I/O Automata for Caching Scheme Described by Arvind

Victor Luchangco

May 29, 1997

Abstract

This paper gives the I/O automata analogues to the rewriting systems that Arvind presented on May 16, 1997.

1 Introduction

This paper describes a simple hierarchical caching scheme, one that is general enough to capture the essence of most simple caching schemes. It is derived from the rewriting systems that Arvind presented in his talk at LCS on May 16, 1997. The main difference between these automata and their analogous rewriting system is that the processor is not modelled explicitly, but rather it is left for a separate automaton with the appropriate interface. This means that the load and store commands of the processor are not atomic, but rather they must be split into a request and response. This is necessary so that the processor and memory states can be separated, as is natural for I/O automata. This is more realistic, as a processor would really experience some delay, but simply blocks until it receives a response. (We could model the processor commands as atomic if we wished, but this would require us to write the composition of the processors and memory as a single automaton. We won't do this for now.)

2 The Memory Interface

First we describe the interface (signature) for the memory. We assume that there is a set $\mathcal{A}$ of addresses and a set $\mathcal{V}$ of values that can be stored at each address. The processors are indexed by the set $\mathcal{P}$. Processors may issue load and store commands to the memory, and they are assumed to block until the memory responds. (A well-formedness requirement, but we will ignore this for now.)

Thus the memory has two input and two output actions parameterized by the processor, and possibly an address and value: $\textit{load-request}_p(a)$, $\textit{load-response}_p(a, v)$, $\textit{store-request}_p(a, v)$, $\textit{store-response}_p$. These form the interface for all the automata that follow.

3 Sequential Consistency

Actually, the memory that Arvind desribed is atomic memory (and even stronger because of the instantaneous responses). Each operation gets performed atomically at some time between the

request and response. We keep track of the actions that need to be done in a set *pending* of actions. (This is abstract, and in practice, these would probably reside in some input/output buffer.)

Atomic Memory (parameterized by $\mathcal{A}$, $\mathcal{V}$, $\mathcal{P}$)

State

$M : \mathcal{A} \to \mathcal{V}$, initially arbitrary.
pending, a set of interface actions, initially empty.

Actions

Input $load\text{-}request_p(a)$
Eff: $pending \leftarrow pending \cup \{load\text{-}request_p(a)\}$

Internal $load\text{-}perform_p(a)$
Pre: $load\text{-}request_p(a) \in pending$
Eff: $pending \leftarrow pending - \{load\text{-}request_p(a)\}$
$pending \leftarrow pending \cup \{load\text{-}response_p(a, M(a))\}$

Output $load\text{-}response_p(a, v)$
Pre: $load\text{-}response_p(a, v) \in pending$
Eff: $pending \leftarrow pending - \{load\text{-}response_p(a, v)\}$

Input $store\text{-}request_p(a, v)$
Eff: $pending \leftarrow pending \cup \{store\text{-}request_p(a, v)\}$

Internal $store\text{-}perform_p(a, v)$
Pre: $store\text{-}request_p(a, v) \in pending$
Eff: $pending \leftarrow pending - \{store\text{-}request_p(a, v)\}$
$M(a) \leftarrow v$
$pending \leftarrow pending \cup \{store\text{-}response_p\}$

Output $store\text{-}response_p$
Pre: $store\text{-}response_p \in pending$
Eff: $pending \leftarrow pending - \{store\text{-}response_p\}$

4 Caching

We assume we have a hierarchy of caches. We cache has an identifier, and the set of identifiers is $\mathcal{I}$. For consistency, we also give the main memory an identifier, 0, and define $\mathcal{I}_0$ to be $\mathcal{I} \cup \{0\}$. The caches are arranged in a tree rooted at 0 specified by $parent : \mathcal{I} \to \mathcal{I}_0$. (Obviously, *parent* will have to satisfy certain properties that we will need in the proof eventually. But these should follow from *parent* defining a tree, with 0 as the root.) Each processor $p \in \mathcal{P}$ will access a particular cache (or the memory directly) specified by $mem : \mathcal{P} \to \mathcal{I}_0$. Each entry in any cache (or the memory) has some additional information indicating whether it is an exclusive or shared copy, and if it has given the value to any of its children.

This is more general than what Arvind allows in his rewriting rule, since he requires that each processor have its own cache, not shared with any other processor. It gets all its values through this cache. His system could be extended by allowing execution units to be mixtures of processors and systems:

```
S -> (MU, EG)
EG -> E || E|EG
E -> P || S
```

Is there some reason why this was not allowed in the first place?

Hierarchical Caching (parameterized by $\mathcal{A}$, $\mathcal{V}$, $\mathcal{P}$, $\mathcal{I}$, *parent*, and *mem*)[1]

State

$M_0 : \mathcal{A} \to \mathcal{V}$, initially arbitrary.
$M_i : \mathcal{A} \to \mathcal{V}_\perp$ for $i \in \mathcal{I}$, initially all $\perp$.
$CS_0 : \mathcal{A} \to \{\texttt{Ex}, \texttt{Sh}\}$, initially (and always) Ex. (Arvind's Cstate)
$HS_0 : \mathcal{A} \to (\{(\texttt{R}, S) : S \subseteq \mathcal{I}\} \cup \{(\texttt{W}, i) : i \in \mathcal{I}\})$, initially $(\texttt{R}, \emptyset)$. (Arvind's Hstate)
$CS_i : \mathcal{A} \to \{\texttt{Ex}, \texttt{Sh}\}_\perp$ for $i \in \mathcal{I}$, initially all $\perp$.
$HS_i : \mathcal{A} \to ((\{(\texttt{R}, S) : S \subseteq \mathcal{I}\} \cup \{(\texttt{W}, i) : i \in \mathcal{I}\}))_\perp$ for $i \in \mathcal{I}$, initially all $\perp$.
pending, a set of interface actions, initially empty.

Actions

Input $\textit{load-request}_p(a)$
Eff: $\textit{pending} \leftarrow \textit{pending} \cup \{\textit{load-request}_p(a)\}$

Internal $\textit{load-perform}_p(a)$ $\quad (i = \textit{mem}(p),\ S)$
Pre: $\textit{load-request}_p(a) \in \textit{pending}$
$HS_i(a) = (\texttt{R}, S)$
Eff: $\textit{pending} \leftarrow \textit{pending} - \{\textit{load-request}_p(a)\}$
$\textit{pending} \leftarrow \textit{pending} \cup \{\textit{load-response}_p(a, M_i(a))\}$

Output $\textit{load-response}_p(a, v)$
Pre: $\textit{load-response}_p(a, v) \in \textit{pending}$
Eff: $\textit{pending} \leftarrow \textit{pending} - \{\textit{load-response}_p(a, v)\}$

Input $\textit{store-request}_p(a, v)$
Eff: $\textit{pending} \leftarrow \textit{pending} \cup \{\textit{store-request}_p(a, v)\}$

Internal $\textit{store-perform}_p(a, v)$ $\quad (i = \textit{mem}(p))$
Pre: $\textit{store-request}_p(a, v) \in \textit{pending}$
$CS_{\textit{mem}(p)}(a) = \texttt{Ex}$
$HS_{\textit{mem}(p)}(a) = (\texttt{R}, \emptyset)$
Eff: $\textit{pending} \leftarrow \textit{pending} - \{\textit{store-request}_p(a, v)\}$
$M_{\textit{mem}(p)}(a) \leftarrow v$
$\textit{pending} \leftarrow \textit{pending} \cup \{\textit{store-response}_p\}$

Output $\textit{store-response}_p$
Pre: $\textit{store-response}_p \in \textit{pending}$
Eff: $\textit{pending} \leftarrow \textit{pending} - \{\textit{store-response}_p\}$

Internal $\textit{rcache}(i, i', a)$ $\quad (S)$
Pre: $\textit{parent}(i') = i$
$HS_i(a) = (\texttt{R}, S)$
$i' \notin S$
Eff: $HS_i(a) \leftarrow (\texttt{R}, S \cup \{i'\})$
$CS_{i'}(a) \leftarrow \texttt{Sh}$
$HS_{i'}(a) \leftarrow (\texttt{R}, \emptyset)$
$M_{i'}(a) \leftarrow M_i(a)$

Internal $\textit{wcache}(i, i', a)$
Pre: $\textit{parent}(i') = i$
$CS_i(a) = \texttt{Ex}$
$HS_i(a) = (\texttt{R}, \emptyset)$
Eff: $HS_i(a) \leftarrow (\texttt{W}, i')$
$CS_{i'}(a) \leftarrow \texttt{Ex}$
$HS_{i'}(a) \leftarrow (\texttt{R}, \emptyset)$
$M_{i'}(a) \leftarrow M_i(a)$

Internal $\textit{writeback}(i, i', a)$ $\quad (S)$
Pre: $\textit{parent}(i') = i$
$CS_{i'}(a) = \texttt{Ex}$
$HS_{i'}(a) = (\texttt{R}, S)$
$[CS_i(a) = \texttt{Ex}]$
$[HS_i(a) = (\texttt{W}, i')]$
Eff: $CS_{i'}(a) \leftarrow \texttt{Sh}$
$HS_i(a) \leftarrow (\texttt{R}, \{i'\})$
$M_i(a) \leftarrow M_{i'}(a)$

Internal $\textit{invalidate}(i, i', a)$ $\quad (S)$
Pre: $\textit{parent}(i') = i$
$CS_{i'}(a) = \texttt{Sh}$
$HS_{i'}(a) = (\texttt{R}, \emptyset))$
$[HS_i(a) = (\texttt{R}, S)$ with $i' \in S]$
Eff: $HS_i(a) \leftarrow (\texttt{R}, S - \{i'\})$
$CS_{i'}(a) \leftarrow \perp$
$HS_{i'}(a) \leftarrow \perp$
$M_{i'}(a) \leftarrow \perp$

[1] Clauses in the precondition that appear in brackets do not really constrain action. They are implied by invariants satisfied by the equivalent automaton without those clauses, or else they simply bind values used in the action clauses.

To prove that the hierarchical caches implement the atomic memory, we need several invariants such as:

Invariant 4.1 For all $a \in \mathcal{A}$, $i, i' \in \mathcal{I}$:

- $HS_i(a) = (\mathtt{W}, i') \implies CS_i(a) = \mathtt{Ex}$
- $CS_0(a) = \mathtt{Ex}$ for all $a \in \mathcal{A}$.
- If $CS_i(a) = \mathtt{Ex}$ then $HS_{parent(i)}(a) = (\mathtt{W}, i)$.

These imply that for each address, there is a single branch of the the tree, starting from the root, that holds exclusive copies of the value at that address. (It may be only the root.) The "abstract" value at that address is then the value at the end of that branch. This yields the abstract atomic memory state. (Obviously, this should be made more formal, but I think it may involve being more explicit about the restrictions on *parent*, and I don't think it is necessary for the intuition.)

Invariant 4.2 For all $a \in \mathcal{A}$, $i \in \mathcal{I}$:

- $M_i(a) = \perp \iff CS_i(a) = \perp \iff HS_i(a) = \perp$
- If $CS_i(a) = \mathtt{Sh}$ then $M_i(a) = M_{parent(i)}(a)$.
- If $CS_i(a) = \mathtt{Sh}$ then $HS_{parent(i)}(a) = (\mathtt{R}, S))$ where $i \in S$.

We can also prove a stronger invariant which shows that the *HS* state component is redundant:

Invariant 4.3 For all $a \in \mathcal{A}$, $i \in \mathcal{I}$:

- $CS_i(a) = \mathtt{Sh} \iff HS_{parent(i)}(a) = (\mathtt{R}, S)$ with $i \in S$.
- $CS_i(a) = \mathtt{Ex} \iff HS_{parent(i)}(a) = (\mathtt{W}, i)$

Actions are simulated by actions with corresponding names. The extra actions in the implementation correspond to empty execution fragments.

5 Eliminating False Atomicity

One problem with the scheme described above is that the state of the child and the parent must be updated simultaneously, which is not possible in a real system. But this is not really necessary. It is sufficient for a message to be sent between them. If we are not concerned with the particular interconnect, as long as it is completely reliable, we model this with a single *msgs* set. (We could also have separate channels.)

Each of the $rcache(\cdot,\cdot,\cdot)$, $wcache(\cdot,\cdot,\cdot)$, $writeback(\cdot,\cdot,\cdot)$, and $invalidate(\cdot,\cdot,\cdot)$ actions are replaced with two actions corresponding to the changes at each of the caches involved. (The $\cdot$ indicates that any value is acceptable.) Here are the changes to the automaton:

Additional State Variable

msgs: a set of messages, initially empty.

Internal *rcache-parent*(i, i', a) (S)
Pre: $parent(i') = i$
$HS_i(a) = (\mathtt{R}, S)$
$i' \notin S$
Eff: $HS_i(a) \leftarrow (\mathtt{R}, S \cup \{i'\})$
$msgs \leftarrow msgs \cup \{\langle \mathtt{rcache}, i, i', a, M_i(a)\rangle\}$

Internal *rcache-child*(i, i', a, v)
Pre: $[parent(i') = i]$
$\langle \mathtt{rcache}, i, i', a, v\rangle \in msgs$
Eff: $msgs \leftarrow msgs - \{\langle \mathtt{rcache}, i, i', a, v\rangle\}$
$CS_{i'}(a) \leftarrow \mathtt{Sh}$
$HS_{i'}(a) \leftarrow (\mathtt{R}, \emptyset)$
$M_{i'}(a) \leftarrow v$

Internal *wcache-parent*(i, i', a)
Pre: $parent(i') = i$
$CS_i(a) = \mathtt{Ex}$
$HS_i(a) = (\mathtt{R}, \emptyset)$
Eff: $HS_i(a) \leftarrow (\mathtt{W}, i')$
$msgs \leftarrow msgs \cup \{\langle \mathtt{wcache}, i, i', a, M_i(a)\rangle\}$

Internal *wcache-child*(i, i', a, v)
Pre: $[parent(i') = i]$
$\langle \mathtt{wcache}, i, i', a, v\rangle \in msgs$
Eff: $msgs \leftarrow msgs - \{\langle \mathtt{wcache}, i, i', a, v\rangle\}$
$CS_{i'}(a) \leftarrow \mathtt{Ex}$
$HS_{i'}(a) \leftarrow (\mathtt{R}, \emptyset)$
$M_{i'}(a) \leftarrow v$

Internal *writeback-child*(i, i', a) (S)
Pre: $parent(i') = i$
$CS_{i'}(a) = \mathtt{Ex}$
$HS_{i'}(a) \leftarrow (\mathtt{R}, S)$
$[CS_i(a) = \mathtt{Ex}]$
$[HS_i(a) = (\mathtt{W}, i')]$
Eff: $CS_{i'}(a) \leftarrow \mathtt{Sh}$
$msgs \leftarrow msgs \cup \{\langle \mathtt{writeback}, i, i', a, M_{i'}(a)\rangle\}$

Internal *writeback-parent*(i, i', a, v)
Pre: $[parent(i') = i]$
$\langle \mathtt{writeback}, i, i', a, v\rangle \in msgs$
Eff: $msgs \leftarrow msgs - \{\langle \mathtt{writeback}, i, i', a, v\rangle\}$
$HS_i(a) \leftarrow (\mathtt{R}, \{i'\})$
$M_i(a) \leftarrow v$

Internal *invalidate-child*(i, i', a)
Pre: $parent(i') = i$
$CS_{i'}(a) = \mathtt{Sh}$
$HS_{i'}(a) = (\mathtt{R}, \emptyset)$
Eff: $CS_{i'}(a) \leftarrow \perp$
$HS_{i'}(a) \leftarrow \perp$
$M_{i'}(a) \leftarrow \perp$
$msgs \leftarrow msgs \cup \{\langle \mathtt{invalidate}, i, i', a\rangle\}$

Internal *invalidate-parent*(i, i', a) (S)
Pre: $[parent(i') = i]$
$\langle \mathtt{invalidate}, i, i', a\rangle \in msgs$
$[HS_i(a) = (\mathtt{R}, S)$ with $i' \in S]$
Eff: $msgs \leftarrow msgs - \{\langle \mathtt{invalidate}, i, i', a\rangle\}$
$HS_i(a) \leftarrow (\mathtt{R}, S - \{i'\})$

This allows messages to be delivered out of order. The only requirement is that they be delivered exactly once. I think this is sufficient, but I haven't checked this carefully. Here are the pieces I think are needed for the proof.

These invariants, which I think need to be proved together, are probably necessary:

- If $\langle \mathtt{rcache}, i, i', a, v\rangle \in msgs$ then $i = parent(i')$, $M_i(a) = v$, and $HS_i(a) = (\mathtt{R}, S)$ where $i' \in S$.
- If $\langle \mathtt{wcache}, i, i', a, v\rangle \in msgs$ then $i = parent(i')$, $M_i(a) = v$, $CS_i(a) = \mathtt{Ex}$, and $HS_i(a) = (\mathtt{W}, i)$.
- If $\langle \mathtt{writeback}, i, i', a, v\rangle \in msgs$ then $i = parent(i')$, $M_{i'}(a) = v$, $CS_{i'}(a) = \mathtt{Sh}$, $CS_i(a) = \mathtt{Ex}$, and $HS_i(a) = (\mathtt{W}, i')$.
- If $\langle \mathtt{invalidate}, i, i', a\rangle \in msgs$ then $i = parent(i')$, $M_{i'}(a) = \perp$, and $HS_i(a) = (\mathtt{R}, S)$ where $i' \in S$.
- At most one of $\langle \mathtt{rcache}, i, i', a, \cdot\rangle$, $\langle \mathtt{wcache}, i, i', a, \cdot\rangle$, $\langle \mathtt{writeback}, i, i', a, \cdot\rangle$, and $\langle \mathtt{invalidate}, i, i', a\rangle$ is in $msgs$.

The action correspondence then would be that the $rcache(\cdot,\cdot,\cdot)$, $wcache(\cdot,\cdot,\cdot)$, $writeback(\cdot,\cdot,\cdot)$, and $invalidate(\cdot,\cdot,\cdot)$ actions are all done with the children. The other action at the parent corresponds to the null execution fragment. The state correspondence can be derived from this.