seq_2 of Q are said to be comparable, if

1. $\pi_i^*(seq_1) = \pi_i^*(seq_2)$, where $i = 1, 2$.
   The above formula when $i = 1$ indicates that they are built from the same sequence of states, when $i = 2$ stands for that two sequences of operation type are the same.
2. Any state contributed by a parallel process cannot be made by both of its components, i.e., $2 \notin \pi_3^*(seq_1) + \pi_3^*(seq_2)$.

Next, their merge is defined as below.

$$Merge(seq, seq_1, seq_2) =_{df} \begin{pmatrix} (\pi_1^*(seq) = \pi_1^*(seq_1) = \pi_1^*(seq_2)) \wedge \\ (\pi_2^*(seq) = \pi_2^*(seq_1) = \pi_2^*(seq_2)) \wedge \\ (\pi_3^*(seq) = \pi_3^*(seq_1) + \pi_3^*(seq_2)) \wedge \\ (2 \notin \pi_3^*(seq_1) + \pi_3^*(seq_2)) \end{pmatrix}$$

Then, we define the trace semantics of parallel composition. The purpose for concatenating the sequence s contributed by the environment of P is to facilitate merging, and it is the same for Q, i.e., $\pi_3^*(s) \in 0^*$, and $\pi_3^*(t) \in 0^*$.

$$\begin{aligned} & traces(P||Q) \\ =_{df} & \{tr | tr_1 \in traces(P) \wedge tr_2 \in traces(Q) \wedge (Merge(tr, tr_1^\frown s, tr_2) \vee Merge(tr, tr_1, tr_2^\frown t))\} \end{aligned}$$

3 Algebraic Properties

Program properties can be expressed as algebraic laws (equations usually). In this section, we investigate algebraic laws for the MCA ARMv8 architecture including a set of sequential and parallel expansion laws. They can facilitate producing all the valid in-order and out-of-order executions. In our approach, every program can be expressed as a head normal form of guarded choice. Therefore, the linearizability of MCA ARMv8 is supported.

3.1 Guarded Choice

The introduction to guarded choice is to support the sequential and parallel expansion laws. It has the ability to model the execution of a program including various reorderings under ARMv8. $h\&(action, tid, index)[q] \hookrightarrow P$ is a guarded component. Here, h is a Boolean condition, and others are defined below.

1. (a) If the element *action* is the operation writing to the store buffer taking $\langle x = e \rangle$ for example, q is in the form of $h\&(action', tid, index')$, and $action'$ is propagating to the main memory $x = e$.
   (b) Furthermore, *action* may be assigning to a local variable $a = e$ or special actions such as Fence and cfence. Then q is ε.
   (c) In particular, $h\&(action, tid, index)[q]$ where *action* and q are both ε, indicates that the configuration is of a branching condition.

2. *tid* is the identity of the thread which performs the action.
3. We use the parameter *index* to denote the location of an action, and it is a pair shown as $(num, isMem)$. *num* indicates the sequence number of the action in the program order, and it starts from 1 for each single process. $isMem$ is to distinguish whether the action is propagation or not. If yes, it is 2, otherwise, it is 1. Example 6 below helps to illustrate the intuitive understanding of *index*.

Example 6. Consider the process $P =_{df} x := 1; a := x$, where x and a are global and local respectively. Since $x := 1$ is the first statement, two actions $\langle x = 1\rangle$ and $x = 1$ split from it have the same *num*. The value of *num* is 1 and it is framed in Fig. 6. $\langle x = 1\rangle$ and $x = 1$ target at the buffer and memory respectively. Then the values of $isMem$ are 1 and 2, and they are circled in Fig. 6. The action $a = x$ is extracted from the second statement $a := x$, thus its *num* is 2. Because it is not a memory action, its $isMem$ is 1. Hence the indices of the three actions $\langle x = 1\rangle$, $x = 1$ and $a = x$ are $(1, 1)$, $(1, 2)$ and $(2, 1)$. □

We use Example 7 below to describe the intuitive understanding of *tid*.

Example 7. Consider the parallel process $(P||Q)||R$ shown in Fig. 7. The left edge is assigned a label whose value is 1. Otherwise, the label is 2.

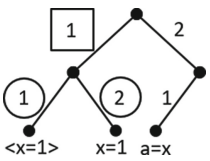

Fig. 6. The presentation of *index*.

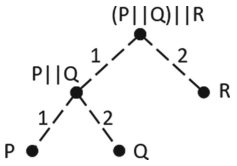

Fig. 7. The structure of thread id.

We assume that every sequential process has the thread id λ. For parallel composition, the thread id of $P||Q$ is $\langle 1\rangle$, and that of R is $\langle 2\rangle$. Lower down, the processes P and Q can be labeled by $\langle 1, 1\rangle$ and $\langle 1, 2\rangle$ respectively. From the point of view of the tree structure, P, Q and R are all leaf processes. Please note, for any thread id (i.e., *tid*), we have $tid^{\wedge}\lambda = tid$. □

Now we introduce the concept of guarded choice, which is in the form of $[\!]_{i\in I}\{h_i\&(action_i, tid_i, index_i)[q_i] \hookrightarrow P_i'\}$, where $h_i\&(action_i, tid_i, index_i)[q_i] \hookrightarrow P_i$ is a guarded component. For the component $h\&(action, tid, index)[q] \hookrightarrow P$, if h is satisfied, the subsequent is $(action, tid, index)[q] \hookrightarrow P$.

Every program can be represented in the form of a guarded choice. And then for MCA ARMv8, the guarded choice can only have the following three types.

1. $[\!]_{i\in I}\{h_i\&(action_i, tid_i, index_i)[(action_i', tid_i, index_i')] \hookrightarrow P_i'\}$
2. $[\!]_{i\in I}\{h_i\&(action_i, tid_i, index_i) \hookrightarrow P_i'\}$
3. $[\!]_{i\in I}\{h_i\&(action_i, tid_i, index_i)[(action_i', tid_i, index_i')] \hookrightarrow P_i'\}[\!]$
   $[\!]_{j\in J}\{h_j\&(action_j, tid_j, index_j) \hookrightarrow Q_j'\}$

- The first type of guarded choice is only composed of a set of global assignment components. The operation committing any memory write can be scheduled to execute, provided that the corresponding Boolean condition is satisfied.
- The second type of guarded choice is made up of local assignment, or Fence, or cfence, or branching condition components.
- The third type can be obtained through combining the first and second types of guarded choice.

3.2 Head Normal Form

Now, we assign every program P a normal form, which is named *head normal form, HF(P).* $HF(P)$ is in the form of guarded choice.
(1) For a global assignment, two actions committing to the write buffer and propagating to the whole memory are separated from it. Therefore, the two configurations corresponding to the above actions have the same num. However, the value of $isMem$ of the former is 1, while that of the latter is 2. And we use the notation E to denote the empty process.

$$HF(x := e) =_{df} [\![\{\text{true}\&(\langle x = e\rangle, \lambda, (1,1))[(x = e, \lambda, (1,2))] \hookrightarrow E\}$$

(2) For a local assignment, after the first step expansion, there remains the empty process. The treatment of Fence and cfence instructions is similar.

$$\begin{aligned} HF(a := e) &=_{df} [\![\{\text{true}\&(a = e, \lambda, (1,1)) \hookrightarrow E\} \quad .....HF(2\text{–}1) \\ HF(\text{Fence}) &=_{df} [\![\{\text{true}\&(\text{Fence}, \lambda, (1,1)) \hookrightarrow E\} \quad .....HF(2\text{–}2) \\ HF(\text{cfence}) &=_{df} [\![\{\text{true}\&(\text{cfence}, \lambda, (1,1)) \hookrightarrow E\} \quad .....HF(2\text{–}3) \end{aligned}$$

(3) For conditional, $h\&(\varepsilon, \lambda, (1,1))$ and $\neg h\&(\varepsilon, \lambda, (1,1))$ are used to produce the head normal form. That *action* is ε says that the evaluation does not have an effect on the registers, buffers and the unique memory.

$$HF(\text{if } h \text{ then } P \text{ else } Q) =_{df} ([\![\{h\&(\varepsilon, \lambda, (1,1)) \hookrightarrow P, \neg h\&(\varepsilon, \lambda, (1,1)) \hookrightarrow Q\})$$

(4) With regard to iteration, the analysis of it is similar to that of conditional.

$$\begin{aligned} &HF(\text{while } h \text{ do } P) \\ =_{df}\ &\big([\![\{h\&(\varepsilon, \lambda, (1,1)) \hookrightarrow (P; \text{while } h \text{ do } P), \quad \neg h\&(\varepsilon, \lambda, (1,1)) \hookrightarrow E\}\big) \end{aligned}$$

The definition of the head normal form for sequential and parallel composition can be achieved, with the application of corresponding expansion laws which are discussed in the following section.

3.3 Algebraic Laws

In this section, we study a set of sequential and parallel expansion laws. Based on these laws, every program can be converted to a guarded choice, which supports the linearizability of the MCA ARMv8 architecture.

Firstly, we focus on sequential expansion laws. Law (*guar*–1) indicates that the sequential composition distributes leftward over guarded choice.

(guar–1) $[\!]_{i\in I}\{P_i\};Q = [\!]_{i\in I}\{P_i;Q\}$

As a special case of the law (*guar*–1), law (*seq*–1) teaches us to transfer the program into configurations statement by statement. And the subsequent program Q is only attached to the tail of the selected P_i.

(seq–1) Let $P = [\!]_{i\in I}\{h_i\&(action_i, tid_i, index_i)[q_i] \looparrowright P_i'\}$
Then $P;Q = [\!]_{i\in I}\{h_i\&(action_i, tid_i, index_i)[q_i] \looparrowright (P_i';Q)\}$

After the transformation, we construct the relations among those configurations. Except for $h\&(action, tid, index)[q]$ fetched, the parameter *num* of every configuration left increases 1 to guarantee the program order. Law (*seq*–2) describes this, and *seq* denotes the sequence of the remaining configurations.

(seq–2) $h\&(action, tid, index)[q] \looparrowright seq = (h\&(action, tid, index) \rightarrow q) \hookrightarrow (seq \uparrow 1)$

Table 3. The description of three operators.

Operator	Exhibiting configurations	Program order relation	Fixed executing order
$\looparrowright$	√	×	×
$\hookrightarrow$	√	√	×
$\rightarrow$	√	√	√

Note that, the operator $\looparrowright$ is used to connect the configurations with original indices. Different from $\looparrowright$, the operator $\hookrightarrow$ links the configurations whose indices can reflect the program order (*po*) relation. The configurations connected by the two operators above can still be reordered, but those linked by the operator $\rightarrow$ cannot. Table 3 gives a brief and intuitive description of them.

Now, we give the definition of the function $seq \uparrow 1$. Only *num* in each configuration in *seq* adds 1, and other parameters remain unchanged. Here, '/' denotes the replacement operator.

$$seq \uparrow 1 =_{df} \forall h\&(action, tid, index) \in seq \bullet \\ seq[h\&(action, tid, (num + 1, isMem))/h\&(action, tid, (num, isMem))]$$

Example 8. Consider the sequential process $P;Q$, where $P =_{df} x := 1$, $Q =_{df} a := x$, and x and a are global and local respectively.

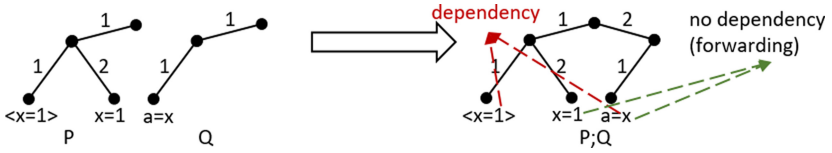

Fig. 8. The combination of configurations.

With the laws (*seq*–1) and (*seq*–2), we get the normal form of $P;Q$ formalized as below. The combination of configurations of P and Q are shown in Fig. 8. For simplicity, if the guard is true, it is ignored.

$$HF(x := 1; a := x) = (\langle x = 1\rangle, \lambda, (1,1))[(x = 1, \lambda, (1,2))] \nrightarrow (a = x, \lambda, \boxed{(1,1)})$$
$$= ((\langle x = 1\rangle, \lambda, (1,1)) \rightarrow (x = 1, \lambda, (1,2))) \hookrightarrow (a = x, \lambda, \boxed{(2,1)}) \quad \square$$

Law (*seq*–3) is used to obtain all the configuration sequences (including the results of reorderings) under MCA ARMv8. The first configuration with the least *num*, formalized as c_{11}, can always be scheduled. If we want to select the configuration after the operator $\hookrightarrow$ and its *num* is greater than that of c_{11}, modeled as c_{i1} where $i \neq 1$, the conditions covered by $cond_i$ should be satisfied.

$$\textbf{(seq–3)}\ (c_{11} \rightarrow c_{12} \rightarrow ...c_{1n_1}) \hookrightarrow (c_{21} \rightarrow c_{22} \rightarrow ...c_{2n_2}) \hookrightarrow ...(c_{m1} \rightarrow c_{m2} \rightarrow ...c_{mn_m})$$
$$= c_{11} \rightarrow \boxed{(c_{12} \rightarrow ...c_{1n_1})} \hookrightarrow (c_{21} \rightarrow c_{22} \rightarrow ...c_{2n_2}) \hookrightarrow ...(c_{m1} \rightarrow c_{m2} \rightarrow ...c_{mn_m})$$
$$\parallel c_{21} \rightarrow (c_{11} \rightarrow c_{12} \rightarrow ...c_{1n_1}) \hookrightarrow \boxed{(c_{22} \rightarrow ...c_{2n_2})} \hookrightarrow ...(c_{m1} \rightarrow c_{m2} \rightarrow ...c_{mn_m}) \text{ if } cond_2$$
$$\parallel ...$$
$$\parallel c_{m1} \rightarrow (c_{11} \rightarrow c_{12} \rightarrow ...c_{1n_1}) \hookrightarrow (c_{21} \rightarrow c_{22} \rightarrow ...c_{2n_2}) \hookrightarrow ... \boxed{(c_{m2} \rightarrow ...c_{mn_m})} \text{ if } cond_m$$

$cond_i$ has a number of situations, and these situations are similar to $case_j$ under the trace model (page 10), where $j \in \{1,2,3,4,5\}$. For lack of space, we only give the description and formalization of the situation that is corresponding to $case_1$, combining the features of the algebraic model in the following.

If the action in c_{i1} is a Fence instruction, any configuration c whose *num* is less than that of c_{i1} can only have an action in the form of $a = e$. Furthermore, the expression e does not refer to global variables. In a consequence, c has nothing to do with any global variable, and we use *dom* to collect all the variables appearing in $a = e$. Then this situation is formalized as below.

$$\forall c \bullet \left(\begin{array}{l} (\pi_1(\pi_3(c)) < \pi_1(\pi_3(c_{i1}))) \rightarrow \\ (\pi_1(c) \text{ is in the form of part of HF(2–1)} \wedge \forall x \in dom(\pi_1(c)) \bullet x \notin Globals) \end{array} \right)$$

Example 8: Continuation

According to the dependencies in Fig. 8, with the first application of the law (*seq*–3), only the configuration $(\langle x = 1\rangle, \lambda, (1,1))$ can be the head. After removing it, we apply the law (*seq*–3) for the second time, and both of the remaining configurations can be scheduled. The formalization is shown as below.

$$\begin{aligned} HF(x := 1; a := x) =& (\langle x = 1\rangle, \lambda, (1,1)) \to ((x = 1, \lambda, (1,2)) \hookrightarrow (a = x, \lambda, (2,1))) \\ =& (\langle x = 1\rangle, \lambda, (1,1)) \to (x = 1, \lambda, (1,2)) \to (a = x, \lambda, (2,1)) \\ & [\!](\langle x = 1\rangle, \lambda, (1,1)) \to (a = x, \lambda, (2,1)) \to (x = 1, \lambda, (1,2)) \end{aligned}$$ □

Next, we consider the parallel expansion law. Our parallel model can be explained as an interleaving model. The detail we pay attention to is that when the configuration in the left branch is selected, the prefix $\langle 1\rangle$ should be added to the corresponding tid_i. The prefix $\langle 2\rangle$ is attached to the corresponding tid_j with the configuration in the right being chosen.

(par–1) Let $P = [\!]_{i\in I}\{h_i \& (action_i, tid_i, index_i) \to P_i'\}$,
$Q = [\!]_{j\in J}\{h_j \& (action_j, tid_j, index_j) \to Q_j'\}$
Then $P||Q = [\!]_{i\in I}\{h_i \& (action_i, \langle 1\rangle^\frown tid_i, index_i) \to (P_i'||Q)\}$
$[\!][\!]_{j\in J}\{h_j \& (action_j, \langle 2\rangle^\frown tid_j, index_j) \to (P||Q_j')\}$

Example 9. Consider the parallel program $P||Q$, where $P =_{df} x := 1$, $Q =_{df} a := 1; b := x$, a and b are local variables, and x is a global variable.

$$\begin{aligned} & HF(P||Q) = HF(x := 1)||HF(a := 1; b := x) \\ =& ((\langle x = 1\rangle, \lambda, (1,1)) \to (x = 1, \lambda, (1,2)))||((a = 1, \lambda, (1,1)) \hookrightarrow (b = x, \lambda, (2,1))) \\ =& \left((\langle x = 1\rangle, \lambda, (1,1)) \to (x = 1, \lambda, (1,2)) \right) || \left(\begin{array}{l} (a = 1, \lambda, (1,1)) \to (b = x, \lambda, (2,1)) \\ [\!] \; (b = x, \lambda, (2,1)) \to (a = 1, \lambda, (1,1)) \end{array} \right) \end{aligned}$$

For lack of space, we only describe the generation of one sequence of $P||Q$ shown in Fig. 9 here. □

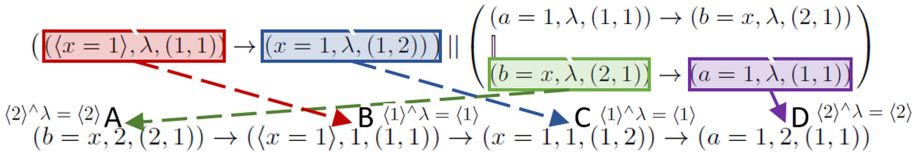

Fig. 9. One configuration sequence of $P||Q$.

4 Conclusion and Future Work

The MCA ARMv8 architecture allows out of order execution through thread-local out-of-order, speculative execution and thread-local buffering. In this paper, we have studied the trace semantics for ARMv8, acting in the denotational semantics style. In addition, a set of algebraic laws including sequential and parallel expansion laws has been investigated with the concept of the guarded choice. Therefore, the linearizability of ARMv8 is supported in our model. Our semantics study for MCA ARMv8 is based on UTP approach.

In the future, we would like to continue our work on ARMv8. We plan to explore further relating theories for the ARMv8 architecture [19–21]. Using the theorem proof assistant Coq [22–24] to formalize the UTP-based semantics for ARMv8 is also in our plan.

Acknowledgements. This work was partly supported by National Natural Science Foundation of China (Grant Nos. 61872145 and 62032024) and Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (Grant No. ZF1213).

A Read Function

Now, we present the read function r in detail. Above all, we need to judge if the variable read from is global. If true, we introduce the function g to complete the following operations. Otherwise, the function l is given. For simplicity, we only use $r(x)$ in the snapshots. Here, $Globals$ is the set of all the global variables.

$$r(x, tr^\frown\langle event\rangle) =_{df} g(x, tr^\frown\langle event\rangle) \lhd\ x \in Globals\ \rhd l(x, tr^\frown\langle event\rangle)$$
$$r(x, \langle\rangle) =_{df} g(x, \langle\rangle) \lhd\ x \in Globals\ \rhd l(x, \langle\rangle)$$

The read mechanism for global variables supported by this architecture is that when a thread performs a read, if its buffer cannot provide the concrete value, the shared memory will be explored.

$$g(x, tr^\frown\langle event\rangle) =_{df} \begin{pmatrix} m(x, tr^\frown\langle event\rangle) \\ \lhd \begin{pmatrix} w(x, tr^\frown\langle event\rangle) = \text{null} \vee \\ cnt_1(x, tr^\frown\langle event\rangle) = cnt_2(x, tr^\frown\langle event\rangle) \quad ...(A.1) \end{pmatrix} \rhd \\ w(x, tr^\frown\langle event\rangle) \end{pmatrix}$$
$$g(x, \langle\rangle) =_{df} m(x, \langle\rangle)$$

It means that the execution of g will jump to that of m, if the values of x have not been committed to the buffer, or the writes to x have all been propagated to the memory. The latter situation is modeled as the formula (A.1) in the trace model. It illustrates that the number of the snapshots which contain x and target at the buffer, and that aiming at memory contributed by the same thread are identical. The numbers mentioned above can be calculated by the functions cnt_1 and cnt_2. We ignore the definition of cnt_2, because it is similar to that of cnt_1.

$$cnt_1(x, tr^\frown\langle event\rangle) =_{df} \begin{pmatrix} cnt_1(x, tr) + 1 \\ \lhd \begin{pmatrix} \text{ASCII}(\pi_1(\pi_1(event))) = \text{ASCII}(x) \\ \wedge \pi_2(event) = 1 \wedge \pi_3(event) = 1 \end{pmatrix} \rhd \\ cnt_1(x, tr) \end{pmatrix}$$
$$cnt_1(x, \langle\rangle) =_{df} 0$$

The function w is used to search the store buffer. Since we always want the most recent value, the trace (the sequence of snapshots) will be checked in reverse order, and the same is true for the functions as below. When executing w, for each snapshot, we first examine whether its $oflag$ and $eflag$ are both 1, because all threads can see their own buffers merely. If the conditions are satisfied, we have a look at the variable contained in $\pi_1(\pi_1(event))$ of the snapshot. Once it is identical to the one that we want to read, the corresponding value $\pi_2(\pi_1(event))$ is returned, and the process terminates. If we do not achieve anything until the trace becomes ε, null will be assigned to this function.

$$
\begin{aligned}
& w(x, tr^{\frown}\langle event\rangle) \\
=_{df} & \begin{pmatrix} (\pi_2(\pi_1(event)) \triangleleft \ \mathrm{ASCII}(\pi_1(\pi_1(event))) = \mathrm{ASCII}(x) \ \triangleright w(x, tr)) \\ \quad \triangleleft \ \pi_2(event) = 1 \wedge \pi_3(event) = 1 \ \triangleright \\ w(x, tr) \end{pmatrix} \\
& w(x, \langle\rangle) =_{df} \text{null}
\end{aligned}
$$

We know that ASCII is used to specify the binary numbers of common symbols.

We use the function m to seek the shared memory for the value of a specific variable. Due to the fact that the main memory is visible to all threads, we are only demanded to check whether $oflag$ of the snapshot we meet is 2 or not. The remainder is similar to that of w. However, the difference between them is that the return value of the function m is set to the initial value 0 if we cannot get the value from the trace.

$$
\begin{aligned}
& m(x, tr^{\frown}\langle event\rangle) \\
=_{df} & \begin{pmatrix} (\pi_2(\pi_1(event)) \triangleleft \ \mathrm{ASCII}(\pi_1(\pi_1(event))) = \mathrm{ASCII}(x) \ \triangleright m(x, tr)) \\ \quad \triangleleft \ \pi_2(event) = 2 \ \triangleright \\ m(x, tr) \end{pmatrix} \\
& m(x, \langle\rangle) =_{df} 0
\end{aligned}
$$

When reading a variable from the register, what we should do is to check whether $oflag$ is 3 and $eflag$ is 1, because the registers are all private.

$$
\begin{aligned}
& l(x, tr^{\frown}\langle event\rangle) \\
=_{df} & \begin{pmatrix} (\pi_2(\pi_1(event)) \triangleleft \ \mathrm{ASCII}(\pi_1(\pi_1(event))) = \mathrm{ASCII}(x) \ \triangleright l(x, tr)) \\ \quad \triangleleft \ \pi_2(event) = 3 \wedge \pi_3(event) = 1 \ \triangleright \\ l(x, tr) \end{pmatrix} \\
& l(x, \langle\rangle) =_{df} 0
\end{aligned}
$$

Based on the read function generated from the read mechanism of the MCA ARMv8 architecture, we can know that the private information will not be visible to other threads.

References

1. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. Proc. ACM Program. Lang. **2**(POPL), 1–29 (2017)
2. Pulte, C.: The Semantics of Multicopy Atomic ARMv8 and RISC-V. University of Cambridge (2019)
3. Flur, S., et al.: Modelling the ARMv8 architecture, operationally: concurrency and ISA. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 608–621 (2016)
4. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_27

5. Colvin, R.J., Smith, G.: A wide-spectrum language for verification of programs on weak memory models. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 240–257. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_14
6. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall, Englewood Cliffs (1998)
7. Plotkin, G.D.: A Structural Approach to Operational Semantics. Aarhus University (1981)
8. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge (1981)
9. Hoare, C.A.R., et al.: Laws of programming. Commun. ACM **30**(8), 672–686 (1987)
10. Winter, K., Smith, G., Derrick, J.: Modelling concurrent objects running on the TSO and ARMv8 memory models. Sci. Comput. Program. **184**, 102308 (2019)
11. Smith, G., Winter, K., Colvin, R.J.: Linearizability on hardware weak memory models. Formal Aspects Comput. **32**, 1–32 (2019)
12. Winter, K., Smith, G., Derrick, J.: Observational models for linearizability checking on weak memory models. In: International Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 100–107. IEEE (2018)
13. Kavanagh, R., Brookes, S.: A denotational semantics for SPARC TSO. Electron. Notes Theor. Comput. Sci. **336**, 223–239 (2018)
14. Colvin, R.J., Smith, G.: A high-level operational semantics for hardware weak memory models, arXiv preprint arXiv:1812.00996 (2018)
15. Brookes, S.: Full abstraction for a shared-variable parallel language. Inf. Comput. **127**(2), 145–163 (1996)
16. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Hoboken (1985)
17. Smith, G., Coughlin, N., Murray, T.: Value-dependent information-flow security on weak memory models. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 539–555. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_32
18. Sorin, D.J., Hill, M.D., Wood, D.A.: A primer on memory consistency and cache coherence. Synthesis Lect. Comput. Archit. **6**(3), 1–212 (2011)
19. Zhu, H., Yang, F., He, J., Bowen, J.P., Sanders, J.W., Qin, S.: Linking operational semantics and algebraic semantics for a probabilistic timed shared-variable language. J. Logic Algebraic Program. **81**(1), 2–25 (2012)
20. He, J., Hoare, C.A.R.: From algebra to operational semantics. Inf. Process. Lett. **45**(2), 75–80 (1993)
21. Hoare, C.A.R., He, J., Sampaio, A.: Algebraic derivation of an operational semantics. In: Proof, Language, and Interaction: Essays in Honour of Robin Milner, pp. 77–98 (2000)
22. Sheng, F., Zhu, H., He, J., Yang, Z., Bowen, J.P.: Theoretical and practical aspects of linking operational and algebraic semantics for MDESL. ACM Trans. Softw. Eng. Methodol. (TOSEM) **28**(3), 1–46 (2019)
23. Huet, G., Kahn, G., Paulin-Mohring, C.: The Coq Proof Assistant a Tutorial (2005)
24. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2013)