On the realizability of hardware microthreading. Revisiting the general-purpose processor interface: consequences and challenges
Poss, R.C.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (http://dare.uva.nl)
Chapter 7

Disentangling memory and synchronization for shared state and communication

The important thing in science is not so much to obtain new facts as to discover new ways of thinking about them.

William L. Bragg

Abstract

In this chapter, we highlight the opportunity to synchronize program behavior by means other than memory. Then we show how to expose synchronization and consistency semantics in the proposed interface language from chapter 6.

Contents

7.1 Introduction ................................................................. 116
7.2 Extending C with memory-independent synchronization .................. 119
7.3 Extending C with multi-thread consistency .................................. 123
7.4 Examples of incorrect synchronization ....................................... 127
7.5 Pitfall: atomicity anomalies ................................................ 130
7.6 Relationship with the latest C and C++ specifications ...................... 131
7.7 Position of the model from the programmer’s perspective ................. 132
7.8 Other related work .......................................................... 132
Summary ........................................................................... 135
Side note 7.1: About implicit communication.

From an information-theoretic perspective, communication exists as soon as information is exchanged between a sender and a recipient. Communication requires that the parties share a commonality a priori, usually shared semantics for the language of messages; however it typically does not require a third party to participate in the exchange. We recognize two extended forms of communication which actively rely on a third party. The first is *proxied communication*, where the third party acts as a relay for the signal without sharing the commonality between sender and recipient. Encrypted network channels are examples: the intermediary network nodes that carry the encrypted bits do not know the keys necessary to provide meaning to the messages exchanged. The second is *implicit communication*, where a third party is the shared commonality. For example, the meaning of the message “the meaning of this message is to be read from the first book published tomorrow” cannot be derived from the text of the message itself; instead it must be fetched from the external shared commonality, in this case the first book published tomorrow. In this chapter, we are considering implicit communication that occurs between processors using the information in shared storage devices as shared commonalities.

7.1 Introduction

Ever since the advent of architectures where multiple processors, or virtualizations thereof via threads, can access a single storage device via their interconnect, computer users at all levels of abstraction have attempted to exploit the opportunity for implicit communication which exists between stores issued by one processor and loads issued by another, regardless of whether the processors may be connected to each other via dedicated communication channels. To clarify, we provide our definition of “implicit communication” in side note 7.1.

While investigating diverse applications and languages, we were able to reduce all uses of implicit communication to only two essential patterns. The first is *persistent shared state* in multi-agent computations, e.g. the service provided by a shared file system. This is characterized by the ability to observe the state even when no agent is active, and the ability of the agents to access the state without knowing a priori which other agent last accessed it. The other is *implicit communication channels*, i.e. the ability to communicate an arbitrarily shaped data structure by using only a reference to it. We distinguish these two services here to ground the following observations:

- shared state and communication channels are abstract concepts which do not require a physically shared storage to be implemented. For example, if shared state can be represented logically, it can be transferred between the physical locations where it is used without the need for a central repository; this is the principle behind REpresentational State Transfer (REST) [Fie00]. Communication channels, in turn, can be implemented by serializing the arbitrarily shaped data over dedicated links between processors, separate from the memory system;
- each of these two concepts requires different kinds of support from the substrate hardware with regards to synchronization. To implement shared state, the substrate must provide mechanisms for transactions, to provide an illusion of atomicity of multiple updates by one agent from the perspective of other agents. To implement channels, the substrate must provide mechanisms to guarantee order of communication events and signal the availability of data or the readiness of the agents to participate in a communication activity.

Despite this distinction, the conceptual model of general-purpose processors has been historically simplified to the point that the memory interface has become the only communication interface between the processor and its outside world. There are three consequences to this:
7.1. INTRODUCTION

- this simplification has allowed programming language designers to conflate the concept of side effect with the concept of memory operation, whereas previously side effects would also include actions on other physical links than those connecting the processors to memory (e.g. dedicated I/O links);
- the simplification has diluted the need in programming languages to expose the shape and purpose of data accessed concurrently by different processors to the underlying platform: all memory addresses are considered symmetrical with regards to their reachability, and programs do no longer explicitly distinguish which addresses participate in a multi-processor activity from those that don’t;
- it has created a general understanding and expectation from software implementers that the memory system is “the great coordinator” between processors, and that the only synchronization primitives available in programs are one-size-fits-all memory-centric mechanisms like “atomic fetch-and-add” or “atomic compare and swap” that can be used to implement both shared state and communication channels.

This conflation of concepts is worrying. By erasing the conceptual nuance between state, communication and memory, it erases the fact that shared state and communication are actually built upon fundamentally separate synchronization mechanisms. The innovation space where architects could introduce other features, next to memory, with different and more efficient characteristics, is thereby reduced. Moreover, it creates pressure on the providers of memory services, i.e. memory architects and system integrators, to provide increasingly comprehensive mechanisms for synchronization. Mechanisms to control the ordering and signalling required by logical communication channels, in particular, increase the overall logic and energy cost of the memory system as a whole even for those uses of memory not concerned by multi-processor interactions. This pressure is the origin of discussions about “memory consistency”, which we revisit below in section 7.1.1.

Hopefully, future architectural innovations will safeguard and exploit the conceptual nuances identified above in general-purpose computers. We can recognize an example step in this direction with “scratchpad memories,” [BSL+02] where small, dedicated and non-synchronizing memories are placed next to cores on Multi-Processor Systems-on-Chip (MPSoCs) with other mechanisms like hardware mailboxes [FH76] to organize synchronization and communication between cores.

The proposed architectural innovation we described in part I also contributes to this vision. As we explain below in section 7.1.2, it negotiates synchronization between processors via mechanisms distinct from memory. We show a possible abstraction of these semantics in a system-level language interface in sections 7.2 and 7.3, which we apply in sections 7.4 and 7.5. We then discuss related work in sections 7.6 and 7.8.

7.1.1 The pressure on memory: from simple to less simple

The original definition of “memory” is a physical device which provides some guarantee that the values sent via store operations will be read by subsequent load operations. It is fundamental because it is one of the two halves of the general computer, the other half being the processor, which makes the computing system analogous to a Turing machine, and thus able to compute (cf. section 1.2.1). It is also quite simple and inflexible: the memory must guarantee that loads from one processor will return the value most recently stored by that processor, no matter what; otherwise, computation is not possible. This inflexibility holds for a single-threaded processor, or in a concurrent system for its simplest virtualization, the single thread.
Once multiple processors, or threads, are connected together, a choice then exists in a design: either each of them can only access its own memory isolated from other processors, or they can share storage devices. As outlined above, the purpose of such sharing is a desire of software implementors to enable implicit communication between stores issued by one processor and loads from another, in a quest for more simplicity in programming models. Without this sharing, processors can typically only communicate via explicit point-to-point messages (“message passing”).

The first implementations were simple: each “memory” was a single physical device, and the links between processors and memory simply preserved the ordering of the operations as issued by the processors. The use of such an architecture in programs is not too complicated: stores by any processor are visible in the same order by all other processors sharing the same memory. This guarantee is called “sequential consistency,” and was long the de facto machine abstraction for multi-programming.

What happened then was a back-pressure from hardware architects.

One force was the introduction of caching, especially separate caches for separate processors connected to the same backing store. As storage capacity grows, latency of access grows too. When its sharing factor increases (more processors per storage unit), contention occurs and the latency of access grows more. Caching is then desirable to provide the illusion of uniform, lower latency to access larger storage, given some locality of access. However, cache protocols, in particular coherency protocols responsible for propagating cache updates between multiple processors, are expensive to implement if they must provide the illusion of sequential consistency. It is much cheaper for the architect to “break the contract” in part, for example by saying that a store by a processor will only affect its local cache and that further loads by other processors will not see the update until the first processor explicitly flushes its changes. This is simpler to implement and incurs less pressure on the cache-memory network, but it does not respect the expectation of global visibility in the sequential model. As such, it makes the implementation of shared state and communication channels over memory, the two applications identified above, somewhat more difficult.

The other force was the introduction of reordering across network links, for example for congestion control or extra ILP within processors. Even in the absence of caches and using a single memory, reordering can cause stores from two processors to arrive in a different order than they were emitted. For example, a thread A can perform “a := 10; b := 20; barrier” and a thread B can perform “barrier; a := 30; b := 40” and the memory ends up containing “a=30” and “b=20”. Again, this contradicts an expectation of the sequential model, this time the preservation of store order. This constitutes another obstacle to the implementation of shared state and communication channels.

In addition, both techniques (caching and reordering) can combine, creating even more exotic situations. The status quo at the time of this writing is a trade-off, where the architect combines the cheaper consistency protocols with extra control over the memory system, namely barriers to flush stores globally, acquire-release locks to delineate a subset of stores that must be implicitly propagated globally, and reordering fences which disable reordering selectively, usable by programs to declare their intent to obtain the illusion of sequential consistency for identified sequences of loads and stores. We should deem this state of affairs still unsatisfactory, as these primitives do not scope synchronization: when updates must be propagated, the memory system must assume that they must become visible everywhere. Even though application knowledge may exist to inform that the data is needed only in a specific area of the system, and thus create an opportunity to save synchronization latency...
and energy, there is currently no mechanism to communicate this knowledge to the hardware in general-purpose designs.

To summarize, issues of consistency, which are at the heart of the discussion about the programmable semantics of multi-processor systems where memory coordinates all interactions, is the result of the collision between software implementers, who strive for conceptual simplicity, and hardware architects, who strive for simplicity in the implementation towards more efficiency and scalability.

7.1.2 Synchronization and memory in the proposed architecture

The innovation from part I is focused on the internal organization of cores for latency tolerance, and the efficient coordination of work distribution across multiple cores. It proposes that threads can synchronize using dedicated dataflow synchronizers in hardware within cores, and uses a dedicated delegation network combined with a distributed bulk creation and synchronization protocol implemented in hardware.

What interests us here is that this innovation is fundamentally agnostic of how the cores are connected to storage, i.e. it does not mandate a specific memory architecture nor does it propose specific mechanisms to coordinate access to data. Instead, it proposes the same synchronization mechanisms regardless of whether a shared memory system is available, and regardless of what consistency semantics are available in the shared memory, if any.

Of course, this approach does not preclude a chip design using a strongly consistent memory system in combination with the proposed core design and inter-core synchronization subsystem. For example, the FPGA-based implementation introduced in section 4.7 connects the proposed core design to a traditional memory bus. However, at the same time the approach strongly suggests exploring whether the hardware synchronization mechanisms, which are independent from memory, could provide a powerful multi-core programming environment when coupled with a simple, efficient, scalable but weakly coherent memory system that does not offer synchronization mechanisms.

This is the approach that we introduced previously in section 3.4.1. For example, the reference processor chip implementation that we used to define the platform from chapter 5 provides cache coherency between cores that share a single L2 cache, but updates to L2 caches are only propagated upon bulk creation and synchronization events. This means that a thread running on a core connected to an L2 cache cannot communicate or share state reliably using memory with a thread running on a core connected to another L2 cache, even though they can organize communication and synchronization using the dedicated synchronization network.

Given these circumstances, we explored whether we could abstract the semantics of our platform in an abstract machine suitable for a programming language like C. This process is especially important because most computing ecosystems rely on the abstract machine of system-level language interfaces to construct software and define its semantics, instead of using knowledge about the specific hardware platforms. The rest of this chapter documents our findings.

7.2 Extending C with memory-independent synchronization

- In this section we explain how the ordering of concurrent operations is decided from the order of execution of special synchronizing operations by programs, independently from the order of loads and stores to memory.
7.2.1 Synchronizing operations and “scheduled before”

1 We consider the execution of programs on a parallel machine, where virtualized processors, i.e. threads, enter and leave the system dynamically.
2 Each thread executes a thread program consisting of instructions, each specifying the execution of one operation at run-time. The operations correspond to an observable effect on the direct environment of the processor that executes it, for example changing the state of an I/O device, advancing an instruction counter, or issuing a load or store to a memory system.
3 We call the program order between instructions, specified by the thread program’s control flow in a programming language, the “sequenced before” order, and we denote it →. This is equivalent to the relation of the same name in [II11b, 5.1.2.3§3].
4 Certain operations synchronize with other operations performed by another thread. This relation is asymmetric: if b synchronizes with a, the execution of b does not start before a completes, but no extra information is given as to when the execution of a starts.
5 We name the partial ordering of the execution of operations at run-time “scheduled before”, we denote it ∼, and we specify:
   - “sequenced before” constrains “scheduled before”: if a and b are instructions, and ar and br are the corresponding operations at run time, then a → b ⇒ ar ∼ br.
   - all implementations of the abstract machine guarantee that the execution of a ready operation x starts a finite amount of time after all operations \{y | y ∼ x\} have completed. This guarantees progress of execution for all schedulable operations.
6 We say that two operations a and b are concurrent if a ⊈ b and b ⊈ a. An implementation may execute concurrent operations simultaneously, or choose an arbitrary scheduling order between them.
7 The definition of the special synchronizing operations below extends the ∼ relation and thus constrains scheduling further.

7.2.2 Threads and families

7.2.2.1 Threads

1 We denote the set of all logical threads ℰ.
2 For readability, we will denote henceforth “x(i)” for “operation x executed by thread i ∈ ℰ,” and → the sequence order of operations executed by i.
3 We denote begin(i) and end(i) the minimum and maximum of →, i.e. the first and last operations executed by a thread. Threads that perform no operations can be considered to execute a single pseudo-operation with no effect so that begin and end are always defined.

7.2.2.2 Families

1 ℰ is partitioned in totally ordered subsets named families.
   (the grouping of threads into families are an emergent sub-structure of ℰ caused by the use of the c operation (described below in section 7.2.3) by programs)
2 We denote F(i) ∈ ℰ the family of thread i, and < the total order within a family.
3 We denote:
   - for any family F, the first thread of F, denoted first(F), that dominates all other threads in F via <;
for any family $F$, the last thread of $F$, denoted $\text{last}(F)$, which is dominated by all other threads in $F$ via $\triangleleft$;
for any family $F$ and $t \neq \text{last}(F) \in F$, the successor of $t$ in $F$, denoted $\text{succ}(t)$, verifying $\not\exists t_2 \in F : t \triangleleft t_2 \triangleleft \text{succ}(t)$.

(the last thread in a family has no successor; in a family of one thread, the only thread is both the first and last thread)

Each family $F$ is further partitioned as a set of sequential segments $\text{seq}(F)$, which are sub-sets of participating threads that are executed internally sequentially.

We further name segment prefix the sub-set of a sequential segment that contains all threads in that segment but the last via $\triangleleft$, i.e. $\forall S \in \text{seq}(F), \text{prefix}(S) = \{ t \mid t \in S \land \exists t' \in S : t \triangleleft t' \}$.

Sequential segments further constrain scheduling as follows:

$$\forall S \in \text{seq}(F), \forall t \in \text{prefix}(S), \text{end}(t) \sim \text{begin}(\text{succ}(t))$$

Rationale: This denotes that within a sequential segment, the end of a thread dominates the start of its successor in the scheduling order. This expresses that there is no concurrency between threads that belong to the same segment, whereas concurrency may still exist across separate segments. This constraint reflects the sequential scheduling of logical threads over thread contexts introduced in sections 3.3.1 and 4.2.

### 7.2.3 Schedule ordering of threads from family creation

1 The abstract machine provides a special synchronizing operation $c[F]$, called “family creation.”

2 This constrains scheduling as follows:

$$\forall t \in F \quad c[F] \sim \text{begin}(t)$$

### 7.2.4 Schedule ordering of threads upon family termination

1 The abstract machine provides a special synchronizing operation $s[F]$, called “family synchronization.”

2 This constrains scheduling as follows:

$$\forall t \in F \quad \text{end}(t) \sim s[F]$$

### 7.2.5 Dataflow synchronizers

1 The abstract machine defines a set $\mathcal{O}$ of dataflow synchronizers and a set $\mathcal{V}$ of unit values.

2 It then provides the following special synchronizing operations on these synchronizers, denoted for any $o \in \mathcal{O}$:

- $r_s(o)$ for “start synchronizing read,”
- $r_f(o)$ for “finish synchronizing read,”
- $q(o)$ for “query synchronizer,”
- $w(o, v)$ for “synchronizing write” ($v \in \mathcal{V}$), and
- $e(o)$ for “clear synchronizer.”
Each thread \( i \) and “finish synchronizing read” are pseudo-operations that always come in pairs in programs, expressed as a single concrete “synchronizing read” instruction that entails both operations immediately one after the other during execution.

Rationale: We distinguish them because only the completion of a synchronizing read is constrained, as described below.

We will denote “\( xs(o) \)” for “any operation that is either a \( r_s, r_f, q, w \) or \( e \) operation on \( o \)”.

All \( xs \) operations on a given synchronizer are atomic, even across threads:

\[
\forall o, \forall xs(o), \forall xs'(o) \neq xs \quad xs \Rightarrow xs' \vee xs' \Rightarrow xs
\]

“Finish read” operations synchronize with writes the first time after a clear:

\[
\forall e(o), \forall r_f(o) : e \Rightarrow r_f \wedge (\exists e' : e \Rightarrow e' \Rightarrow r_f), \forall w(o, v) : e \Rightarrow w \wedge (\exists w' : e \Rightarrow w' \Rightarrow w) \wedge (\exists e' : e \Rightarrow e' \Rightarrow w), w \Rightarrow r_f
\]

The effect of executing an unprovisioned “finish read” operation is undefined, i.e. any \( r_f \) such that \( e \Rightarrow r_f \wedge (\exists w : e \Rightarrow w \Rightarrow r_f) \) may not complete, trigger a fault to signal deadlock or effect some other unspecified behavior.

Each read \( r_f(o) \) evaluates to the value \( v \in V \) stored by the most recent \( w(o, v) \) according to \( \Rightarrow : \forall r(o), \forall w(o, v) \) (\( \exists e(o) : e \Rightarrow e \Rightarrow r \wedge (\exists w' : w \Rightarrow w' \Rightarrow r) \Rightarrow r(o) \) yields \( v \)

Each query \( q(o) \) evaluates to the value \( v \in V \) stored by the most recent \( w(o, v) \) according to \( \Rightarrow \), if any exists; otherwise, i.e. if there is no last \( w \) or if the last non-\( q \) operation on \( o \) is a \( r_s \) or \( e \) operation, it evaluates to a non-deterministic, unspecified value of \( V \).

Each thread \( i \) is associated with a partial function of \( \mathbb{Z} \) to \( \mathcal{O} \), denoted \( V[i] \), that represents its visible dataflow synchronizers. (For example \( V[i](n) \in \mathcal{O} \) is the \( n \)-th synchronizer visible by \( i \).)

### 7.2.6 Mapping of synchronizers to families

\( \mathcal{O} \) is partitioned between families, i.e. a given synchronizer \( o \) may be visible from multiple threads within a family but is not visible from threads belonging to different families:

\[
\forall (i, n, i', n') \quad F(i) \neq F(i') \Rightarrow V[i](n) \neq V[i'](n')
\]

The execution of a program shall occur as if an initial “clear” operation was issued to every visible synchronizer prior to the execution of all threads, i.e. \( \forall i \in T, \forall o \in V[i], \exists e_0(o) : e_0 \Rightarrow \text{begin}(i) \)

### 7.2.7 Dataflow synchronization between families

The abstract machine provides two operations \( w[i](n, v) \) (\( i \in T, n \in \mathbb{Z}, v \in V \)) and \( q[i](n) \), respectively for “remote synchronizing write” and “remote query,” which may be executed by other threads than \( i \).

The execution of one \( w[i](n, v) \) operation incurs, after a finite amount of time, the execution of one \( w(V[i](n), v) \).

The execution of one instance of \( q[i](n) \) incurs, after a finite amount of time, the execution of one \( q(V[i](n)) \); the execution of \( q \) further produces in the thread where it is issued the value produced by the corresponding \( q \).

### 7.2.8 Mapping in the language interface

The items defined above are exposed in the proposed SL extensions to C as follows:

- C’s evaluations map to operations of the abstract machine;
7.3. EXTENDING C WITH MULTI-THREAD CONSISTENCY

- thread programs in SL, and the C functions called recursively from them, map to thread programs in the abstract machine;
- the “sl_create...sl_sync” construct entails the execution of a c (family creation) operation no earlier than the point “sl_create” is reached during execution, and no later than the point “sl_sync” is reached;
- the “sl_create...sl_sync” construct entails the execution of a s (family synchronization) operation at the point “sl_sync” is reached during execution, and thus execution does not proceed past “sl_sync” until all threads in the created family have terminated;
- the logical index range and “block size” parameters to “sl_create” define the sequential segments of the created family, as detailed in Appendix I.5.8.1;
- SL’s dataflow channels map to the abstract machine’s dataflow synchronizers;
- the “sl_seta” construct entails the execution of a \( \overline{w} \) operation, and “sl_geta” entails the execution of a \( \overline{q} \) operation;
- the “sl_setp” construct entails the execution of a \( w \) operation, and “sl_getp” entails the execution of a \( r_s/r_f \) operation pair.

The \( q \) and \( e \) operations are intentionally not exposed in SL, although the platform implementation may support them, in order to encourage the expression of deterministic, sequentializable programs.

7.3 Extending C with multi-thread consistency

\[\square\] In this section, we explain under which circumstances stores by one thread are visible to loads by another thread, as a function of the scheduling order defined in the previous section.

7.3.1 General condition for consistency

1 The abstract machine provides a set \( \mathcal{L} \) of locations, corresponding to “memory addresses.”
2 It then defines two operations \( ld(l \in \mathcal{L}) \) and \( st(l \in \mathcal{L}, v \in \mathcal{V}) \), for “load” and “store” respectively.
3 It then defines for every location \( l \in \mathcal{L} \), a partial consistency order for \( l \), denoted \( \prec \), over all \( ld \) and \( st \) operations operating on \( l \).

\[\text{Rationale:}\] The partial order \( \prec \) is defined independently for every memory address. This is intended to support the independent ordering of memory operations touching separate cache lines in a cache coherency protocol.

4 The abstract machine then defines a visibility property over \( ld \) and \( st \) operations as follows. Given some operations \( st(l, v) \) and \( ld(l) \), if the following conditions all hold:

- \( st \prec ld \), and
- \( \exists st' \neq st \text{ such that } st \prec st' \prec ld \),
- \( \exists st' \neq st \text{ such that } ld \prec st \prec st' \),

then \( st \) is visible from \( ld \).
Rationale: This property establishes visibility as a function of the consistency ordering, separately for every memory address. Note that the third condition is possibly non-intuitive: unless $\prec$ is further constrained, any store not related to a load via $\prec$ effectively "hides" any other store to the same location from that load, even those related via $\prec$ and including those dominated by the load. This provision exists because we consider weakly coherent cache systems which do not protect against erroneous race conditions in programs.

The abstract machine then defines a provision property as follows: any load $ld$ is provisioned if there exists at least one store $st$ visible from $ld$ in $\prec$.

Further, the following hold:

- if an $st$ operation starts, then it completes within a finite amount of time;
  
  Rationale: Stores should complete eventually.

- if a provisioned $ld$ operation starts, then it completes within a finite amount of time; and the actual value produced by its execution is an element of the set of values written by the $st$ operations to the same location that are visible from $ld$.
  
  Rationale: Loads should complete eventually if they are provisioned. Furthermore, if they are provisioned they yield one of the values stored most "recently" according to $\prec$.
  
  Meanwhile, the behavior of unprovisioned $ld$ operations is undefined. For example, an implementation may cause halting, a deadlock, or produce a value that causes halting or deadlock of any subsequent operation using it, or produce a non-deterministically chosen valid value.

A program is consistent if all its possible executions that are compatible with the abstract machine guarantee that all loads are provisioned.

A load operation is a data race if it is provisioned by more than one store.

A program is deterministically consistent if it is consistent and executes no data race.

7.3.2 Communication domains

The abstract machine provides the notion of Implicit Communication Domain (ICD).

Each pair of (thread, location) is associated with exactly one ICD, noted $C(i,l)$.

The abstract machine then specifies that precedence implies visibility, within the same communication domain:

$$\forall(i,j), \forall x^{(i)}(l), \forall y^{(j)}(l) \ [C(i,l) = C(j,l) \land x \sim y] \Rightarrow x \prec y$$

Rationale: ICDs capture the notion of implicit communication between stores and loads within a region of the system, and constrained by the partial ordering between threads. In particular stores from different threads that are not mutually ordered via $\sim$ can "hide" each other as per clause 7.3.1§4. ICD boundaries correspond to boundaries in the system where implicit communication is not guaranteed, even between operations ordered via $\sim$.

7.3.3 Consistency domains

The abstract machine provides the notion of Consistency Domain (CD).
7.3. EXTENDING C WITH MULTI-THREAD CONSISTENCY

Each pair of (thread, location) is associated with exactly one CD, noted $\bar{C}(i,l)$.

The abstract machine then specifies that all loads and stores within the same CD to the same location appear in some order globally visible within the CD:

$$\forall (i,j), \forall x(i)(l), \forall y(j)(l) \quad \bar{C}(i,l) = \bar{C}(j,l) \Rightarrow x^l < y \forall y^l < x$$

**Rationale:** CDs capture the notion of “eventual visibility” of stores for all subsequent loads within a region of the system. Stores not synchronized via $\sim$ cannot “hide” each other within a CD if there are no unordered stores issued outside the CD. CD boundaries correspond to boundaries in the system across which stores may not be propagated automatically.

Henceforth the set of all consistency domains is noted $\mathcal{C}$. We will also denote “$x_C$” for “an operation $x$ executed by a thread associated to consistency domain $C$.”

### 7.3.4 Memory communicators

With the definitions so far, threads can communicate arbitrarily via memory within CDs, and the store-load visibility follows the edges of the scheduling order within ICDs. However, no provision is made to make stores executed in one ICD visible to loads executed in another ICD. For this purpose, we introduce semi-explicit communication operations as follows.

1. The abstract machine defines a set $\mathcal{M}$ of memory communicators, and a set $\mathcal{R}$ of relative locations for use with communicators.
2. It then provides three communicating operations:
   - $b[R](m)$, for “bind $m$ to a set $R$ of relative locations” ($m \in \mathcal{M}, R \subseteq \mathcal{R}$);
   - $p(m)$, for “propagate” updates to the locations bound to $m$” ($m \in \mathcal{M}$);
   - $a(m)$, for “activate” updates to the locations bound to $m$” ($m \in \mathcal{M}$).
   
   $(b$ and $p$ are intended for use by the writer side, whereas $a$ is intended for use by the reader side$)$

3. It also provides a relative addressing partial function $t$ which translates, within a given consistency domain and relative to a given memory communicator, a relative address usable in programs into an address suitable for implicit communication through the communicator:

   $$t : \mathcal{C} \times \mathcal{M} \times \mathcal{R} \mapsto \mathcal{L}.$$

4. The abstract machine then proposes the following service: any store to a location that is scheduled after a $b$ operation and before a $p$ operation and which operates on a location bound by $b$, becomes visible to loads scheduled after an $a$ operation if $a$ is scheduled after $p$, given that $b$, $p$ and $a$ operate on the same communicator. In other words:

   - For any given communicator $m$,
   - given two ICDs $C_1$ and $C_2$,
   - given a range of relative addresses $R$,
   - for any $b$, $p$ operations executed in $C_1$ and $a$ executed in $C_2$ such that $b$ is scheduled before $p$ and $p$ is scheduled before $a$,
   - for any relative address $l$ in that range,
   - if $l_1 = t(C_1, m, l)$ and $l_2 = t(C_2, m, l)$ then
   - for any $st$ operation on $l_1$ scheduled between $b$
   - and $p$ in $C_1$,
   - and for any $ld$ operation on $l_2$ scheduled after $a$
   - in $C_2$,

   $$\forall m \in \mathcal{M}, \forall (C_1, C_2) \in \mathcal{C}^2, \forall R \in \mathcal{R}, \forall (b_{C_1}(R)(m), p_{C_1}(m), a_{C_2}(m) : b \sim p \sim a)$$

   $$\forall l \in \mathcal{R}, \forall v \in \mathcal{V}, \forall st_{C_1}(l_1, v) : b \sim st \sim p,$$

   $$\forall ld_{C_2}(l_2) : a \sim ld,$$
the execution of $ld(l_2)$ occurs as if an operation $st(l_2, v)$ was visible from it in the same ICD, i.e.

$$\exists st_{C_2}(l_2, v) : st \frac{L}{2} ld$$

(the $b(m)$ operation "captures" a range of addresses into $m$ and starts "recording" stores, until a $p(m)$ operation which "propagates" the changes. The $a$ operation in a different consistency domain then "activates" the changes so they can be used by further loads.)

5 If multiple stores to the same location occur between a $b$ and $p$ operation pair, or if multiple $p$ operations are scheduled before a corresponding $a$, then only the last in the $<$ order will be visible to loads occurring after $a$.

7.3.5 Possible implementations of $t$, $b$, $p$ and $a$

Distinct platform implementations may provide the $b$, $p$ and $a$ operation and $t$ function via separate mechanisms. We provide three examples below.

Note that $a$, $b$ and $p$ are not synchronizing; therefore they must be combined with the other synchronization mechanisms from section 7.2 to ensure e.g. that $a$ is not executed before $p$ completes.

7.3.5.1 Using message passing and a “push” protocol

- $b$ prepares output buffers for a communicator;
- $t$ accesses the output buffers;
- $p$ sends the buffer to the “destination” of the communicator, the data is received asynchronously at the destination;
- $a$ finishes waiting on reception of the data.

7.3.5.2 Using message passing and a “pull” protocol

- $b$ prepares output buffers and associates addresses with the communicator;
- $p$ is no-op;
- $a$ remotely reads the data from the “sources” of the communicator;
- on the origin side, $t$ is the identity; on the pulling side, $t$ either is the identity (single address space) or addresses a receive buffer.

7.3.5.3 Using the proposed platform

- the $b$ and $m$ operations are no-ops,
- $t$ is the identity;
- $p$ is implemented using a write memory barrier which flushes outstanding stores globally.

7.3.6 Mapping in the language interface

The items defined above are exposed in the proposed SL extensions to C by stating that the addresses of bytes in C objects map to locations in the abstract machine, and that accesses to objects in programs map to $ld$ and $st$ operations. The visibility of object updates from object reads is then decided by the consistency rules and the scheduling order as per the specification above.
7.4. EXAMPLES OF INCORRECT SYNCHRONIZATION

<table>
<thead>
<tr>
<th>thread i:</th>
<th>thread j:</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>A: st(l1, v1)</td>
<td>D: st(l2, v3)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>B: st(l1, v2)</td>
<td>E: ld(l2)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>C: ld(l1)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Listing 7.1: Two unrelated threads.

Note that no further language support is available here to control the communication operations $b$, $p$ and $a$ and use the $t$ function. Because of this, the language described in Appendix I can only be used to program within one communication domain.

Extra SL support for cross-ICD communication has been explored by peers and is reported on in [Mat10]: their work proposes to introduce explicit “memory objects” into the language, which are a combination of dataflow synchronizers and memory communicators, operations on these that control $b$, $p$ and $a$, and accessors for the $t$ function. These are then automatically removed, at compile time or run-time, if both sides of a communicating activity are known to be running within the same consistency domain.

We also sought to support cross-ICD communication in our proposed platform from chapter 5 without inserting new language constructs. We noted that $a$, $b$ and $t$ are transparent (cf. section 7.3.5.3 above), and then we proposed to insert a $p$ operation in the generated code immediately before every $c$ and $s$ operation. This enables implicit cross-ICD communication from all cores. While we implemented this feature in our proposed compiler from chapter 6, we highlight that this technique incurs excess memory barriers when threads are created within the same ICD. A more efficient approach should thus attempt to determine automatically when $p$ is not needed based on placement information and elide it in those cases.

7.4 Examples of incorrect synchronization

7.4.1 Abstract example to illustrate the hiding effect

Consider a program causing the concurrent execution of the two threads in listing 7.1, unrelated through synchronization. By construction, we have $A \mathrel{\mathcal{R}} C$ and $B$ is visible from $C$. Likewise, $D \mathrel{\mathcal{R}} E$ and $D$ is visible from $E$. The program is consistent.

Consider now a program executing concurrently the threads in listing 7.2. Because we do not know that $C \mathrel{\mathcal{R}} D$, nor do we know that $D \mathrel{\mathcal{R}} C$, we must understand that $D$ hides both $A$ and $B$, so $C$ is not dominated by a store to $l$ and the program is not consistent.

7.4.2 Invalidated idioms

We assume the following examples run within one CD.

A load $ld$ may observe the value written by a store $st$ that happens concurrently with $ld$. Even if this occurs, it does not imply that loads happening after $ld$ will observe stores
### Listing 7.2: Two unrelated threads with a race condition.

```plaintext
thread i: thread j:
... ... 
A: st(1, v1) D: st(1, v3)
... ...
B: st(1, v2)
...
C: ld(1)
...
```

Listing 7.2: Two unrelated threads with a race condition.

```plaintext
int a, b;
sl_def(f) {
  a = 1; b = 2;
} sl_enddef

sl_def(g) {
  print(b); print(a);
} sl_enddef

sl_def(main) {
  sl_create(,,,,,f);
  sl_create(,,,,,g);
  sl_sync();
  sl_sync();
} sl_enddef
```

Listing 7.3: Independent ordering of loads/stores to different addresses.

to another address that happened before \textit{st}. For example, in the program from listing 7.3, it can happen that \texttt{g} prints 2 and then 0.

Double-checked locking is an attempt to avoid the overhead of synchronization. For example, the \texttt{twoprint} program might be incorrectly written as in listing 7.4: there is no guarantee that, in \texttt{doprint}, observing the store to \texttt{done} implies observing the store to \texttt{a}. This version can unexpectedly, but correctly print an empty string instead of "\texttt{hello, world}". Instead, proper synchronization is achieved using listing 7.5.

Another incorrect idiom is busy waiting for a value, as in listing 7.6. As before, there is no guarantee that, in \texttt{main}, observing the store to \texttt{done} implies observing the store to \texttt{a}, so this program could print an empty string too. Moreover, if the threads are merely in the same ICD but maybe in different CDs, there is no guarantee that the store to \texttt{done} will ever be observed by \texttt{main}, since there is no synchronization between the two threads. The loop in \texttt{main} is not guaranteed to finish. There are subtler variants on this theme, such as the program in listing 7.7. Even if \texttt{main} observes \texttt{g != NULL} and exits its loop, there is no guarantee that it will observe the initialized value for \texttt{g->msg}.
7.4. EXAMPLES OF INCORRECT SYNCHRONIZATION

---

```c
char* a; bool done;
sl_def(setup) {
   a = "hello ,world";
   done = true;
} sl_enddef

sl_def(doprint) {
   if (!done) {
      sl_create( , , , , , , setup);
      sl_sync();
   }
   print(a);
} sl_enddef

sl_def(twoprint) {
   sl_create( , , , , , , doprint);
   sl_create( , , , , , , doprint);
   sl_sync();
} sl_enddef
```

Listing 7.4: Implementation of `twoprint`, insufficiently synchronized.

---

```c
sl_def(twoprint) {
   sl_create( , , , , , , doprint); sl_sync();
   sl_create( , , , , , , doprint); sl_sync();
} sl_enddef
```

Listing 7.5: Proper synchronization for `twoprint`.

---

```c
char* a; bool done;
sl_def(setup) {
   a = "hello ,world"; done = true;
} sl_enddef

sl_def(main) {
   sl_create( , , , , , , setup);
   while (!done) {} 
   print(a);
   sl_sync();
} sl_enddef
```

Listing 7.6: Invalid busy waiting for a value.
CHAPTER 7. DISENTANGLING MEMORY AND SYNCHRONIZATION

```c
typedef struct {
  char *msg;
} T;

T *g;

sl_def(setup) {
  T *t = malloc(sizeof(T));
  t->msg = "hello ,\_\_world";
  g = t;
} sl_enddef

sl_def(main) {
  sl_create( , , , , , , setup);
  while (g == NULL) {}
  print (g->msg);
  sl_sync();
} sl_enddef
```

Listing 7.7: Invalid busy waiting on a pointer.

7.5 Pitfall: atomicity anomalies

Atomicity anomalies can appear when the granularity of consistency offered by an implementation is different from the one assumed by the high-level description of a program. Intuitively, the granularity of consistency corresponds to the minimum “size” of two values so that, if they are stored “next to each other” they will still have different locations.

The first anomaly is the following. Suppose that machine addresses identify 4-byte long objects, but the programmer is treating one machine object as 4 different 1-byte abstract objects each with a different (abstract) location. Two independent threads may each perform an update to two different abstract locations, expecting each update to be visible to a successor load within the same thread from the same (abstract) location. But, if these 1-byte values are packed into the same 4-byte platform location, then these stores are really concurrent stores to the same location. Consequently, one of the two stores may non-deterministically mask the other, or they may mask each other, and the update to one of the bytes, or both, may be lost. The solution to avoid this problem is to avoid packing together abstract objects that might be updated by concurrent stores. In our platform from part I and chapter 5, the machine granularity is 1 byte so this problem is avoided.

The other anomaly is the following. Suppose the system supports 4-byte concrete objects, but the programmer wants to manipulate an 8-byte logical object. If two concurrent threads each update the entire 8-byte object, the programmer might expect a common successor 8-byte load via → to receive one of the two 8-byte values written previously. However, the 8-byte load may non-deterministically receive 4 bytes of one value and 4 bytes of the other value, because the 8-byte load is really two 4-byte loads, and the consistency of the two halves is maintained separately. Note that this problem can only occur if the load is a data race. When programs are written to avoid data races entirely, programmers need not worry about it. In our platform from part I and chapter 5, this granularity anomaly only appears with assignments to aggregate types whose size is larger than the underlying ISA’s word size, that is 8 bytes on Alpha or 4 bytes on SPARC.
7.6 Relationship with the latest C and C++ specifications

Concurrently to our own research, the American National Standards Institute (ANSI) published a new specification for both the C and the C++ languages [II11b, II11a]. Compared to [II99, II03], this new version introduces concurrency semantics in both languages, purposely crafted by their respective working groups to be mostly compatible with each other. These additions can be summarized as follows:

- the notion of concurrently executing threads is introduced. C++ specifies that all threads are fairly scheduled (“Implementations should ensure that all unblocked threads eventually make progress.” [II11a, 1.10§2]), whereas C does not.
- synchronization between threads is introduced via special atomic objects, which are regular memory-based objects declared with the specifier _Atomic. Accesses to the same atomic object are globally ordered, and special “acquire” operations (used for mutex locks) synchronize with special “release” operations (used for unlocks).
- the visibility of updates to non-atomic objects relative to subsequent evaluations of those objects is decided via a “happens before” partial order between operations, constrained between concurrent threads by the interleaving of accesses to atomic objects.

In short, the designers of these extensions assume that the memory is “the great coordinator” as we explained in section 7.1. Their definition of the “happens before” relation between operations is particularly complex, because it must simultaneously specify the visibility of non-atomic object updates and the visibility of updates to atomic objects. Although these updates have quite separate consistency rules (atomic accesses are synchronizing, non-atomic accesses are not), because they are both memory objects their definition must be interleaved to decide what constitutes a race condition vs. a consistent access.

In contrast, our semantics allow the programmer to reason about execution order independently from memory accesses. The resulting situation is that although we started from the same language (C and C++ as per [II99, II03]), our proposed semantics diverge from the direction taken by ANSI for C and C++. The question thus arises of whether this situation is desirable and what are its consequences.

Here we start by observing that the concurrency semantics of [II11b, II11a] can be emulated in our environment by containing the execution of the entire program within one consistency domain, and requiring the memory system to provide support for atomic transactions. This is possible because:

- neither [II11b] nor [II11a] mandates a minimum number of simultaneously executing threads, so the platform restrictions on the number of thread contexts may freely constrain how many of [II11b, II11a]’s thread creations may occur;
- all valid non-racy executions according to [II11b, II11a] are valid non-racy executions in our abstract machine within one CD;
- data races are defined in both [II11b, II11a] and our abstract machine to result in undefined behavior.

Because [II11b, II11a] can be emulated, we can provide a backward compatibility environment in an implementation of SL, where any program that requires [II11b, II11a]’s semantics is forced to run on a region of the system contained within one CD, without changing the abstract machine. We conclude that our proposed abstract machine is more general than those envisioned by ANSI for C and C++. We then revisit how operating software can constrain the placement of programs to specific consistency domains in chapter 11.
7.7 Position of the model from the programmer’s perspective

In [KMZS08] the author attempts to isolate criteria to classify programming models for parallel systems. The proposed criteria are:

1. **System architecture**: shared memory vs. distributed memory.
2. **Programming methodologies**: how concurrency is exposed to programmers (API, directives, language extensions, etc)
3. **Worker management**: how execution units are managed, *implicit* (MPI, OpenMP) vs. *explicit* (Pthreads)
4. **Workload partitioning scheme**: how workloads are divided in chunks (tasks), *implicit* (OpenMP) vs. *explicit* (MPI)
5. **Task-to-worker mapping**: how tasks are mapped to workers, *implicit* (OpenMP) vs. *explicit* (POSIX threads)
6. **Synchronization**: time order in which shared data is accessed, *implicit* (UPC) vs. *explicit* (MPI)
7. **Communication model**: private shared address space, message passing, global partitioned address space, etc.

Within this classification we place our system as follows. With regards to system architectures, our system aims to target both shared memory systems and distributed memory systems. Information about locality, if available, can be used to optimize code and reduce communication overhead by eliding uses of the $b$, $p$ and $a$ primitives. The proposed abstract machine does not mandate a specific programming methodology. Our SL implementation (including the extensions presented by [Mat10]) use language extensions while another [vTK11] uses an API. Workload partitioning is explicit, but fine grained. Our system suggests the definition of separate threads even for very small units of work. Once this is done, the granularity can be made coarse again by aggregating threads together automatically (run as loops), as we suggest in chapter 10. This has been described also in [Mat10]. Worker management, task-to-worker mapping and synchronization are mostly implicit in our system. A form of optional explicit mapping is introduced later in chapter 11. The communication model, subject of this chapter, is a global address space augmented with *consistency and implicit communication domains*, that defines regions of the address space where communication can be largely implicit. Across domains, communication must be more explicit.

7.8 Other related work

Our abstract machine borrows concepts both from Global Address Space (GAS)-derived models such as UPC [UPC05], Fortress [ACH*08], Titanium [YSP*98], Co-Array Fortran [NR98], and from dataflow (implicit synchronization) models such as Cilk [BJK*95] and Google Go [Goo].

7.8.1 Weak consistency within ICDs, overview from Go

Go is a concurrent programming language developed by Google. It provides *goroutines*, which are small lightweight threads. Memory consistency between concurrent goroutines is relaxed.
The weak consistency model proposed by our system within a given implicit communication domain shares much similarity with Go. The similarity is so close that the description of Go's memory consistency\(^1\) also describes consistency within an ICD in our system. In particular, by stating that stores not ordered by thread synchronization can hide each other, Go shares our visibility semantics from clause 7.3.1§4.

### 7.8.2 Comparison with Cilk

Cilk [BL93, BJK+95, Joe96] was developed by MIT as both a programming language and an execution model. The Cilk programming language extends a subset of C in a way that a deterministic Cilk program stripped of all Cilk keywords becomes a valid C program with the same functional behavior.

However the Cilk execution model is more general than the Cilk language itself. The unit of work in Cilk, as with our system, is a logical thread. Thread definitions in Cilk are also schedule agnostic, i.e. the synchronizing dependencies between threads are explicit and the run-time system only guarantees that they are satisfied without specifying the execution order.

The essential difference between Cilk and our system is the granularity of synchronization. In Cilk, a thread begins to execute only after all its dependencies are satisfied; it then runs until completion at which point its output is checked to see if it satisfies a dependency for another thread. In our system, threads can start even when not all their dependencies are satisfied; running threads can wait for termination of threads they have created previously; and ordering is guaranteed between individual reads and writes to dataflow synchronizers visible across threads. Also, new threads can be created and start executing even during the execution of their parents.

Due to this difference, our execution model is more general and fine-grained than Cilk with respect to synchronization and concurrency. This stems from the following observation: any Cilk program using \(N\) concurrent execution units can be executed within our model with \(N\) execution units, using only one consistency domain. However a program that can use \(N\) concurrent execution units in our model may not be able to use more than \(M\) units when expressed using Cilk primitives, with \(M < N\), because parent/child and family siblings which are concurrent in our system would need to be scheduled in sequence in Cilk to satisfy dependencies.

Apart from this principal difference, the consistency models proposed for Cilk ([Joe96, Chap. 6], [BFJ+96]) and our model are similar. In Cilk, stores are visible from loads if they dominate loads according to the directed acyclic synchronization graph defined by inter-thread dependencies. This is called “DAG consistency”. In our model, stores are also visible from loads if they dominate loads according to the directed acyclic precedence graph with \(\rightarrow\) edges. In both models, multiple stores visible from a single load are resolved non-deterministically. Our model further states that consistency is not decidable when there exists a store that is not ordered with a load, whereas Cilk leaves this topic unspecified.

Also, our system extends the Cilk model by defining communication across consistency domains. This extension preserves the simplicity of defining consistency based on the dependency graph, but requires to enclose reads and writes to memory between \(a\), \(b\) and \(p\) communication operations.

\(^1\)http://golang.org/doc/go_mem.html
7.8.3 Communication granularity, comparison with CAF / UPC

In [CDMC+05], the authors study Co-Array Fortran (CAF) and Unified Parallel C (UPC) as two languages that offer SPMD over a global address space. Their primary statement is that:

[...] CAF and UPC programs deliver scalable performance on clusters only when written to use bulk communication. However, our experiments uncovered some significant performance bottlenecks of UPC codes on all platforms. We account for the root causes limiting UPC performance such as the synchronization model, the communication efficiency of strided data, and source-to-source translation issues. We show that they can be remedied with language extensions, new synchronization constructs, and, finally, adequate optimizations by the back-end C compilers.

The solution the authors propose to optimize communication is twofold. The first is to adapt the granularity of synchronization to the actual functional requirement of programs, i.e. minimize the scope of synchronization and avoid barriers. This is present in our system already since all synchronization is explicitly scoped by default.

The other solution is to group individual accesses to memory into bulk communication of entire chunks of memory. Then they suggest directions to partly automate this process and synthesize bulk transfers:

We believe that developing effective compiler algorithms for synchronization strength reduction is appropriate. However, we also believe that having point-to-point synchronization available explicitly within the languages is important to avoid performance loss when compiler optimization is inadequate. [...] 

The SL constructs for synchronization using explicit dataflow channels, and our choice to expose the b, p and a operations for consistency, go along with this suggestion.

7.8.4 ZPL’s Ironman interface

ZPL [Cha01] was a precursor to Chapel [CCZ07] designed at the University of Washington. The designers of ZPL were already highly concerned by issues of memory consistency over weakly coherent systems and distributed memories. Like us, they were interested to both support distributed memory and automate the elision of communication primitives when communication partners were running in the same memory domain. Their solution was the Ironman interface, introduced in [CCS98] and detailed in [Cha01, Chap. 4]. As we do in section 7.3, this interface proposes abstract primitives for use in the code generated by compilers; the primitives are then projected to different implementations in the run-time environment depending on the underlying communication infrastructure. The primitives in Ironman are, on the writer side:

SR (Source Ready) : The values at the source processor will not be written again prior to transfer. The source processor is now ready to begin the data transfer.

SV (Source Volatile) : The data at the source processor is about to be overwritten. Execution cannot continue until the transfer is completed.

Our proposed p operation combines the function of Ironman’s SR and SV operations. The primitives on the reader side are:
7.8. OTHER RELATED WORK

**DR (Destination ready)**: The locations at the data destination will not be used again until the transfer has completed. The destination processor is now ready to accept data from the source processor.

**DN (Destination Needed)**: The values at the data destination are about to be read. Execution cannot continue until the data from the source processor has been received.

Our proposed *a* operation combines the function of DR and DN.

The Ironman primitives are designed to facilitate the overlap of communication with computation, by separating issuing a communication operation from completing it. As such they are similar to e.g. MPI’s asynchronous send/receive APIs; their difference with an API is that they can be elided at run-time by direct shared memory synchronization if the communication endpoints are close to each other.

These primitives further assume that any address in the local address space can serve as communication buffer. In particular, they does not offer a handle on system that require buffers to be allocated in a dedicated memory space, like our *b* and *a* operations do.

---

**Summary**

Abstract machines establish a contract between a platform provider and the designers of higher-level abstractions, including programmers. This should be provided and detailed, so as to create an abstraction interface where external observers can reason about program semantics independently from specific implementations.

- In our work, we have designed such an abstract machine to describe the semantics of the platform introduced in part I and chapter 5. The abstract machine describes dataflow synchronization and establishes memory consistency as a derived property of the directed acyclic graph of the scheduling order in programs. It also introduces “communication and consistency domains” to expose different kinds of implicit communication using hierarchical shared memory systems. We have postulated that the new concurrency semantics added to the latest revisions of the C and C++ standards can be emulated using the proposed abstract machine.