ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**He Jifeng, Oxford University Computing Laboratory, UK, John Cooke,
Loughborough University, UK, and Peter Wallis, University of Bath, UK (Eds)**

BCS-FACS 7th Refinement Workshop

Proceedings of the BCS-FACS 7th Refinement Workshop, Bath, 3-5
July 1996

Design and Verification of a Coherent Shared Memory

He Jifeng, A. McIsaac and G. Barrett

Published in Collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

ISBN 3-540-76104-7

Design and Verification of a Coherent Shared Memory

He Jifeng *

A. McIsaac and G. Barrett[†]

1 Introduction

This paper describes the formalization and verification of a coherent memory system described in [9]. It addresses questions that arose in the architectural design of a family of commercial 64-bit microprocessors. We are concerned with multi-processor systems in which programs execute concurrently on several threads that share areas of memory. If the operation of the load and store memory instructions is defined on a model that contains both main memory and caches, then two questions arise: what extra machine instructions are required in order to make it possible to maintain coherency of the caches; and what requirements are imposed on software to ensure that it does keep the caches coherent. This paper treats such a shared system as a set of mathematical objects, and addresses two related issues: specification and refinement.

We construct an abstract model for such a shared system first where there is just the main memory, and load and store instructions operate directly on memory, while cache control instructions have no effect. In this model, communication between co-operating programs is via the main memory only, and each thread owns a set of registers as its private store. The behaviour of each instruction is modelled by an assignment which describes the state change caused by the execution of the instruction. As in [4, 8] the behaviours of the system running a set of programs concurrently are specified by a set of traces which represent possible interleaving of interactions between threads and the main memory. Each individual program is described by a set of *local traces* which describes the change of local registers that its user can observe independently. The notion of *local equivalence* presented in [5, 7] is used to identify two global traces which generate the same local traces for all of its users. Then we provide a concise model for a new machine where every processor is equipped with a cache as its local store, and load and store instructions operate in the first instance on the caches. In the second model, execution of a cache control instruction has a certain effect on both main memory and caches, and furthermore communication between threads of the same processor can be via the local store.

Using refinement we show what it means for a memory system with caches to be a suitable replacement for one without caches in an environment of independent users. As the programs that run on a shared memory are constructed using concurrent combinations of isolated but co-operating users, the appropriate criterion for a replacement is a *local refinement* of the original one; i.e., no user can individually detect the substitution of a single store by a replicated one. That is to say, any local trace which can be recorded in the system with caches can also be generated by the system without caches. In order to establish such a refinement relation between the old machine and the new machine, we first use the data refinement technique advocated in [6, 10] to build a link invariant between the states in the new machine with the states in the old one, and we then show that execution of each machine instruction in the two machines preserves such a link. Based on that link invariant, we demonstrate that the concurrent execution of *locally regular programs* within a single processor in two memory systems will have the same effect on the local registers. To ensure that the concurrent execution of programs in different processors produce the same local traces in two machines, we introduce the notion of *global regularity*. Finally we show that it is possible to reorganize the interleaving of machine instruction executions of *coherent programs* so that it produces the same local traces as that of the regular one.

*Oxford University Computing Laboratory

[†]SGS-Thomson Microelectronics Limited, 1000 Aztec West, Bristol BS12 4SQ

The rest of this paper is organized as follows. Section 2 contains a model for a shared memory system without caches, and a formal description for a simple machine language. It also introduces the notion of local traces and local equivalence. We devote Section 3 to the memory system with caches. Section 4 shows how to establish a proper link between the new machine and the old one, and explains the proof strategy. In Section 5 we define an invariant link between the states of two machines and prove that it is preserved by all machine instructions. Section 6 first examines concurrent execution of programs in one processor, and introduces the notion of local regularity. It later shows that the execution of a regular sequence of machine instructions in two machines has the same effect on the local registers. The final section concludes the paper by showing that the behaviours of coherent programs are characterised by a set of regular traces.

2 A Shared Memory System without Caches

This section presents a model AM for a machine consisting of a number of processors sharing a single monolithic store. Each individual processor runs a set of programs in parallel. It is postulated that communication between co-operating programs is only via the single store, and memory access instructions operate directly on the main memory. The model serves as a simple interface to the users of the machine language. We use $M1$ to stand for the main memory which maps a location (an actual address) to a value held in that location:

$$M1 : Loc \rightarrow Value$$

where Loc and $Value$ represent the sets of locations and values respectively.

We use P to stand for the finite set of processor names. Each processor $p \in P$ is associated with a set T_p of threads. A thread $t \in T_p$ owns a set of private registers $R1_{p,t}$:

$$R1_{p,t} : Name \rightarrow Value$$

where $Name$ stands for a finite set of names.

The components of a machine state are:

1. the main memory $M1$;
2. a collection of registers $R1_{p,t}$ for $p \in P$ and $t \in T_p$.

We have not included the program pointer, virtual address translation and other control registers into the machine state since they are irrelevant to the verification of the design of a coherent memory system. To specify a complete set of machine instructions, the machine state would need some additional components. We will use the notation $A\Sigma$ to represent the set of machine states in the model AM , and let the variable $a\sigma$ range over $A\Sigma$. In the following we first specify the memory access instructions.

The load instruction $\mathbf{Load}(r, l)_{p,t}$ executed by a thread t in the processor $p \in P$ is associated with a pair (r, l) of parameters where the first one stands for a register name, and the second one for a memory location. The instruction reads the contents of the memory location l into the register $R1_{p,t}[r]$.

$$\mathbf{Beh}(\mathbf{Load}(r, l)_{p,t}) \stackrel{def}{=} R1_{p,t}[r] := M1[l]$$

The instruction $\mathbf{Store}(r, l)_{p,t}$ has the same parameters as the load instruction. It writes the contents of register $R1_{p,t}[r]$ back to the memory location l .

$$\mathbf{Beh}(\mathbf{Store}(r, l)_{p,t}) \stackrel{def}{=} M1[l] := R1_{p,t}[r]$$

The flush instruction $\mathbf{Flush}(l)_{p.t}$ has a single parameter l for a memory location. The change within the memory system caused by the execution of the flush instruction is invisible to the external users; in other words, it has no effect on the machine state.

$$\mathbf{Beh}(\mathbf{Flush}(l)_{p.t}) \stackrel{\text{def}}{=} \text{skip}$$

Like the flush instruction, the validation instruction $\mathbf{Validate}(l)_{p.t}$ leaves the machine state unchanged.

$$\mathbf{Beh}(\mathbf{Validate}(l)_{p.t}) \stackrel{\text{def}}{=} \text{skip}$$

The remaining instructions operate on the registers only, and do not depend on the main memory. We will use $\mathbf{Rinstr}_{p.t}$ to range over this set of instructions. Their definitions are irrelevant to the design of the coherent cache, and thus left unspecified in this paper.

A *trace* of the behaviour of the system is a finite sequence of interactions between the memory and its users in which the system has engaged up to some moment. In the model AM a trace has the following form

$$\langle a\sigma_0, \mathbf{instr}_1, a\sigma_1, \mathbf{instr}_2, \dots, \mathbf{instr}_n, a\sigma_n \rangle$$

where each $a\sigma_i$ is a machine state, and for each i , the execution of the instruction $\mathbf{instr}_i$ at the state $a\sigma_i$ leads to the state $a\sigma_{i+1}$. We assume that all instructions are deterministic. In this case all machine states (except the first one) in a trace are uniquely determined by the instructions in the trace, and can be dropped for convenience.

Since there is a set of programs running in parallel in the machine, what an individual user can observe during the execution is a local trace which consists of all interactions between this user and the memory. The relation between a trace of the system and a local trace is defined as follows:

Definition 2.1 (Local trace)

Let tr be a trace $\langle a\sigma_0, \mathbf{instr}_1, a\sigma_1, \mathbf{instr}_2, \dots, \mathbf{instr}_n, a\sigma_n \rangle$.

The *local trace* generated by tr for a thread (say $p.t$) is the sequence of actions in tr performed by that thread:

$$\langle a\sigma_{j(1)}.R1_{p.t}, \mathbf{instr}_{j(1)}, a\sigma_{j(2)}.R1_{p.t}, \dots, a\sigma_{j(k)}.R1_{p.t}, \mathbf{instr}_{j(k)}, a\sigma_{j(k)+1}.R1_{p.t} \rangle$$

where $\langle \mathbf{instr}_{j(1)}, \dots, \mathbf{instr}_{j(k)} \rangle$ is the subsequence of $\langle \mathbf{instr}_1, \dots, \mathbf{instr}_n \rangle$ consisting of all instructions executed by $p.t$.

We will use the notation $\mathbf{localtrace}_{p.t}(tr)$ to represent the local trace yielded by tr for the thread $p.t$. □

Definition 2.2 (Local equivalence)

Traces $tr1$ and $tr2$ are *locally equivalent*, denoted by

$$tr1 \approx tr2$$

if they satisfy the following conditions:

1. They start at the same state.
2. None of the threads can individually detect their difference; i.e.:

$$\forall p \in P, \forall t \in T_p \bullet \mathbf{localtrace}_{p.t}(tr1) = \mathbf{localtrace}_{p.t}(tr2)$$

3. They end with the same state. In other words, the substitution of $tr1$ by $tr2$ has no effect on the future behaviour of the system.

Clearly $\approx$ is an equivalence relation. □

$\approx$ is preserved by concatenation.

Lemma 2.3 If $u1 \approx v1$ and $u2 \approx v2$ then $(u1; u2) \approx (v1; v2)$. □

3 A Memory System with Caches

When a set of processor shares a single monolithic store, accessing store on a remote processor is relatively slow when compared with accessing local store. This section investigates how to replace a single store $M1$ by replicated store. The hope is that all programs run to completion faster on the new machine than on the old one. This section gives an implementation of replicated store.

The new machine has a similar structure to the old one, except that it associates each processor with a cache. Now communication between threads of the same processor can be via that cache. Furthermore, memory access instructions will operate on both caches (acting as a local store) and main memory (serving as a global store). The components of a state of this new machine include

1. The main memory $M2 : Loc \rightarrow Value$.
2. Each processor $p \in P$ is equipped with a cache

$$C_p : Loc \rightarrow Value \times \{dirty, clean, invalid\}$$

where the first component of $C_p[l]$ (denoted $C_p[l].value$) is the data being held in $C_p[l]$; and the second component (denoted $C_p[l].state$) indicates whether that data is clean, dirty or invalid.

3. a collection of registers $R2_{p.t}$.

We will use Σ to stand for the set of all machine states, and σ to range over Σ .

The instruction $Load(r, l)_{p.t}$ of the new machine reads the contents of local store location l into register $R2_{p.t}[r]$ if the state of that location is valid; otherwise it loads the contents of $M2[l]$ to both $R2_{p.t}[r]$ and $C_p[l].value$ and sets the state of $C_p[l]$ to clean. The behaviour of its execution can be specified by a conditional:

$$\begin{aligned} \text{Beh}(Load(r, l)_{p.t}) &\stackrel{def}{=} R2_{p.t}[r], C_p[l] := M2[l], (M2[l], clean) \\ &\triangleleft C_p[l].state = invalid \triangleright R2_{p.t}[r] := C_p[l].value \end{aligned}$$

where the notation $P \triangleleft b \triangleright Q$ stands for **if** b **then** P **else** Q .

The store instruction $Store(r, l)_{p.t}$ sends the contents of the register $R2_{p.t}[r]$ back to the local store location l , and sets its state to dirty.

$$\text{Beh}(Store(r, l)_{p.t}) \stackrel{def}{=} C_p[l] := (R2_{p.t}[r], dirty)$$

$Flush(l)_{p.t}$ is a cache control instruction, and acts as a communication between a local store with the global store. It flushes the data held in the local store location l into the global store location l , and cleans the state of that local location if it is initially dirty; otherwise it leaves the machine state unchanged.

$$\text{Beh}(Flush(l)_{p.t}) \stackrel{def}{=}$$

$$M2[l], C_p[l].state := C_p[l].value, clean \triangleleft C_p[l].state = dirty \triangleright skip$$

$Validate_{p.t}(l)$ is also a cache control instruction. Its execution invalidates the contents of local store location l when its state is clean initially, otherwise it has no effect.

$$\text{Beh}(Validate(l)_{p.t}) \stackrel{\text{def}}{=} C_p[l].state := invalid \triangleleft C_p[l].state = clean \triangleright skip$$

The remaining machine instructions have no effect on either main memory or caches, and are defined in the same way as as their counterparts in the old machine.

4 Relationship between the systems

This section aims to find an appropriate link between the old machine and the new one. The old machine serves as a specification of a shared memory system, and it has a rather simple machine language. The users of such a machine can be designers of a compiler. The simplicity of the target language is a great help for the establishment of correctness of a compiler where the time performance of the target program is not the main concern: it is possible to separate the issues of correct compilation and efficient use of the cache control instructions. The replicated store in the new machine provides a more efficient memory access service to its users.

A desired property of the implementation is upward compatibility; that is every program that runs on the monolithic store must run in the same way on the replicated store, but with possibly different timing. Of course, this will not in general be the case: the way the load and store instructions operate allows the possibility of introducing incoherency. In order to reduce the interference among the users of the memory system, we impose the following protocols on the programs executed in the processors.

Definition 4.1 (Protocol)

A sequence s of machine instructions is said to follow the write protocol if for any thread each of its store instructions on a location is immediately followed by a flush instruction on that location; i.e. for any thread $p.t$ let u be the sequence of machine instructions in s executed by that thread, then

$$\forall i \bullet (1 \leq i < \#u \wedge u[i] = \mathbf{Store}(-, l)_{p.t} \Rightarrow u[i+1] = \mathbf{Flush}(l)_{p.t})$$

where $\#u$ stands for the length of the sequence u , and $u[i]$ for its i th element.

A sequence s of machine instructions is said to follow the read protocol if for any thread each of its load instruction is immediately preceded and followed by a validate instruction and a flush instruction on the same location; i.e., let u be the sequence of machine instructions in s executed by a thread $p.t$, then

$$\begin{aligned} & \forall i \bullet (1 \leq i < \#u \wedge u[i] = \mathbf{Load}(-, l)_{p.t} \Rightarrow u[i+1] = \mathbf{Flush}(l)_{p.t}) \wedge \\ & (1 \leq i \leq \#u \wedge u[i] = \mathbf{Load}(-, l)_{p.t} \Rightarrow (i > 1 \wedge u[i-1] = \mathbf{Validate}(l)_{p.t})) \end{aligned}$$

A sequence s of machine instructions is *coherent* if it follows both read and write protocols. □

The intuitive justification for this definition is that if all writes to memory in the new system were performed by means of an atomic operation consisting of a store followed by a flush, then they would immediately become visible to all memory users, and furthermore caches that were initially clean would remain so; and if all reads from memory were performed by means of an atomic operation consisting of a validate followed by a load, and all caches are clean, then the values read would be those in main memory, which are the most recently written. The additional flush is required because instructions on different threads may interleave, so that the store and flush are not performed as one atomic operation. Then one thread could store a value to its cache, and another thread sharing the same cache could load this

value to a register before the first thread had performed a flush. If the second thread did not flush this value immediately, it could store the value from its register to a different cache location, and then flush the value from this second location, all before the first thread had flushed the first value. From the point of view of other processors, the store to the second location would then have occurred first, which would be inconsistent with this processor's view that the store to the first location had occurred first.

The connection between the behaviours of the two machines will be established for traces whose instruction sequences are coherent. For a given state (say $a\sigma_0$) we will use the notation $\mathbf{Traces}(AM, a\sigma_0)$ to stand for the set of all coherent traces with the initial state $a\sigma_0$

$$\mathbf{Traces}(AM, a\sigma_0) \stackrel{\text{def}}{=} \{ \langle a\sigma_0, s \rangle \mid s \text{ is coherent} \}$$

The behaviour of the machine with the initial state $a\sigma_0$ is defined as the set of all local traces:

$$\mathbf{Localtraces}(AM, a\sigma_0) \stackrel{\text{def}}{=} \{ \{ \mathbf{localtrace}_{p,t}(tr) \mid p \in P \wedge t \in T_p \} \mid tr \in \mathbf{Traces}(AM, a\sigma_0) \}$$

Since the two machines operate on different state spaces we cannot compare their behaviour directly. Because the programs that run on the old machine are constructed using concurrent combinations of isolated but co-operating threads, the appropriate correctness criterion for an implementation is a refinement of the original one from each individual user's viewpoint; i.e., no thread can individually detect the substitution of a single store by a replicated one. We claim that the new machine is a *local refinement* of the original one in the following sense:

$$\mathbf{Localtraces}(M, \sigma_0) \subseteq \mathbf{Localtraces}(AM, a\sigma_0)$$

provided that on σ_0 all caches are invalid,
and the following binary relation holds between σ_0 and $a\sigma_0$:

$$\mathbf{Equal}(R) \wedge M1 = M2$$

where $\mathbf{Equal}(R)$ abbreviates

$$\forall p \in P, \forall t \in T_p \bullet R1_{p,t} = R2_{p,t}$$

For writers of code for the new system, this result means that, as long as their programs are coherent, they can use the old system as the abstract machine on which they view the programs as running. This is a first step towards providing guidelines for the use of the cache control instructions; it is possible to weaken the condition considerably when there is a disciplined way for threads to gain and relinquish exclusive access to areas of memory.

To prove the result, we adopt a three-step strategy:

1. The first step is to compare concurrent programs running in the same processor. We introduce a link invariant inv_p for each processor $p \in P$ which relates a state of the new machine to one in the old machine, and satisfies

$$inv_p \Rightarrow \mathbf{Equal}(R)$$

Then we introduce the notion of *safely-loaded* and *flush-tailed* sequences of machine instructions, and prove that executing such a sequence in two machines preserves the link invariant inv_p between the old machine states with their corresponding ones in the new machine.

2. Secondly we introduce the notion of *regular* trace which consists of a set of programs running in several processors. We show that running a regular sequence preserves the relation $\mathbf{Equal}(R)$.
3. Finally we demonstrate how a coherent trace in the new machine can be *reordered* into a regular one, and prove that the reordering relation is stronger than the local equivalence $\approx$.

In summary, this strategy bridges the gap between the old machine and the new one by giving the following link:

If the state σ_0 satisfies

$$\forall p \in P, \forall t \in T_p \bullet (C_p[l].state = invalid)$$

and the binary relation

$$\mathbf{Equal}(R) \wedge M1 = M2$$

holds between $a\sigma_0$ and σ_0 , then

$$\begin{aligned} \forall (\sigma_0, s) \in \mathbf{Traces}(M, \sigma_0), \exists (a\sigma_0, s1) \in \mathbf{Traces}(M, \sigma_0) \bullet \\ (\sigma_0, s) \approx (\sigma_0, s1) \wedge \\ \forall p \in P, \forall t \in T_p \bullet \mathbf{localtrace}_{p,t}(\sigma_0, s1) = \mathbf{localtrace}_{p,t}(a\sigma_0, s1) \end{aligned}$$

5 Link Invariant

The binary relation inv_p defined below relates a state of the new machine to one of the old machine. For any subsets $U1$ and $U2$ of Loc let

$$inv_p(U1, U2) \stackrel{def}{=} \mathbf{Equal}(R) \wedge \forall l \in U1 \bullet wink_p(l) \wedge \forall l \in U2 \bullet link_p(l)$$

where

$$\begin{aligned} link_p(l) \stackrel{def}{=} & (C_p[l].state = invalid \wedge M1[l] = M2[l]) \vee \\ & (C_p[l].state = dirty \wedge M1[l] = C_p[l].value) \vee \\ & (C_p[l].state = clean \wedge M1[l] = M2[l] = C_p[l].value) \end{aligned}$$

and

$$\begin{aligned} wink_p(l) \stackrel{def}{=} & (C_p[l].state = invalid \wedge M1[l] = M2[l]) \vee \\ & (C_p[l].state = dirty \wedge M1[l] = C_p[l].value) \vee \\ & (C_p[l].state = clean \wedge M1[l] = M2[l]) \end{aligned}$$

In order to show that the link invariant inv_p is preserved by the execution of every machine instruction we introduce the notion of *assertion* and *assumption* below. Let b stand for a Boolean expression, we use $b^\top$ to stand for the assumption of the condition b , which can be regarded as a miraculous test: it leaves the state unchanged (behaving like the program *skip*) if b is true; otherwise it behaves like the command *Miracle* (denoted by $\top$)

$$b^\top \stackrel{def}{=} skip \triangleleft b \triangleright \top$$

The assertion of b , $b_\perp$, also behaves like *skip* when b is true; otherwise it fails, behaving like the command *Abort* (denoted $\perp$).

$$b_\perp \stackrel{def}{=} skip \triangleleft b \triangleright \perp$$

The intended purpose of assumptions and assertions is to give *pre* and *postconditions* the status of programs. For example

$$a^\top; Q; b_\perp$$

represents the fact that the assumption of a is an obligation placed on the environment of program Q : if the environment fails to provide a state satisfying a , $a^\top$ behaves like *Miracle*, which makes the whole program behaves miraculously. On the other hand, an assertion is an obligation placed on the program Q itself. If Q fails to make b true

on its completion, it ends up like *Abort*. Assumption and assertion are governed by the following algebraic laws:

Lemma 5.1

$$\mathbf{L1} \quad b^\top; c^\top = (b \wedge c)^\top \quad b_\perp; c_\perp = (b \wedge c)_\perp.$$

$$\mathbf{L2} \quad b^\top; b_\perp = b^\top. \quad \square$$

The following five lemmas illustrate that the link invariant is preserved by all machine instructions.

Lemma 5.2 If $l \in U1 \cup U2$ then

$$\begin{aligned} & inv_p(U1, U2)^\top; \text{Beh}(\mathbf{Flush}(l)_{p.t}); \text{Beh}(\text{Flush}(l)_{p.t}) = \\ & inv_p(U1, U2)^\top; \text{Beh}(\mathbf{Flush}(l)_{p.t}); \text{Beh}(\text{Flush}(l)_{p.t}); inv_p(U1, U2)_\perp \end{aligned} \quad \square$$

Lemma 5.3 If $l \in U1$ then

$$\begin{aligned} & inv_p(U1, U2)^\top; \text{Beh}(\mathbf{Validate}(l)_{p.t}); \text{Beh}(\text{Validate}(l)_{p.t}) = \\ & inv_p(U1, U2)^\top; \text{Beh}(\mathbf{Validate}(l)_{p.t}); \text{Beh}(\text{Validate}(l)_{p.t}); inv_p(U1, U2 \cup \{l\})_\perp \end{aligned} \quad \square$$

Lemma 5.4

$$\begin{aligned} & inv_p(U1, U2)^\top; \text{Beh}(\mathbf{Store}(r, l)_{p.t}); \text{Beh}(\text{Store}(r, l)_{p.t}) = \\ & inv_p(U1, U2)^\top; \text{Beh}(\mathbf{Store}(r, l)_{p.t}); \text{Beh}(\text{Store}(r, l)_{p.t}); inv_p(U1 \cup \{l\}, U2 \cup \{l\})_\perp \end{aligned} \quad \square$$

Lemma 5.5 If $l \in U2$ then

$$\begin{aligned} & inv_p(U1, U2)^\top; \text{Beh}(\mathbf{Load}(r, l)_{p.t}); \text{Beh}(\text{Load}(r, l)_{p.t}) = \\ & inv_p(U1, U2)^\top; \text{Beh}(\mathbf{Load}(r, l)_{p.t}); \text{Beh}(\text{Load}(r, l)_{p.t}); inv_p(U1, U2)_\perp \end{aligned} \quad \square$$

Lemma 5.6

$$\begin{aligned} & inv_p(U1, U2)^\top; \text{Beh}(\mathbf{Rinstr}_{p.t}); \text{Beh}(\text{Rinstr}_{p.t}) = \\ & inv_p(U1, U2)^\top; \text{Beh}(\mathbf{Rinstr}_{p.t}); \text{Beh}(\text{Rinstr}_{p.t}); inv_p(U1, U2)_\perp \end{aligned} \quad \square$$

6 Regular trace

This section will explore some conditions on sequences of machine instructions such that their execution can preserve the link invariant inv_p .

Definition 6.1

Let $tr = \langle \sigma_0, instr_1, \sigma_1, instr_2, \dots, instr_n, \sigma_n \rangle$ be a trace in the new machine. A load instruction $instr_k = \text{Load}(r, l)_{p.t}$ is said to be *safely-loaded* if

- either it is preceded by a validate or store instruction on that location; i.e.,

$$\exists j < k, \exists \tau \in T_p \bullet instr_j \in \{\text{Store}(-, l)_{p.\tau}, \text{Validate}(l)_{p.\tau}\}$$

- or $C_p[l]$ is not clean on the state σ_{k-1} .

tr is *safely-loaded* on a location l if its first load instruction on that location (when it exists) is safely-loaded.

tr is *store-guarded* on a location l if its first instruction on that location is a store instruction. $\square$

Theorem 6.2

Let $tr = \langle \sigma_0, instr_1, \sigma_1, \dots, instr_n, \sigma_n \rangle$ be a trace in the new machine

and $abstr = (a\sigma_0, \mathbf{instr}_1, a\sigma_1, \dots, \mathbf{instr}_n, a\sigma_n)$ a trace with the same sequence of machine instructions in the old machine.

Assume that the relation $inv_p(U1, U2)$ holds between σ_0 and $a\sigma_0$ with

$$(Loc \setminus \mathbf{SG}(tr)) \subseteq U1$$

and

$$(Loc \setminus \mathbf{SL}(tr)) \subseteq U2$$

where

$$\mathbf{SG}(tr) \stackrel{\text{def}}{=} \{l \in Loc \mid tr \text{ is store-guarded on } l\}$$

$$\mathbf{SL}(tr) \stackrel{\text{def}}{=} \{l \in Loc \mid tr \text{ is safely-loaded on } l\}$$

Then the relation $inv_p(U1, U2)$ holds between σ_m and $a\sigma_m$ for all $1 \leq m \leq n$.

Proof: We proceed by induction on n . The conclusion is trivial when $n = 0$. Now assume $n \geq 1$, and let

$$tr' = \langle \sigma_1, instr_2, \sigma_2, \dots, instr_n, \sigma_n \rangle$$

The remainder of the proof is based on case analysis:

Case 1: $instr_1$ is a load instruction on a location l .

Case 1.1: $l \notin \mathbf{SL}(tr)$. In this case we conclude

(1.1.a): $\mathbf{SL}(tr) \subseteq \mathbf{SL}(tr')$.

(1.1.b): $\mathbf{SG}(tr) \subseteq \mathbf{SG}(tr')$.

From the assumption it follows that

$$l \in U2$$

From Lemma 5.5 we conclude that σ_1 and $a\sigma_1$ are also related by $inv_p(U1, U2)$ with

$$(Loc \setminus \mathbf{SG}(tr')) \subseteq (Loc \setminus \mathbf{SG}(tr)) \subseteq U1$$

and

$$(Loc \setminus \mathbf{SL}(tr')) \subseteq (Loc \setminus \mathbf{SL}(tr)) \subseteq U2$$

Case 1.2 $l \in \mathbf{SL}(tr)$. It is evident that $l \notin \mathbf{SG}(tr)$. From the assumption on $U1$ we have

$$l \in U1$$

Since $C_p[l]$ is not clean on σ_0 we conclude that

$$wlink_p(l) = link_p(l)$$

on the state σ_0 which implies $l \in U2$ as well. In this case one has

(1.2.a): $\mathbf{SL}(tr) = \mathbf{SL}(tr')$.

(1.2.b): $\mathbf{SG}(tr) \subseteq \mathbf{SG}(tr')$.

The remainder of the proof is similar to Case 1.1.

Case 2: $instr_1$ is a flush instruction on a location l . In this case one has

(2.a): $l \notin \mathbf{SG}(tr)$.

(2.b): $\mathbf{SL}(tr) = \mathbf{SL}(tr')$

(2.c): $\mathbf{SG}(tr) \subseteq \mathbf{SG}(tr')$

From the assumption it follows that

$$l \in U1$$

From Lemma 5.2 we conclude that σ_1 and $a\sigma_1$ are also related by $inv_p(U1, U2)$ where hold

$$(Loc \setminus \mathbf{SG}(tr')) \subseteq (Loc \subseteq \mathbf{SG}(tr)) \subseteq U1$$

and

$$(Loc \setminus \mathbf{SL}(tr')) = (Loc \setminus \mathbf{SL}(tr)) \subseteq U2$$

The remaining cases can be proved in a similar way. From the fact

$$inv_p(U1 \cup V1, U2 \cup V2) \Rightarrow inv_p(U1, U2)$$

and the induction hypothesis on tr' we complete the proof $\square$

Corollary 6.3

Let $tr = \langle \sigma_0, instr_1, \sigma_1, \dots, instr_n, \sigma_n \rangle$ be a trace in the new machine and $abstr = (a\sigma_0, \mathbf{instr}_1, a\sigma_1, \dots, \mathbf{instr}_n, a\sigma_n)$ a trace in the old machine. Assume that tr is safely-loaded on all location $l \in Loc$, and none of $C_p[l]$ (for $l \in Loc$) is dirty on the initial state σ_0 and the relation $M1 = M2 \wedge \mathbf{Equal}(R)$ holds between σ_0 and $a\sigma_0$. Then the relation $\mathbf{Equal}(R)$ holds between σ_m and $a\sigma_m$ for all $1 \leq m \leq n$. $\square$

Corollary 6.4

Let $tr = \langle \sigma_0, instr_1, \sigma_1, \dots, instr_n, \sigma_n \rangle$ be a trace in the new machine and $tr1 = (a\sigma_0, \mathbf{instr}_1, a\sigma_1, \dots, \mathbf{instr}_n, a\sigma_n)$ a trace in the old machine. Assume that s is store-guarded on all location $l \in Loc$, and the relation $\mathbf{Equal}(R)$ holds between σ_0 and $a\sigma_0$. Then the relation $\mathbf{Equal}(R)$ holds between σ_m and $a\sigma_m$ for all $1 \leq m \leq n$. $\square$

We rely on the following condition to clean the state of a cache.

Definition 6.5 (Flush-tailed)

Let s be a sequential of machine instructions. For $l \in Loc$ let u be the subsequence of s consisting of all store and flush instructions on that location. s is *flush-tailed* on the location l if

$$u \neq \langle \rangle \Rightarrow \exists t \in T_p \bullet \mathbf{final}(u) = \mathbf{Flush}(l)_{p,t}$$

where $\mathbf{final}(u)$ stands for the final element of a nonempty sequence u . $\square$

Definition 6.6 (Locally regular)

Let tr be a safely-loaded trace on all locations. tr is *locally regular* if its machine instruction sequence is flush-tailed on all locations as well. $\square$

Theorem 6.7

Let $tr = \langle \sigma_0, instr_1, \sigma_1, \dots, instr_n, \sigma_n \rangle$ be a locally regular trace in the new machine, and $abstr = (a\sigma_0, \mathbf{instr}_1, a\sigma_1, \dots, \mathbf{ins}_n, a\sigma_n)$ a trace of the same sequence of machine instructions in the old machine.

Assume that none of $C_p[l]$ (for $l \in Loc$) is dirty on the initial state σ_0 , and that the relation $M1 = M2 \wedge \mathbf{Equal}(R)$ holds between σ_0 and $a\sigma_0$.

Then none of $C_p[l]$ (for $l \in Loc$) is dirty on the state σ_n ,

and the relation $M1 = M2 \wedge \mathbf{Equal}(R)$ also relates the final states σ_n to $a\sigma_n$.

Proof: From the definition of machine instructions it follows that the only instruction which can switch a cache location from a non-dirty state to dirty state is a store instruction. From the flush-tailed property of s we can ensure that the cache location will turn back to a non-dirty state. Since none of the caches is dirty in the state σ_0 , from the previous argument we conclude that this is still true in the final state σ_n .

From Theorem 6.2 it follows that $inv_p(Loc, \emptyset)$ holds between σ_n and $a\sigma_n$. From the definition of $wlink_p(l)$ we have

$$(wlink_p(l) \wedge C_p[l].state \neq dirty) \Rightarrow (M1[l] = M2[l])$$

which leads to the following conclusion:

$$(inv_p(Loc, \emptyset) \wedge \forall l \in Loc \bullet C_p[l].state \neq dirty) \Rightarrow (\mathbf{Equal}(R) \wedge M1 = M2)$$

as required. $\square$

Definition 6.8 (Regular trace)

$tr = \langle \sigma_0, instr_1, \sigma_1, \dots, instr_n, \sigma_n \rangle$ is a *regular trace* if its sequence of machine instructions can be divided into several segments $(s(1), \dots, s(m))$:

$$tr = \langle \sigma_{i(0)}, s(1), \sigma_{i(1)}, s(2), \dots, \sigma_{i(m-1)}, s(m) \rangle$$

with $i(0) = 0$ such that each segment $s(i)$ of machine instructions is wholly executed in one processor, and furthermore there exists $0 \leq j \leq m$ satisfying the following condition:

(1) $\langle \sigma_{i(k-1)}, s(k) \rangle$ is locally regular for all $k \leq j$. (2) $s(k)$ is store-guarded on all locations when $k > j$. $\square$

Theorem 6.9 (Link invariant for regular trace)

Let $tr = \langle \sigma_0, instr_1, \sigma_1, instr_2, \dots, \sigma_{n-1}, instr_n, \sigma_n \rangle$ be a regular trace in the new machine, and $abstr = \langle a\sigma_0, \mathbf{instr}_1, a\sigma_1, \mathbf{instr}_2, \dots, a\sigma_{n-1}, \mathbf{instr}_n, a\sigma_n \rangle$ be a trace of the same sequence of instructions in the old machine.

Assume that the binary relation $\mathbf{Equal}(R) \wedge M = M1$ holds between the initial states σ_0 and $a\sigma_0$, and that none of cache is dirty on the initial state σ_0 .

Then the binary relation $\mathbf{Equal}(R)$ relates σ_i to $a\sigma_i$ for all $1 \leq i \leq n$.

Proof: The proof is based on induction on the number of segments in tr . The base case is $m = 1$. If $s(1)$ is locally regular then the conclusion follows directly from Corollary 6.4. Otherwise $s(1)$ is store-guarded on all locations, and the conclusion follows from Corollary 6.5.

Now assume that $m > 1$. If all segments are store-guarded, then the conclusion follows from Corollary 6.4. Otherwise assume that $s(1) = \langle instr_1, \dots, instr_k \rangle$ is locally regular, and define

$$\begin{aligned} tr' &\stackrel{def}{=} \langle \sigma_k, instr_{k+1}, \sigma_{m+1}, \dots, instr_n, \sigma_n \rangle \\ abstr' &\stackrel{def}{=} \langle \sigma_k, \mathbf{instr}_{k+1}, a\sigma_{m+1}, \dots, \mathbf{instr}_n, \sigma_n \rangle \end{aligned}$$

From Corollary 6.3 and Theorem 6.7 it follows that

1. $\mathbf{Equal}(R)$ holds between states σ_i and $a\sigma_i$ for all $1 \leq i \leq k$.
2. $M1 = M2$ holds between σ_k and $a\sigma_k$.
3. None of cache is dirty on the state σ_k .

It is clear that tr' is also a regular trace. The conclusion follows from the induction hypothesis for tr' . $\square$

7 Reordering

In this section we are going to show how any coherent trace in the new machine can be re-ordered to a regular trace tr' satisfying

$$tr \approx tr'$$

The re-ordering relation is captured by a relation **reg**, which is defined in terms of the composition of two others:

$$\mathbf{reg} \stackrel{\text{def}}{=} \mathbf{regstore}; \mathbf{regload}$$

The relation **regstore** reorganizes the store instruction in the trace, while **regload** reorganizes load instructions.

regstore is defined as an iteration

$$\mathbf{regstore} \stackrel{\text{def}}{=} (\mathbf{later})^*$$

where the construct $\mathbf{R}^*$ is defined in the standard way:

$$tr \mathbf{R}^* tr' \stackrel{\text{def}}{=} \exists n \geq 0 \bullet (tr \mathbf{R}^n tr' \wedge tr' \notin \text{domain}(\mathbf{R}))$$

The relation **later** relates a trace tr to a similar one tr' but the number of segments which are not flush-tailed is decreased. Let

$$tr = (\sigma_0, s(1), s(2), \dots, s(n))$$

where none of the caches is dirty in the state σ_0 and each segment $s(i)$ is wholly executed in one processor. Define

$$\text{Head}(tr) = \{s(k) \mid \exists j \bullet (k < j \wedge s(k) \text{ and } s(j) \text{ are executed in the same processor})\}$$

For any segment $s(m)$ let

$$L(s(m)) \stackrel{\text{def}}{=} \{l \mid s(m) \text{ is not flush-tailed on } l\}$$

For $s(m) = \langle instr_1, \dots, instr_k \rangle$ with $L(s(m)) \neq \emptyset$ we define

$$s(m)_1 = \langle instr_{n(1)}, \dots, instr_{n(i)} \rangle$$

as the longest subsequence of $s(m)$ consisting of load and store instructions on the location set $L(s(m))$ satisfying

$$\begin{aligned} \forall 1 \leq h \leq i \bullet \\ instr_{n(h)} = \text{Store}(-, l) \Rightarrow \text{Flush}(l) \notin \{instr_j \mid n(h) < j \leq k\} \wedge \\ instr_{n(h)} = \text{Load}(-, l) \Rightarrow \exists j < h \bullet instr_{n(j)} = \text{Store}(-, l) \end{aligned}$$

where the first parameters of Load and Store instructions are irrelevant, and left unspecified. Let $s(m)_2$ be the remaining sequence of instructions in $s(m)$.

Case 1: $s(m) \notin \text{Head}(tr)$ with $L(s(m)) \neq \emptyset$ such that there exists $(j \geq m)$ where the segment $s(j)$ is not store-guarded on all locations.

Let tr' be

$$\langle \sigma, s(1), \dots, s(m-1), s(m)_2, s(m+1), \dots, s(n), s(m)_1 \rangle$$

Then

$$tr \mathbf{later} tr'$$

Case 2: $s(m)$ is the last segment in $\text{Head}(tr)$ with $L(s(m)) \neq \emptyset$.

Let $s(j)$ be the earliest segment after $s(m)$ executed in the same processor as $s(m)$ satisfying the following condition:

- either there is $l \in L(s(m))$ such that $s(j)$ contains a flush instruction on that location.
- or $s(j) \notin \text{Head}(tr)$

Let tr' be

$$\langle \sigma_0, s(1), \dots, s(m-1), s(m)_2, s(m+1), \dots, s(j-1), (s(m)_1 \cdot s(j)), s(j+1), \dots, s(n) \rangle$$

Then

$$tr \text{ later } tr'$$

We give this relation the name **later** as it moves store instruction later in the trace.

Like **regstore** the relation **regload** is also defined as an iteration

$$\text{regload} \stackrel{\text{def}}{=} \text{earlier}^*$$

The relation **earlier** moves load instructions earlier in the trace.

Let

$$tr = \langle \sigma_{i(0)}, s(1), \sigma_{i(1)}, s(2), \dots, \sigma_{i(n)}, s(n) \rangle$$

Suppose that $\langle \sigma_{i(k-1)}, s(k) \rangle$ is the first subtrace with $s(k) \in \text{Head}(tr)$ which is not safely-loaded on Loc . Assume that it is executed in the processor p , and that $\text{Load}(r, l)_{p.t}$ is the first load instruction in $s(k)$ with

$$l \notin \text{SL}(\langle \sigma_{i(k-1)}, s(k) \rangle)$$

and $s(k)_2$ is the remaining sequence of instructions in $s(k)$.

Let $s(m)$ be the latest segment before $s(k)$ executed in the processor p satisfying the following condition:

- either $s(m)$ contains a load, or store or validate instruction on l ;
- or $s(m)$ is the first segment in tr executed in the processor p .

Let tr' be

$$\langle \sigma_0, s(1), \dots, (s(m) \cdot \langle \text{load}(r, l)_{p.t} \rangle), s(m+1), \dots, s(k-1), s(k)_2, s(k+1), \dots, s(n) \rangle$$

Then

$$tr \text{ earlier } tr'$$

The following three lemmas describe the safety properties of **reg**. The first one states that **reg** relates a coherent trace to a regular one.

Lemma 7.1 If $tr \text{ reg } tr'$ then tr' is regular. □

The next two lemmas indicate that **reg** is a stronger relation than $\approx$.

Lemma 7.2 If $tr \text{ regload } tr'$ then $tr \approx tr'$. □

Lemma 7.3 If $tr \text{ store } tr'$ then $tr \approx tr'$. □

The following lemma deals with the liveness of **reg**.

Lemma 7.4 $tr \in \text{domain}(\text{reg})$ for any coherent trace tr .

Proof: We exploit the analogy with sequential programs by giving bound functions on the bodies of the two loops `regstore` and `regload`. The remaining proof obligations are trivial.

For `later` we define

$$\text{bound1}(tr) \stackrel{\text{def}}{=} \sharp \text{Seg}(tr) + \sharp \text{Lbag}(tr)$$

where $\text{Seg}(tr)$ stands for the set

$$\{j \mid s(j) \notin \text{Head}(tr) \wedge L(s(j)) \neq \emptyset \wedge \exists k \geq j \bullet s(k) \text{ is not } \textit{store-guarded} \text{ on } \textit{Loc}\}$$

and $\text{Lbag}(tr)$ represents the bag

$$\text{bag}\{l \mid \exists i \bullet s(i) \in \text{Head}(tr) \wedge l \in L(s(i))\}$$

For Case 1 of the definition of `later` it is easy to see

$$\begin{aligned} \text{Seg}(tr') &\subset \text{Seg}(tr) \\ \text{Lbag}(tr') &= \text{Lbag}(tr) \end{aligned}$$

For Case 2 of the definition of `later` we have

$$\begin{aligned} \text{Seg}(tr') &\subseteq \text{Seg}(tr) \\ \text{Lbag}(tr') &\subset \text{Lbag}(tr) \end{aligned}$$

In both cases we have

$$\text{bound1}(tr') \leq \text{bound1}(tr) - 1$$

Clearly, if $\text{bound1}(tr) = 0$, then $tr \notin \text{domain}(\text{later})$.

Similarly, for `earlier` we define

$$\text{bound2}(tr) \stackrel{\text{def}}{=} \sharp \text{Instrbag}(tr)$$

where $\text{Instrbag}(tr)$ represents the bag

$$\text{bag}\{\text{Load}(-, l) \mid \exists k \bullet \text{Load}(-, l) \text{ is not } \textit{safely-loaded} \text{ in } \langle \sigma_{i(k-1)}, s(k) \rangle\}$$

It is also clear that

$$\text{bound2}(tr') \leq \text{bound2}(tr) - 1$$

if $tr \text{ earlier } tr'$. Also if $\text{bound2}(tr) = 0$ we have $tr \notin \text{domain}(\text{earlier})$. □

Now comes the main theorem of this paper:

Theorem 7.5 (Local Refinement)

If $\text{Equal}(R) \wedge M1 = M2$ holds between σ_0 and $a\sigma_0$, and none of cache is dirty on the initial state σ_0 , then

$$\text{Localtraces}(M, \sigma_0) \subseteq \text{Localtraces}(AM, a\sigma_0)$$

Proof: From the previous lemmas it follows that a coherent trace $tr = (\sigma_0, s)$ can be replaced by a regular trace $tr' = (\sigma_0, s')$ satisfying

$$\forall p, \forall t \in T_p \bullet \mathbf{localtrace}_{p,t}(tr) = \mathbf{localtrace}(tr')_{p,t}$$

Define $abstr \stackrel{def}{=} (a\sigma_0, s')$. From Theorem 6.1 we obtain

$$\forall p \in P, \forall t \in T_p \bullet \mathbf{localtrace}_{p,t}(tr') = \mathbf{localtrace}_{p,t}(abstr)$$

as required. $\square$

The memory system without caches is not a local refinement of the system with caches. Consider the coherent sequence of machine instructions

$$s = \langle Store(r, l)_{p,t}, Validate(\hat{r}, l)_{q,\tau}, Load(\hat{r}, l)_{q,\tau} \rangle$$

Assume that $p \neq q$ and initially

$$R1_{p,t}[r] \neq R1_{q,\tau}[\hat{r}]$$

The local trace on the thread $q.\tau$ produced by the execution of s in the old machine has different state, as $R1_{q,\tau}[\hat{r}] \neq R1_{p,t}[r]$. But there is no corresponding trace in the new machine which will yield such a local trace for the thread $q.\tau$.

References

- [1] M. Abadi and L. Lamport. The existence of refinement mapping. *Theoretical Computer Science*, 82(2): 253–284, 1991.
- [2] R. J. R. Back. A calculus of refinements for program derivation. *Acta Informatica*, 25: 593–624, 1988.
- [3] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.
- [4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [5] J. Jacob. On Shared Systems. PhD thesis, Programming Research Group, Oxford University, 1988.
- [6] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1986.
- [7] S. S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4): 325–342, 1984.
- [8] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Semantics*, 5(2): 190–222, 1983.
- [9] A. McIsaac. A coherent memory model. Technical Report. SGS-Thomson Microelectronics, 1994.
- [10] C. C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.