Pervasive Theory of Memory

Ulan Degenbaev[1]*, Wolfgang J. Paul[1]**, and Norbert Schirmer[2]*

[1] Saarland University, P.O. Box 15 11 50, 66041 Saarbrücken, Germany
[2] German Research Center for Artificial Intelligence (DFKI),
P.O. Box 15 11 50, 66041 Saarbrücken, Germany

Abstract. For many aspects of memory theoretical treatment already exists, in particular for: simple cache construction, store buffers and store buffer forwarding, cache coherence protocols, out of order access to memory, segmentation and paging, shared memory data structures (e.g. for locks) as well as for memory models of multi-threaded programming languages. It turns out that we have to unite all of these theories into a single theory if we wish to understand why parallel C compiled by an optimizing compiler runs correctly on a contemporary multi core processor. This pervasive theory of memory is outlined here.

1 Introduction

One subproject of the Verisoft-XT project[3] is to formally verify as big a portion as possible of the Microsoft Hyper-V virtualization product that is shipped as a component of Microsoft Windows Server 2008. This hypervisor is a multi-threaded C program with involved parallel algorithms and external assembler functions running in translated mode on contemporary multi core processors. The verification tool VCC [1] used to verify such programs is developed in parallel with the verification effort for the hypervisor and other programs. This paper is motivated by the question how one would prove the soundness of VCC. The rough road map is clear and was for instance followed with formal proofs in the former Verisoft project[4] [2]:

1. Define a semantics S for the subset C' of C used in the project. In the former Verisoft project big step and small step semantics for C'=C0 were used [3–5].
2. Show that the verification condition generator used is sound with respect to semantics S [5, 6].

* Work funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008.
** Work partly funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008.

[3] `http://www.verisoftxt.de`

[4] `http://www.verisoft.de`

3. Show that the compiler used correctly translates programs from C' to the instruction set architecture (ISA) of the processor used. In the Verisoft project a non optimizing compiler from C0 to the ISA of the VAMP processor [7,8] was verified [3,4].
4. in case one has doubts that the ISA in the manuals is the ISA realized by the hardware: show that the processor hardware correctly interprets the ISA [9].

In the context of the hypervisor effort we have to deviate from this road map due to the following difficulties:

1. Complexity of the processor: the documentation of the x64 ISA of contemporary multi core processors consists of thousands of pages [10,11]. Of course the building plans of the processors are not public. Even if we had access to them they would be too complex to be completely verified using the present state of the art tools.
2. Memory model of the processor: modern processors use a weak shared memory model [12–15]. A ten page white paper [16] is supplied to clarify this model beyond the thousands of pages of documentation.
3. Complexity of the compiler: in case of the hypervisor an optimizing Microsoft compiler (to whose source code we could gain access) translates multithreaded C programs to the x64 ISA. This compiler is also too complex for present verification technology.
4. Compiler correctness: the theoretical treatment of compiler correctness for target architectures with a weak memory model is still a field of ongoing research [17,18].

We proceed as follows: we first outline how to reverse engineer a memory system for processors which is consistent with the documentation [10,11,16] and with our ideas how to build processors [19,20]. Section 3.2 gives simple sufficient conditions for store buffers (between processors and memory) to become invisible, namely in case of single processors and, trivially, in case of fenced memory transactions (a fenced transaction is only executed when the store buffer is empty). In the spirit of [7] Section 3.3 sketches very briefly how to show hardware correctness of a memory system consisting of a single cache and a main memory. In Section 3.3 we outline a proof of the corresponding result for a cache coherent shared memory. In order to obtain the result we later need, one has to combine three arguments: i) a classical transaction based correctness proof for cache coherence protocols, ii) its extension to compatible families of protocols as introduced in [21] and used in modern processors, and iii) a construction of the sequential order from the termination times of hardware transactions. In Section 3.4 we outline the arguments, why translated 'linear memory' is realized by multi level address translation. In Section 3.5 we reverse engineer a multi core processor with Tomasulo scheduler, memory management units, store buffers as well as coherency snooping as introduced in [22] and outline the correctness proof.

Assuming that we guessed the memory model correctly we then show in Section 3.6 how to initialize a contemporary multi core processor such that the hardware threads see the weak memory model derived previously in translated linear memory.

Finally we turn to the theory of compilation for multi-threaded C programs in weak memory models. Starting from a small step semantics for sequential programs we derive as a starting point an unrestricted naive parallel C semantics, which unfortunately we don't know how to compile into an efficient parallel assembler program. In Section 4.2 we review the correctness theorem from [3] for a non optimizing compiler for a sequential subset of C and then modify its statement (without proof) for optimizing compilers for multi-threading code; for a formal correctness proof of an optimizing compiler for a sequential subset of C see [23]. In the short Section 4.3 we sketch how to compile volatile variables such that in the compiled program they form a sequentially consistent portion of the weak memory. Using test and set operations on volatile variables we can implement locks which in turn permit to implement synchronized parallel C; this last step is explained in Section 4.4.

2 Notation

We denote the concatenation of bit strings $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$ by $a \circ b$. For bits $x \in \{0,1\}$ and positive natural numbers $n \in \mathbb{N}^+$ we define inductively $x^1 = x$ and $x^n = x^{n-1} \circ x$. Thus, for instance $0^5 = 00000$ and $1^2 = 11$.

Overloading symbols like $+$, $\cdot$, and $<$ we will allow arithmetic on bit strings $a \in \{0,1\}^n$. In these cases arithmetic is binary modulo 2^n (with nonnegative representatives).

We model memories m as mappings from addresses a to byte values $m(a)$. For natural numbers d we denote by $m_d(a)$ the content of d consecutive memory cells (from right to left) starting at address a, so $m_d(a) = m(a+d-1) \circ \cdots \circ m(a)$. We select ranges of a bit string by $x[hi{:}lo]$, e.g. $x[11{:}0]$ to select the 12 least significant bits of x.

3 Architecture Aspects

3.1 Sequential Memory

The state of a sequential memory with address range A and data range D is modeled by a function

$$m : A \rightarrow D$$

where $m(a)$ denotes the current content of memory cell with address a. We consider here three kinds of atomic memory transactions: read, write as well as test and set. We number transactions with indices $i \in \mathbb{N}_0$ and define the predicates

- $r(i)$: transaction i is a read,
- $w(i)$: transaction i is a write, and
- $ts(i)$: transaction i is test and set.

With each transaction i we associate an address $ad(i)$ and (input or output) data $data(i)$. We define the memory content before transaction i by m^i. The semantics of read, write and test and set transactions can then be defined by:

- $r(i) \rightarrow data(i) = m^i(ad(i)) \wedge m^{i+1}(a) = m^i(a)$,
- $w(i) \rightarrow m^{i+1}(a) = \begin{cases} data(i) & \text{if } ad(i) = a, \\ m^i(a) & \text{otherwise, and} \end{cases}$
- $ts(i) \rightarrow data(i) = \begin{cases} 1 & \text{if } m^i(ad(i)) = 0, \\ 0 & \text{otherwise} \end{cases}$

  $\wedge$

  $m^{i+1}(a) = \begin{cases} 1 & \text{if } ad(i) = a \wedge m^i(ad(i)) = 0, \\ m^i(a) & \text{otherwise.} \end{cases}$

The predicate

$$W(a,i) \equiv \exists j < i : ad(j) = a \wedge (w(j) \vee (ts(j) \wedge data(j) = 1))$$

says that memory at address a has been written before transaction i. For such a and i we define the last transaction before transaction i that wrote to address a

$$last(a,i) = \max\{j < i : ad(j) = a \wedge (w(j) \vee (ts(j) \wedge data(j) = 1))\}.$$

Because $m^{j+1}(a) = m^j(a)$ for $j \in [last(a,i)+1 : i-1]$ one has

Lemma 1.

$$m^i(a) = \begin{cases} m^{last(a,i)+1}(a) & \textit{if } W(a,i), \\ m^0(a) & \textit{otherwise} \end{cases}$$

and hence

$$r(i) \rightarrow data(i) = \begin{cases} data(last(ad(i),i) & \text{if } W(a,i) \\ m^0(ad(i)). \end{cases}$$

Observe that any system obeying the last equation defines a memory system, namely

$$m^i(a) = \begin{cases} data(last(a,i)) & \text{if } W(a,i), \\ m^0(a) & \text{otherwise.} \end{cases}$$

3.2 Store Buffers

A store buffer *sb* is a small queue between processor and memory m storing pending write transactions (see Fig. 1). We provide store buffer entries *sbe* with the following components:

- *sbe.ad*: the address of the write transaction,
- *sbe.data*: the data to be written, and
- the ghost component *sbe.index*: the index of the write transaction.[5]

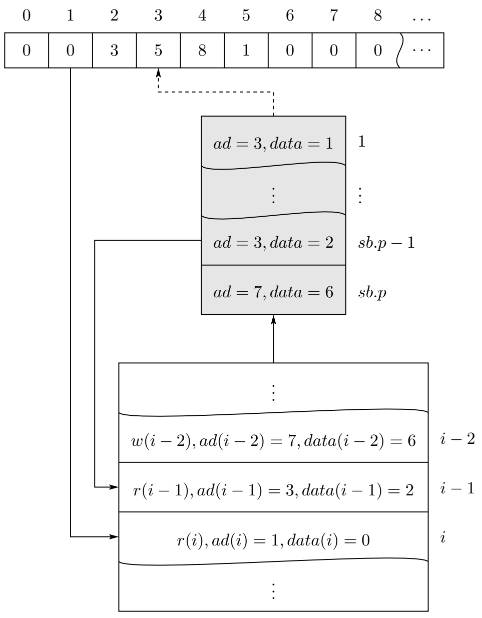

Fig. 1. Store buffer

We model a configuration of a store buffer as a pair $sb = (sb.p, sb.m)$ where $sb.p$ is the number of currently pending store requests and $sb.m$ maps the set $[1 : sb.p]$ to the set of store buffer entries. Initially the store buffer is empty

$$sb^0.p = 0.$$

If transaction i is a write request, its index, address and data are inserted at the end of the queue. Thus for the new configuration sb' we have

$$\begin{aligned} sb'.p &= sb.p + 1 \\ sb'.m(k) &= sb.m(k) \text{if } k < sb'.p \\ sb'.m(sb'.p).ad &= ad(i) \\ sb'.m(sb'.p).data &= data(i) \\ sb'.m(sb'.p).index &= i. \end{aligned}$$

If a write request is sent to the memory, it is deleted from the front of the queue

$$\begin{aligned} sb'.p &= sb.p - 1 \\ sb'.m(k) &= sb.m(k+1) \text{ for } 1 \leq k \leq sb'.p. \end{aligned}$$

The store buffer stores requests in temporal order, i.e. we have

[5] Recall that ghost components are not implemented and serve only for mathematical arguments

Invariant 1

$$k < k' \rightarrow sb.m(k).index < sb.m(k').index.$$

Predicate $hit(a, sb)$ signals that a write request with address a is in the store buffer:

$$hit(a, sb) \equiv \exists k \leq sb.p : sb.m(k).ad = a.$$

The entire system consists of

- the processor, and
- the memory system consisting of memory and store buffer.

A memory system step deletes the first store buffer entry and sends it to the memory. This maintains

Invariant 2 *Let* $j = sb.m(1).index - 1$. *Then*

$$m^i(a) = \begin{cases} data(last(a, j)) & : W(a, j) \\ m^0(a). \end{cases}$$

A processor step sends a transaction to the memory system. Write transactions are written into the store buffer. A test and set transaction causes the store buffer to be flushed before being executed. Read transactions $r(i)$ are answered using store buffer forwarding: in case of a store buffer hit ($hit(ad(i), sb)$ we determine the last store buffer entry which has a write request leading to the hit

$$k = max\{k' : sb.m(k').ad = ad(i)\}$$

and return the data in store buffer entry $sb.m(k')$. Otherwise we return data from memory

$$data(i) = \begin{cases} sb.m(k).data & hit(ad(i), sb) \\ m(ad(i)). \end{cases}$$

The invariants imply that the memory system behaves like a single memory:

Lemma 2.

$$r(i) \rightarrow data(i) = \begin{cases} last(ad(i), i) & \textit{if } W(ad(i), i), \\ m^0(ad(i)) & \textit{otherwise.} \end{cases}$$

Thus store buffers are invisible in systems with a single processor. If several processors are connected with store buffers to a shared memory (Fig. 2) the store buffers are visible. Consider the following two threads where shared variables `x` and `y` are initially 0 and `r1` and `r2` are local variables stored in registers:

```
x = 1;          y = 1;
r1 = y;         r2 = x;
```

Thread 1 Thread 2

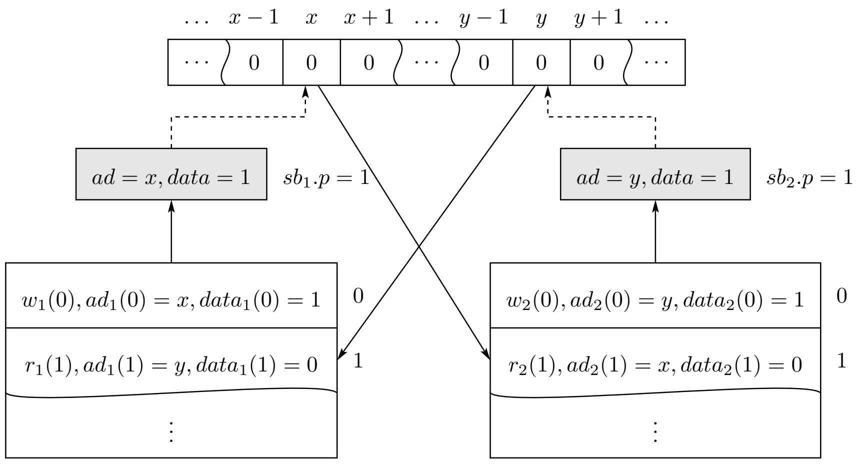
$\ldots$ $x-1$ x $x+1$ $\ldots$ $y-1$ y $y+1$ $\ldots$
$ad = x, data = 1$
$sb_1.p = 1$
$ad = y, data = 1$
$sb_2.p = 1$
$w_1(0), ad_1(0) = x, data_1(0) = 1$
$r_1(1), ad_1(1) = y, data_1(1) = 0$
$w_2(0), ad_2(0) = y, data_2(0) = 1$
$r_2(1), ad_2(1) = x, data_2(1) = 0$

Fig. 2. Multiple store buffers

Can it happen that both `r1` and `r2` contain 0 in the end? On a sequentially consistent machine (for example a machine without store buffers), this cannot happen, since either the assignment to `x` happens before the assignment to `y` or vice versa and the program order is preserved within threads. However, if we take store buffers into account the outcome is valid. Consider the situation where both the assignment to `x` as well as to `y` are in the local store buffers, but have not yet emerged to the memory. Hence the read of `y` in the first thread still sees the value 0 and the read of `x` in the second thread also sees 0.

A brute force way to make store buffers invisible is to use fenced transactions.

Definition 1. *If t is a transaction, we denote by ft the corresponding fenced transaction. A fenced transaction from a processor is directly sent to the memory; it is only executed when its store buffer is empty.*

A trivial consequence is

Lemma 3. *If for a time interval T and an address range A all transactions during T with addresses in A are fenced, then the memory system behaves for this time interval and address range like a shared memory.*

3.3 Caches

Memories m are usually implemented by one or more levels of caches ca which are backed up by a main memory mm. Caches are small and fast memories; they can be implemented in many ways [24–26]. A uniform view on all kinds of cache constructions is provided by *abstract caches.* Decompose addresses $a \in A$ into (cache) line address $a.la$ and offset $a.off$:

$$a = a.la \circ a.off$$

and let LA be the set of line addresses. The configuration ca of an abstract cache is then simply specified by a pair of mappings

– $ca.valid : LA \rightarrow \{0,1\}$. A line la is present in the cache if $ca.valid(la) = 1$.
– $ca.data : A \rightarrow D$. Formally this is just an ordinary memory configuration with the full address range A, but values $ca.data(a)$ are only considered meaningful, if the corresponding cache line is valid, i.e. if $ca.valid(a.la) = 1$.

It is easy to see abstract caches are natural abstractions for the usual cache constructions; we show this for direct mapped caches. Consider the usual decomposition of addresses $a \in A \subseteq \{0,1\}^n$

$$a = a.tag \circ a.line \circ a.off$$

where $a.off \in \{0,1\}^o$ is the offset within a cache line, $a.line \in \{0,1\}^l$ is the local address of a cache line in the cache, $a.tag \in \{0,1\}^t$, and $o + l + t = n$. A direct mapped cache c has three memories all addressed by local line addresses $line$:

– $c.data(line) \in D^o$ is the cache line stored in the local cache line $line$,
– $c.valid(line) \in \{0,1\}$ indicates if the data in line $line$ is valid,
– $c.tag(line) \in \{0.1\}^t$ completes the local line address $line$ to a full line address.

The corresponding abstract cache $ca(c)$ can be defined by

$$\begin{aligned} ca(c).valid(tag \circ line) = 1 &\leftrightarrow c.valid(line) = 1 \wedge c.tag(line) = tag \\ ca(c).data(tag \circ line \circ off) &= c.data(line)[off]. \end{aligned}$$

In a correctness proof for a cache system on the hardware level one has – among others – to consider the following components of the hardware configuration h:

– the main memory $h.mm$, and
– in case of a direct mapped cache, the cache memories $h.c.data$, $h.c.tag$, $h.c.valid$.

From the direct mapped cache $h.c$ one abstracts the abstract cache $ca(h.c)$. From this one defines the (simulated) memory system $m(h)$ simulated by the hardware in configuration h as

$$m(h)(a) = \begin{cases} ca(h.c)(a) & \text{if } ca(h.c).valid(a.line) = 1, \\ h.mm(a) & \text{otherwise.} \end{cases}$$

A hardware correctness proof has also to consider the buses between the cache and main memory as well as the logic controlling transfers of cache lines between cache and main memory. Also hardware correctness proofs have to break read and write transactions etc. down to the cycle level. For memory transaction i one might have to consider start cycles $s(i)$ (a request signal is activated) and end cycles $e(i)$ (a busy signal is taken away or is not activated in the first place). A proof[6] that this memory system is simulated has to establish (among other things):

[6] for a non pipelined cache

Lemma 4.

1. *A read transaction i starting in cycle $s(i)$ returns in cycle $e(i)$ the data $m(h^{s(i)})(ad(i))$*
2. *A write transaction i starting in cycle $s(i)$ produces after cycle $e(i)$ the simulated memory*

$$m(h^{e(i)+1})(a) = \begin{cases} data(i) & \textit{if } a = ad(i), \\ m(h^{s(i)})(a)) & \textit{otherwise.} \end{cases}$$

The first formal correctness proofs for memory systems with caches that can be synthesized to running hardware are reported in [7, 27].

Cache Coherent Shared Memory. The vanilla implementation of shared memory for p processors $P(1), \ldots, P(p)$ is to connect each processor $P(i)$ with its own cache $ca(i)$ and to back up the caches $ca(i)$ with a main memory mm. The entire memory system is supposed to simulate a single sequentially consistent shared memory m [28]. In order to achieve this goal the caches observe each others' transactions via a special bus (this is called snooping) and run a cache coherence protocol. Instead of a single valid bit the caches use a set St of several states for each cache line in order to keep track what cache has what data and how these data match the data in main memory. In abstract caches we therefore replace function $ca.valid$ by a state function

$$ca.s : LA \to St.$$

A very common set of states introduced in [21] is

$$St = \{M, O, E, S, I\}.$$

For $s = ca.s(la)$ the intended meaning is

- $s = M$: the line is exclusive and modified. 'Modified' is not 'clean'.
- $s = O$: the line is shared and modified ('owned').
- $s = E$: the line is exclusive and clean. 'Clean' means that the data in the caches matches the data in main memory and 'exclusive' means that no other cache holds this line la; we formalize this below.
- $s = S$: the line is shared and clean. 'Shared' is not 'exclusive'.
- $s = I$: the line is invalid.

The caches have to maintain the following invariants about the cache states of each line.

Invariant 3 *Exclusive lines are only in one cache:*

$$ca(i).s(la) \in \{E, M\} \land i \neq j \to ca(j).s(la) = I.$$

Invariant 4 *Clean lines match data in main memory:*

$$ca(i).s(la) \in \{E, S\} \wedge a.la = la \rightarrow ca(i).data(a) = mm(a).$$

Invariant 5 *Lines la owned by different caches match:*

$$ca(i).s(la) = ca(j).s(la) = O \wedge a.la = la \rightarrow ca(i).data = ca(j).data.$$

The cache coherence protocol has to decide for each processor transaction whether to announce it on the special bus (in this case we call the transaction *public*) or not (we call the transaction *private*). Private processor transactions are: line invalidations on clean or shared data (the local state is changed to I), read hits (the local state stays the same), write hits on exclusive data (the new state is M), test and set hit if the cached data is $\neq 0$ (the local state stays the same). There is also a private transaction between a cache and main memory, namely if the cache flushes (writes to main memory and invalidates) a clean line. All other transactions are public.

If one views the processors as a distributed system delivering the transactions one by one[7] to their caches then for each of the common protocols it is very easy to show, that the invariants are maintained. Due to the (unrealistically simple) distributed system one can number the transactions $t(x)$ simply by the order in which they are sent to the memory system. The simulated memory m^x before transaction $t(x)$ is

$$m^x(a) = \begin{cases} ca^x(i).data(a) & \text{if } ca^x(i).s(a) \neq I \text{ for some } i, \\ mm^x(a) & \text{otherwise.} \end{cases} \tag{1}$$

Assuming that in the initial caches ca^0 all lines la are invalid ($ca^0(i).s(la) = I$) the invariants imply among other things

Lemma 5.

1. *$m^x(a)$ is is well defined by Equation 1,*
2. $$m^x(a) = \begin{cases} data(last(a, x)) & \textit{if } W(ad(x), x), \\ mm^0(ad(x)) & \textit{otherwise, and} \end{cases}$$
3. *if $t(x)$ is a read transaction, then the memory system returns data $m^x(ad(x))$.*

Variants of this lemma have been extensively studied and model checked. Producing at the hardware level a formal correctness proof for a cache coherent shared memory is still considered a major open problem. Indeed we are not aware of a paper and pencil proof for this problem. For such a proof one has to give a complete implementation, e.g. in the style of [19] or [7].

If we denote transaction j of processor i by $t(i, j)$ we have to consider the start cycles $s(i, j)$ and end cycles $e(i, j)$ of these transactions. Typical durations

[7] which is pointless; the very idea of shared memory is to parallelize transactions

$e(i, j) - s(i, j) + 1$ might be 1 for read hits, 2 for exclusive write hits and many more cycles for public transactions. For a straightforward implementation of the transactions on a single shared data bus between caches and main memory and a single special bus one will be able to show

Lemma 6. *The following transactions do not overlap:*

- *public transactions,*
- *private transactions on the same processor,*
- *private and public transactions with the same address, and*
- *private writes and any transaction with the same address.*

The hardware version of Equation 1 of the memory $m(h)$ simulated by hardware h is

$$m(h)(a) = \begin{cases} ca(i, h).data(a) & \text{if } ca(i, h).s(a) \neq I \text{ for some } i, \\ h.mm & \text{otherwise.} \end{cases} \tag{2}$$

With the help of Lemma 6 one shows that the invariants hold for each cycle, and one gets

Lemma 7. *$m(h)(a)$ is well defined by Equation 2.*

In the spirit of [7] one can define a total order $O(i, j) \in \mathbb{N}$ for the transactions $t(i, j)$ using their end cycles $e(i, j)$: order transactions by their end cycles; order transactions with the same end cycle arbitrarily. Denote by

$$M(t) = max\{O(i, j) : e(i, j) \leq t\}$$

the largest index of a transaction that has completed until cycle t. Let z be the sequential index of a transaction $t(i, j)$ for some i and j, i.e. $z = O(i, j)$ resp. $(i, j) = O^{-1}(z)$.

We define predicate $W'(a, z)$ indicating that address a has been written by a transaction with sequential index $z' = O(i, j) < z$:[8]

$$\begin{aligned} W'(a, z) \equiv \exists z' < z : \; & ad(O^{-1}(z')) = a \wedge \\ & (w(O^{-1}(z')) \vee (ts(O^{-1}(z')) \wedge data(O^{-1}(z')) = 1)). \end{aligned}$$

We define $last(a, z)$ as the last sequential index z' before z of a transaction writing to $ad(i, j)$:

$$\begin{aligned} last(a, z) = max\{z' < z : \; & ad(O^{-1}(z')) = a \wedge \\ & (w(O^{-1}(z')) \vee (ts(O^{-1}(z')) \wedge data(O^{-1}(z')) = 1))\}. \end{aligned}$$

Lemma 8. *The hardware simulates a shared memory which is sequentially consistent with respect to ordering $O(\quad,\quad)$: let $z = M(t)$, then*

[8] We extend functions $ad(j)$, $data(j)$, $w(j)$, $r(j)$, $ts(j)$ defined in 3.1 to multiprocessor case as $ad(i, j)$, $data(i, j)$, $w(i, j)$, $r(i, j)$, $ts(i, j)$ to take additional parameter i – the index of a processor.

1.

$$m(h^t)(a) = \begin{cases} data(last(a,z)) & \text{if } W'(a,z), \\ h.mm^0(a) & \text{otherwise,} \end{cases}$$

2. a read transaction starting in cycle $s(i,j)$ returns in cycle $e(i,j)$ the data $m(h^{s(i,j)})(ad(i,j))$, and

3.

$$W'(ad(i,j), O(i,j)) \rightarrow last(a, O(i,j)) < M(s(i,j)).$$

There is one further highly interesting and important property about cache coherence protocols which to the best of our knowledge have not received much theoretical treatment, namely compatibility within a family F of cache coherence protocols. If the special bus between caches helping to control the cache coherence protocol contains signals like the ones in the classical paper introducing the MOESI protocol [21], then one can specify with each memory access (i,j) the cache coherence protocol $mmode(i,j) \in F$ to be used for transaction i on processor j, and one gets

Lemma 9. *If a compatible family of cache coherence protocols is used in a memory system with caches and the memory mode used is specified separately for each transaction, then the memory system still simulates a sequentially consistent memory.*

Consistent families of cache coherence protocols are implemented in the processors of modern PCs. The result of the lemma is stated quite explicitly in [21]. We have seen neither a paper and pencil proof nor a model checked version of this important result.

3.4 Memory Management Units

Address Translation. We partition memory into pages; here we use the common page size $4K = 2^{12}$. We partition addresses a into page base addresses $a.ba$ and page offsets $a.pof \in \{0,1\}^{12}$, such that $a = a.ba \circ a.pof$. We denote a page with base address ba of memory m by $pg(m, ba)$. It consists of the $4K$ consecutive memory cells starting at address $pa \circ 0^{12}$:

$$pg(m, ba) = m_{4K}(ba \circ 0^{12}).$$

Let $V \subseteq A$ be a set of (virtual user) addresses to be translated by a memory management unit (MMU) and let

$$V.ba = \{a.ba : a \in V\}$$

be the set of page addresses in V. An abstract translation of address range V is specified by

- a translation function $T : V.ba \rightarrow A.ba$ specifying where to redirect memory accesses to pages in A, and

– rights functions $r : V.ba \to \{0,1\}$ specifying the access rights to pages. We consider $r = EXE$ (executable) and $r = RW$ (readable and writable).

For a processor running in translated mode a memory management unit has to redirect memory accesses to addresses $a \in V$ to $T(a.ba) \circ a.pof$ in the following situations

– if a comes from the program counter and $EXE(a.ba) = 1$,
– if a is the effective address of a write access and $RW(a.ba) = 1$, and
– if a is the effective address of a read access.

For all other addresses $a \in V$ and for all addresses $a \notin V$ a page fault has to be generated.

Page Tables are pages that are used to specify abstract translations. They commonly consist of page table entries *pte* occupying 4 bytes resp. one word in a page table. Within a page we can index the entries with page indices $px \in \{0,1\}^{10}$. Thus we can define entry px of page table (with base address) ba as

$$pg(m, ba)[px] = pg(m, ba)_4(px \circ 0^2).$$

Intuitively multilevel address translation is done by traversing the graph G whose nodes are the page tables and whose edges are specified by a certain component *pte.ba* of the page table entries *pte*. But notice that the graph G is dynamic: it can be edited by the processor while the MMU traverses it. Page table entries *pte* usually have components like

– *pte.ba* the base address of the next page table or – at the last step of translation – a user page,
– the present bit *pte.p* indicating that the data of the entry is meaningful,
– rights bits *pte.r*, and
– possibly accessed and dirty bits *pte.acc* and *pte.d*; we will skip over them here most of the time.

Walks. Multi Level Address Translation is achieved by *walking* resp. traversing the page tables in K steps; common values are $K = 3$ or $K = 4$. The information gathered during the traversal is summarized in *walk*. Walk w has the following components:

– $w.vba$: the virtual base address to be translated,
– $w.ba$: the base address of the next page to be accessed,
– $w.r$ for the rights r: the logical AND of bits $pte.r$ in the entries traversed so far, and
– $w.level \in [K : 0]$: the number of page tables that have yet to be traversed.

Walking starts with *initial walks* of level $w.level = K$. No rights have yet been restricted, so $w.r = 1$ for all r. The base address $w.ba$ of the page table where the traversal starts is stored as the *ba*-component of a processor register dedicated for this purpose; in *x64* processors this register is called *CR3*. Thus $w.ba = CR3.ba$.

For the actual traversal we decompose base addresses ba of pages into K page indices $ba.px[i]$:

$$ba = ba.px[K] \circ ba.px[K-1] \circ \ldots \circ ba.px[1] = ba.px[K:1].$$

The width of $px[i]$ depends on the size of a page table entry, which can be 4 or 8 bytes. In case of a 4-byte page table entry, there are $4K/4 = 1024$ entries per page table, and, therefore, the width of $px[i]$ is 10 bits. Respectively, for 8-byte page table entries the width $px[i]$ is 9 bits.

Extension of a level x walk w to a walk $wext(w)$ makes use of the level x page index $w.vba.px[x]$ of the address to be translated, the MMU accesses entry

$$pte = pg(m, w.ba)[w.vba.px[x]].$$

If it is not present $(pte.p = 0)$ a page fault is generated. Otherwise, one sets

$$\begin{aligned} wext(w).ba &= pte.ba \\ wext(w).r &= w.r \wedge pte.r \\ wext(w).level &= w.level - 1 \end{aligned}$$

The walk w is complete if level $w.level = 0$. The translated base address $w.ba$ of a complete walk is a translation for the walks virtual base address $w.vba$. An iterated walk extension $wext^x(w)$ is obtained in the obvious way by $wext^0(w) = w$ and $wext^{x+1}(w) = wext(wext^x(w))$

Translation Look Aside Buffers. Walking the page tables is slow as it requires many memory accesses. Therefore one collects translations $(w.vba, w.ba, w.r)$ found during the walking in a cache called the *translation look aside buffer* resp. the TLB. However, as processors do *not* keep this cache consistent with the page tables, it is the users responsibility to evict translations from the TLB, that should not be used any more. Translations (there can be several) for single virtual base addresses are evicted by so called $invlpg(vba)$ instructions. There are also instructions for clearing the entire TLB.

Implementing an Abstract Translation. Suppose virtual address range V, translation function T, and rights functions r of an abstract translation are given. We construct a tree G of page tables such that walking G produces the translations prescribed by the abstract translation. Denote by

$$P_x = \{vba.px[K:x] : vba \in V.ba\} \text{ for } 2 \leq x \leq K+1$$

the set of prefixes of the virtual base addresses formed by page indices from $K+1$ down to 2. Note that $P_{K+1} = \epsilon$ is an empty prefix. Let

$$P = \bigcup_{x=2}^{K+1} P_x$$

be the set of all prefixes. For each prefix $p \in P$ we allocate in memory m a separate page table with base address $ptba(p)$, such that

$$\forall q \in P : q \neq p \Rightarrow ptba(q) \neq ptba(p).$$

The entries of the page table corresponding to prefix $vba.px[K : x]$ are defined by induction on x from the leaves ($x = 2$) to the root ($x = K + 1$). Let pte be the page table entry with index $vba.px[x-1]$ in the page table corresponding to prefix $vba.px[K : x]$, i.e.

$$pte = pg(m, ptba(vba.px[K : x]))[vba.px[x-1]].$$

For $x = 2$ set the present bits $pte.p = 1$ if $vba \in V.ba$, and for present entries set

$$\begin{aligned} pte.ba &= T(vba) \\ pte.r &= r(vba) \end{aligned}$$

For $x > 2$ set $pte.p = 1$ if $vpa.px[K : x-1] \in P_{x-1}$, and for present entries set

$$\begin{aligned} pte.ba &= ptba(vpa.px[K : x-1]) \\ pte.r &= 1 \end{aligned}$$

and point with special purpose register *CR3* to the root of the tree obtained in this way

$$CR3 = ptba(\epsilon).$$

By induction on x one now easily shows

Lemma 10. *Let $vba \in V.ba$, let $pr = vba.px[K : x+1]$ and let w be a walk with $w.ba = vba, w.level = x, w.ba = ptba(pr)$ and $w.r = 1$. Then x-fold walk extension of w gives the desired translation:*

$$\begin{aligned} wext^x(w).ba &= T(vba) \\ wext^x(w).r &= r(vba). \end{aligned}$$

A similar argument shows that initial walks with $w.vba \notin V.ba$ hit a not present entry at level y where

$$y = max\{x : vba.px[K : x] \notin P_x\}.$$

3.5 Out of Order Execution

Tomasulo schedulers as shown in Fig. 3 are the standard mechanism for the out of order execution of instructions in processors. Instructions are fetched in program order in the instruction fetch (IF) stage. They wait in the issue stage for a free reservation station (RS) of a functional unit capable of executing the instruction, and for a free slot in the reorder buffer (ROB). From the issue stage instructions proceed to a reservation station. At this point three things happen:

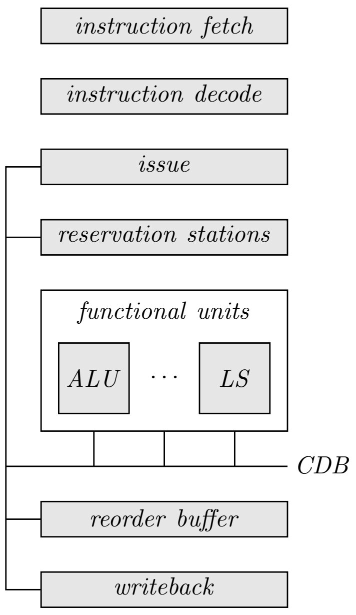

Fig. 3. Tomasulo scheduler

1. The instruction receives a *tag*; this is a local number for instructions issued but not written back. There are as many tags as places in the reorder buffer. The reorder buffer is usually implemented as a RAM implementing a queue that eventually holds the results (including the interrupts produced or sampled) of instructions; at issue time the instruction is inserted at the end of the queue. The natural *tag* to be used for an instruction is its RAM address in the ROB.
2. Register operands are looked up in the register files. If a register does not contain valid data (because an instruction writing the data is in flight), a *tag* field associated with the register contains the tag t of the last such instruction.
3. The results of such instructions with tag t are searched in the ROB and on the common data bus (CDB). If not all operands are found, the instructions producing the desired results are still in the reservation stations or the functional units. The reservation station snoops on the common data bus for the results of instructions with tags t occuring on the CDB. Once all operands are gathered and the functional unit can accept a new operand, instructions proceed to the functional unit, then later via the CDB to the ROB. They are written back when they are at the head of the queue implemented by the ROB. Thus, retirement of instruction is again in program order.

The classical correctness statement of out of order mechanisms then has the form

Lemma 11. *The mechanism of a Tomasulo scheduler (as shown in Fig. 3) preserves the sequential semantics of machine instructions that do not access memory; in these situations reservation stations and reorder buffer are invisible to the programmer.*

A (hopefully) reasonable paper and pencil proof can be found in [19]. There are numerous formal proofs for this result at various levels of detail. At the most detailed level the proof concerns synthesizable hardware [20,27].

Load Store Units. If functional units include load store units LS accessing a memory system m (they should for all practical purposes!) a few extra precautions have to be taken: as long as the functional units do not produce irreversible results, an instruction that has not passed the head of the ROB and thus has not reached the write back stage can be rolled back. This permits to implement precise interrupts (i.e. interrupts with a sequential semantics). But write instructions in memory units (and read instructions to devices with read side effects) cannot easily be rolled back once they have reached the load store unit. One possible way to maintain precise interrupts is to send a write instruction (and a load instruction to an I/O port) to the load store unit only if it is at the head of the ROB.

One often inserts a store buffer between the load store unit and the memory system (see Fig. 4). Because of Lemma 2 the resulting memory system behaves like a single memory and one gets

Lemma 12. *The memory unit shown in Fig. 4 preserves the sequential semantics of machine instructions; thus reservation stations, reorder buffer and store buffer are invisible to the programmer.*

A formal proof for synthesizable hardware is reported in [27].

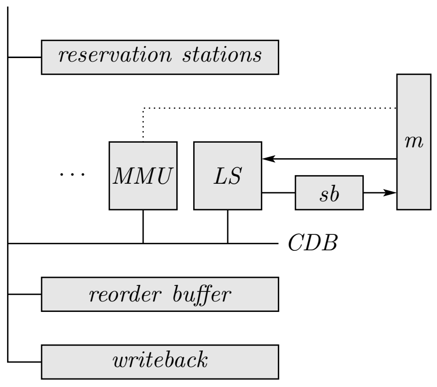

Fig. 4. Memory units

Memory Management Units. Intuitively, the control of a processor with a memory management unit has to split translated loads and stores into two microoperations:

i) find the address translation either by quickly looking it up in the TLB or by slowly walking the page tables, and
ii) perform the memory transaction using the translated address of step i) as an operand.

Tomasulo schedulers permit to implement this in a natural way. MMU and load store unit are separate functional units; if a translated memory transaction is fetched, two microinstructions are issued: one to the MMU computing the desired translation and a second one to the LS unit performing the actual access. For MMUs setting 'accessed' and 'dirty' bits accesses to the tables are potentially writing. Therefore a conservative implementation would perform the accesses only if the microinstruction computing the translation is at the head of the ROB (slowing down the process of page table walking even further). Notice that it is very natural, that a load instruction whose translation is already in the TLB overtakes a previous access whose address needs to be translated by walking. As a result the ROB entries of such load instructions contain kind of precomputed results; we deal with the problems arising from this in the multi processor case shortly.

The data path used by the MMU deserves some attention. One can provide a separate access path bypassing the store buffer from the MMU to the memory system m. Also one can forward results from the MMU directly to the LS unit. Note that due to the different access path into the memory system even in the case of a single processor Lemma 2 does not apply any more. Thus we get

Lemma 13. *The memory unit shown in Fig. 4 with MMU bypassing the store buffer almost preserves the sequential semantics of translated machine instructions: reservation stations and reorder buffer are invisible to the programmer, but the store buffer stays visible.*

Thus, a page table walk might miss a sequentially earlier page table update which is still in the store buffer.

Coherency Snoops. One would fear that in the multiprocessor case the programmer model becomes even more complicated, but in patents like [22] one finds counter measures. One of them is coherency snooping: the ROB entries of processor j holding (precomputed) results of load instructions (i, j) store also the translated address a and participate in the snooping protocol of the caches. If a write to address a is snooped on the cache of a different processor j' (sequentially earlier writes on processor j are handled by store buffer forwarding) the load instruction is either

– rolled back and repeated; this allows other processors to prevent the termination of the load instruction by repeated writes to address a, or

– the result of the load instruction is replaced by the data written by processor j' to address a; this gives the memory model provided by modern processors.

Lemma 14. *Suppose the memory unit shown in Fig. 4 is used with several processors connected to a cache coherent shared memory m and uses coherency snooping. Then locally sequential semantics is almost preserved; reservation stations and reorder buffers are invisible. Store buffers are visible.*

For the proof we use notation from section 3.3. Let t be the last cycle when a read transaction (i, j) is in the ROB before it is retired, and let $m(h^t)$ be the memory simulated by the memory system in cycle t as defined in Equation 2. The memory depends only on the write operations which have completed until cycle t; let $z = M(t)$. Write operations by the LS unit and the MMU are issued on each processor only when they are at the head of the ROB, i.e. they are issued in order. Thus memory $m(h^t)$ already corresponds to an in order execution on each processor and we have as before

$$m(h^t)(a) = \begin{cases} data(last(a, z)) & \text{if } W(a, z), \\ h.mm^0(a) & \text{otherwise.} \end{cases}$$

Let $t' \leq t$ be the last cycle before t when the data for read transaction (i, j) was updated in the ROB, either by the execution of the load instruction or subsequently by the coherency snooping. Then the ROB writes back result $m(h^{t'})(ad(i, j))$ for transaction (i, j). Between cycles t' and t coherency snooping does not update the ROB for transaction (i, j), hence no write to address $ad(i, j)$ even started in this period and we have

$$\begin{aligned} m(h^{t'})(ad(i, j)) &= m(h^t)(ad(i, j)) \\ &= data(last(ad(i, j), z)). \end{aligned}$$

3.6 Initializing an x64 Processor

Figure 5 gives a a very schematic view of the instruction set architecture of contemporary PC processors as documented on about 3000 pages in [10] or only about 1500 pages in [11]. The major blocks are the processor core, MMU with TLB, memory system with main memory and caches, store buffers as well as the I/O devices which are accessed like the main memory. Multi core processors have several processor cores, MMUs and store buffers connected to one memory system and the devices.

Boxes labeled *acc* stand for memory 'access registers' holding addresses, data, etc. of memory transactions. The core contains numerous user registers R as well as numerous system registers; for us system register *CR3* which serves as the origin of TLB walks is particularly important. The segmentation mechanism is a legacy feature going back to the x86 architecture. It can be made invisible by configuring the entire physical address space as a single segment with no restriction of rights. The memory can be accessed in many memory modes. At

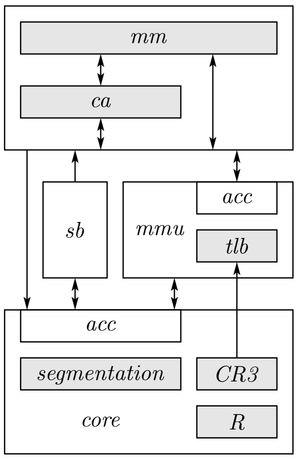

Fig. 5. x64 memory

least one of them (UC – uncachable) completely bypasses the caches and thus makes the caches visible to the programmer. The good news is that the memory modes which do not bypass the caches are compatible. Thus, if no I/O devices are accessed and only compatible memory modes are used, then by Lemmas 9 and 14 the user sees a sequentially consistent physical memory PM and store buffers (see Fig. 6-a). If I/O devices are accessed in uncachable mode life is simple too. But if devices are accessed in memory modes using the cache, than the actual device access only takes place in case of cache misses. This even makes the states of the cache lines visible.

After a reset signal x64 machines are in a very simple operation mode where only a single processor is running and paging is switched off. Because paging is switched off the MMU is not visible. Because only one processor is running by Lemma 2 the store buffer is not visible. In this mode page tables as specified in section 3.4 can be written into the physical memory (Fig. 6-a). If we turn on all processors, clear all TLBs and enable paging (i.e. we run the processors in translated mode, then by Lemma 10 an abstract translation is realized. In this mode the MMUs become invisible and the users see store buffers and a sequentially consistent shared 'linear' memory LM (Fig. 6-b).

4 Programming Language

4.1 Naive Parallel C Semantics

We are considering parallel C programs whose threads run in an interleaved fashion on multi core machines. The obvious approach to define the semantics of such programs is to start with the small step semantics of single threads and then to interleave the steps of the threads. The configuration c of a single threaded abstract C machine can be defined essentially using the following components:

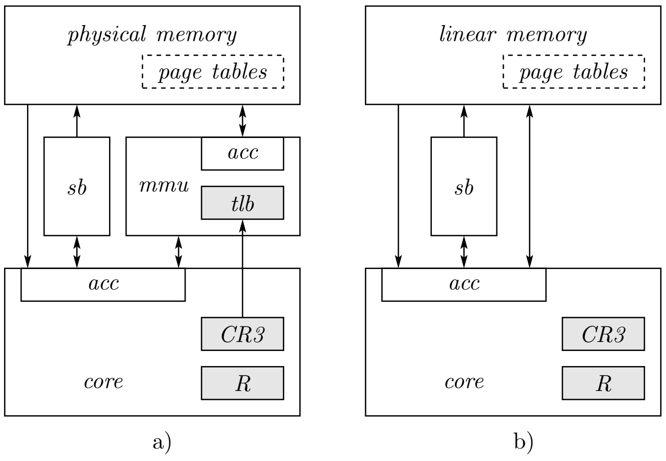

Fig. 6. x64 memory

- a program rest *c.pr* consisting of a sequence of C instructions yet to be executed,
- a global memory *c.gm*,
- a heap memory *c.hm*, and
- a local memory stack *c.lms* consisting of *c.rd* (recursion depth) many memory frames.

For details of the particular semantics used in the Verisoft project[9] see [3, 4]. Generalizing this to a configuration with multiple threads i is straight forward:

- use for each thread i a local program rest $c.pr[i]$ and a local memory stack $c.lms[i]$,
- share the global memory *c.gm* and the heap *c.hm*, and
- now interleave the (small step semantics) steps of the threads defined in this way.

There is no way to beat the elegance of this definition. Unfortunately we don't know how to implement it efficiently. For threads i let $p(i)$ be the program of thread i. Before running on a parallel machine C programs $p(i)$ are first compiled to a machine program $code(p(i)))$; even with the simplest non optimizing compiler a single small step semantics step is usually translated to several machine instructions. The hardware then interleaves the machine instructions instead of the steps of the C semantics. Hence if one wants to define efficiently implementable parallel C semantics one has to worry about the process of compilation, preferably by an optimizing compiler.

[9] `http://www.verisoft.de`

4.2 Compilation

The compiled programs $code(p(i))$ – running say in linear memory $LM(h)$ of a hardware configuration h – have to simulate the programs $p(i)$ of the C threads. For now we only sketch compiler correctness for a single thread. A simulation relation $consis(c, alloc, h)$ between C configurations c and hardware configurations h is defined with the help of an allocation function $alloc$. This functions maps elementary C variables x to 'allocated (linear) base addresses' $alloc(c, x)$. We define some typical properties of relation $consis$.

- For variables x let $asize(x)$ be the number of bytes allocated by the compiler for variable x; it depends only on the type of x. Let $va(c, x)$ be the value of variable x in configuration c. Then the $asize(x)$ bytes in linear memory $LM(h)$ should coincide with the C value of the variable
$$LM_{asize(x)}(alloc(c, x)) = va(c, x).$$
- Suppose p is a pointer; thus its value is another variable $va(c, p) = y$. Assume we have 8 byte addresses. Then the 8 bytes in linear memory following $alloc(c, p)$ should be the allocated base address of y
$$LM_8(alloc(c, p)) = alloc(c, y).$$
- For non optimizing compilers code is compiled statement by statement. The first statement of the program rest is $head(c.pr)$. It is translated to $code(head(c.pr))$. Let $start(code(head(c.pr)))$ be the address in linear memory where this piece of translated code is allocated. Then the program counter $h.pc$ should point there
$$h.pc = start(code(head(c.pr))).$$
Clearly, one needs to modify this condition for optimizing compilers.

For non optimizing compilers one obtains the following step by n-step simulation theorem

Lemma 15. *For every C computation c^0, c^1, ... there exist i) a hardware computation h^0, h^1, ..., ii) a sequence of step numbers s^0, s^1, ... and iii) a sequence of allocation functions $alloc^0$, $alloc^1$, ... such that*
$$consis(c^T, alloc^T, h^{s(T)})$$
holds for all T.

For a formal proof of this result see e.g. [4]. Optimizing compilers exploit the fact, that we do not really care for simulations to hold for every C step. It suffices if the relation holds for the 'visible' C steps T, for example when the program does I/O. Let us call these steps *I/O steps.* Then a possible correctness statement for an optimizing compiler would look like:

Lemma 16. *For every C computation c^0, c^1, … there exist i) a hardware computation h^0, h^1, …, ii) a sequence of step numbers s^0, s^1, … and iii) a sequence of allocation functions $alloc^0$, $alloc^1$, … such that*

$$consis(c^T, alloc^T, h^{s(T)})$$

holds for all I/O steps T.

For a formal correctness proof for an optimizing compiler (with respect to a big step semantics) see [23].

4.3 Volatile Variables

Compiler directives allow to declare shared variables x as volatile. Intuitively speaking this warns the compiler that these variables are shared and thus accesses to such variables should not be optimized to registers. In order to make compiler construction easy we syntactically restrict the use of volatile variables x. A thread can only perform assignments of the form

$$y = x \text{ or } x = y$$

where y is a thread local variable. We include any such assignment into the I/O steps and we implement any such assignment by a fenced read resp. write. By the trivial Lemma 3 then the store buffers in (Fig. 6-b) become invisible for transactions involving volatile variables and we obtain

Lemma 17. *The portion of memory allocated to volatile variables forms a sequentially consistent portion of linear memory LM.*

4.4 Synchronized Parallel C

Using test and set operations on volatile variables it is straightforward to implement locks using textbook shared memory algorithms [29]. Using locks one can exclusively reserve memory regions R of the shared C variables (e.g. certain data structures) temporarily to threads i, for certain intervals I of C-steps. During such intervals I thread i can do computations on region R like in ordinary sequential C computations: due to the locking no other thread accesses region R during interval I, thus the store buffers are by Lemma 2 invisible. However, at the end of interval I when the lock is released the compiler must guarantee that the updates of region R performed by thread i become visible to the other processors. If the compiler treats a lock release of thread i as an I/O step for the thread, then this is guaranteed by Lemma 16.

Currently we work on extensions of these basic programming disciplines and fencing policies for shared memory accesses to cover more programming idioms by our theory framework.

References

1. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Corp. (2009)
2. In der Rieden, T., Paul, W.J.: Beweisen als Ingenieurwissenschaft: Verbundprojekt Verisoft (2003–2007). In Reuse, B., Vollmar, R., eds.: Informatikforschung in Deutschland. Springer (2008) 321–326
3. Leinenbach, D., Petrova, E.: Pervasive compiler verification – From verified programs to verified systems. In: 3rd intl Workshop on Systems Software Verification (SSV 2008). Volume 217C of Electronic Notes in Theoretical Computer Science., Elsevier Science B.V. (2008) 23–40
4. Leinenbach, D.C.: Compiler Verification in the Context of Pervasive System Verification. PhD thesis, Saarland University, Computer Science Department (July 2008)
5. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technical University of Munich (April 2006)
6. Schirmer, N.: A verification environment for sequential imperative programs in Isabelle/HOL. In Baader, F., Voronkov, A., eds.: Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004. Volume 3452., Springer (2005) 398–414
7. Beyer, S., Jacobi, C., Kroening, D., Leinenbach, D., Paul, W.: Putting it all together: Formal verification of the VAMP. International Journal on Software Tools for Technology Transfer **8**(4–5) (August 2006) 411–430
8. Dalinger, I., Hillebrand, M., Paul, W.: On the verification of memory management mechanisms. In Borrione, D., Paul, W., eds.: Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005). Volume 3725., Springer (2005) 301–316
9. Tverdyshev, S., Shadrin, A.: Formal verification of gate-level computer systems. In Rozier, K.Y., ed.: LFM 2008: Sixth NASA Langley Formal Methods Workshop. NASA Scientific and Technical Information (STI), NASA (2008) 56–58
10. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual: Volumes 1–3b. (2009)
11. Advanced Micro Devices (AMD), Inc.: AMD64 Architecture Programmer's Manual: Volumes 1–3. (September 2006)
12. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer **29**(12) (1996) 66–76
13. Steinke, R.C., Nutt, G.J.: A unified theory of shared memory consistency. CoRR **cs.DC/0208027** (2002)
14. Sarkar, S., Sewell, P., Nardelli, F.Z., Owens, S., Ridge, T., Braibant, T., Myreen, M.O., Alglave, J.: The semantics of x86-cc multiprocessor machine code. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2009) 379–391
15. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-tso. In: 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), Springer (2009) To appear.
16. Intel: Intel 64 architecture memory ordering white paper. SKU 318147-001 (2007)
17. Midkiff, S.P., Lee, J., Padua, D.A.: A compiler for multiple memory models. Concurrency and Computation: Practice and Experience **16**(2-3) (2004) 197–220

18. Sevcík, J., Aspinall, D.: On validity of program transformations in the java memory model. In: ECOOP. (2008) 27–51
19. Müller, S.M., Paul, W.J.: Computer Architecture: Complexity and Correctness. Springer (2000)
20. Kröning, D.: Formal Verification of Pipelined Microprocessors. PhD thesis, Saarland University (2001)
21. Sweazey, P., Smith, A.J.: A class of compatible cache consistency protocols and their support by the ieee futurebus. In: ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture, Los Alamitos, CA, USA, IEEE Computer Society Press (1986) 414–423
22. Intel: Us patent 6687809 - maintaining processor ordering by checking load addresses of unretired load instructions against snooping store addresses (2004)
23. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd symposium Principles of Programming Languages, ACM Press (2006) 42–54
24. Smith, A.J.: Cache memories. ACM Comput. Surv. **14**(3) (1982) 473–530
25. Stenström, P.: A survey of cache coherence schemes for multiprocessors. IEEE Computer **23**(6) (1990) 12–24
26. Pong, F., Dubois, M.: Verification techniques for cache coherence protocols. ACM Computing Surveys **29**(1) (1997) 82–126
27. Sawada, J., Jr., W.A.H.: Results of the verification of a complex pipelined machine model. In Pierre, L., Kropf, T., eds.: CHARME. Volume 1703 of Lecture Notes in Computer Science., Springer (1999) 313–316
28. Lamport, L.: How to make a correct multiprocess program execute correctly on a multiprocessor. IEEE Trans. Comput. **46**(7) (1997) 779–782
29. Taubenfeld, G.: Synchronization Algorithms and Concurrent Programming. Pearson / Prentice Hall (2006)