Memory Access Semantics
for a
Multiprocessor Instruction Set

Mike Gordon	??? ??? ???
Computer Laboratory	??? ??? ???
New Museums Site	??? ??? ???
Pembroke Street	??? ??? ???
Cambridge CB2 3QG	??? ??? ???

Contents

1 Introduction

Multiprocessor architectures, such as Digital's Alpha [4] and Sun's SPARC [1], consist of a number of processors that read and write a shared global memory. The documents that define them usually specify the behaviour of the shared memory via constraints on the sets of reads and writes. Formalisms such as process algebra or temporal logic, that are popular in the verification community, are not used. There is thus a 'gap' between the models found in architecture manuals and those underlying standard verification technologies. The work presented here is a start at bridging this gap. A semantics for machine instructions is formulated in terms of constraints on sets of memory references. This is then related to standard state machine models. It is hoped that this will enable state-of-the-art techniques for reasoning about object code (e.g. [5]) to be applied to programs running on multiprocessor architectures.

Although the work described here started out as specifically targeted at Digital's Alpha architecture, the various simplifications that have been made make the results more widely applicable. For example, the generic machine instructions used, although very simple, have as instances instructions found in other processors.

2 Summary of set theory and logic

The application of a function f to an argument x will be written as $f\ x$. If f is 'curried' then its application to n arguments $x_1, \ldots, x_n$ is written $f\ x_1\ \ldots\ x_n$. An application of the form $f(x_1, \ldots, x_n)$ is regarded as the application of f to a single tuple argument $(x_1, \ldots, x_n)$.

Predicates are regarded as functions that return the truth-values T and F. For example, the set membership predicate $\in$ is defined by: $u \in S\ =\ \mathsf{T}$ if u is a member of S and $u \in S\ =\ \mathsf{F}$ if u is not a member of S; $\in$ is regarded as a function taking two arguments and returning a truth value (following convention, it is written infixed).

Sets will normally be written in a slanted font. For example, *num*, *bool* and *string* denote the sets of truth values, natural numbers and strings, respectively. $\mathsf{Finite}\ S$ means S is finite. The empty set is denoted by $\{\ \}$. If S_1 and S_2 are sets, then $S_1 \times S_2$ denotes the Cartesian product of S_1 and S_2 (i.e. the set of all pairs (x_1, x_2), where $x_1 \in S_1$ and $x_2 \in S_2$) and $(S_1 \to S_2)$ denotes the set of all total functions from S_1 to S_2. The union of S_1 and S_2 is $S_1 \cup S_2$.

The lambda-notation $\lambda x.\ \mathcal{E}[x]$ denotes the function (which is sometimes written as $x \mapsto \mathcal{E}[x]$) that mape x to $\mathcal{E}[x]$. For example, $\lambda x.\ 1$ is a function that returns 1 on all arguments and $\lambda x.\ x + 1$ adds 1 to its argument.

The Hilbert epsilon notation $\varepsilon x.\ \mathcal{P}[x]$ (where $\mathcal{P}[x]$ is a truth-valued term) denotes an arbitrarily chosen value a such that $\mathcal{P}[a]$ holds. If $\mathcal{P}[x] = \mathsf{F}$ for all values of x, then a is chosen arbitrarily.

Functions and predicates can be applied to other functions and predicates and return them as results (i.e. higher order predicate calculus is used).

The symbols used for the logical operators are: $\neg$ (negation – "not), $\wedge$ (conjunction – "and"), $\vee$ (disjunction – "or"), $\Rightarrow$ (implication – "implies"), $\forall$ (universal quantification – "for all") and $\exists$ (existential quantificatioEn – "there exists"). Quantifications may be restricted to a set S using the notations $\forall x \in S$ and $\exists x \in S$. The notation $(b \longrightarrow x \mid y)$ means "if b then x else y".

A definition of c has the form $\mathsf{c}\ x_1\ \ldots x_n \stackrel{\mathrm{def}}{=} \mathcal{E}$, where $\mathcal{E}$ is some expression involving $x_1, \ldots, x_n$. The notation $\vdash \mathcal{F}$ means that $\mathcal{F}$ follows from the definitions that precede it. No proofs are given is this paper, but all theorems have been proved with the HOL system [3].

3 A simple generic instruction set

For simplicity, it is assumed that each location in shared memory is represented by a string (think of this as its symbolic name) and that the value of each location is a natural number. Thus a shared memory state is a function of in the set (*string* $\rightarrow$ *num*). These particular representations are somewhat arbitrary and the results in this paper do not really depend on them. In actual implementations locations might, for example, be restricted to be strings of 0s and 1s of length 64, and values might be restricted to be numbers mod 2^{64}.

The local state of each processor contains a program counter *pc*, which is represented by a natural number, and a register file *rg*, which is assumed to be a member of a given set *registers*. For example, if there were 64 quadword registers (as in Alpha) then members of *registers* could be 64-tuples of 64-bit words; however for most of this paper *registers* can be left arbitrary.

Example 1:
A model of registers that will be used in examples is *registers* = (*string* $\rightarrow$ *num*) (i.e. register names are represented by strings and their values by numbers).

Note that the state of the shared memory is also represented by a function that maps locations to values. However, in this example the string arguments of the functions represent register names, but when a function represents a state of the shared memory the string argument represents a location. **End of Example 1.**

First the instruction set is given, then each instruction is explained in detail. Note that the instructions are parameterized on various functions and relations.

Instruction	**Meaning**
`Load` x *lrel*	Read x; use resulting value and *lrel* to determine new values of registers
`Store` x *sfn*	Apply *sfn* to registers; store result in x
`Barrier`	Constrain order of memory accesses
`Branch` *pred sfn*	If *pred* is true of registers then branch to result of applying *sfn*
`Local` *rel*	Modify registers to satisfy *rel*

3.1 Load instructions

The `Load` instruction takes two arguments: (i) a location x from which to read a value in shared memory, and (ii) a relation $\mathit{lrel} \in (\mathit{value} \rightarrow (\mathit{registers} \rightarrow (\mathit{registers} \rightarrow \mathit{bool})))$ that specifies what to do with the resulting value. More specifically, if the value of x in shared memory is a and the state of the registers is rg, then executing `Load` x *lrel* results in a new register state rg' such that $\mathit{lrel}\ a\ rg\ rg' = \mathsf{T}$.

Example 2:
Suppose that registers are modelled as in Example 1 by functions from *string* to *num*. The instruction `LD` x r that reads location x in shared memory and stores the result in register r is defined by:

$$\mathtt{LD}\ x\ r \ \stackrel{\text{def}}{=}\ \mathtt{Load}\ x\ (\lambda a\ \mathit{regs}\ \mathit{regs}'.\ \mathit{regs}' = \mathsf{Update}\ r\ a\ \mathit{regs})$$

where `Update` is a function that updates a given register in a register bank; it is defined by:

$$\mathsf{Update}\ r\ a\ \mathit{regs}\ \stackrel{\text{def}}{=}\ \lambda r'.\ (r = r' \longrightarrow a \mid \mathit{regs}\ r')$$

End of Example 2.

Note that `Load` x *lrel* may be nondeterministic, but this generality is not used here. However, it might be useful to consider cases in which there is no successor state for some read values, i.e. for some values of a and rg to use an *lrel* such that $lrel\ a\ rg\ rg' = \mathsf{F}$ for all values of rg' (i.e. the processor halts).

Example 3:
Assume registers as in Example 1. The instruction `LDHALT` x *pred* that reads location x in shared memory and then halts if the resulting value satisfies *pred* is defined by:

$$\texttt{LDHALT}\ x\ pred \stackrel{\text{def}}{=} \texttt{Load}\ x\ (\lambda a\ regs\ regs'.\ \neg(pred\ a))$$

End of Example 3.

3.2 Store instructions

The `Store` instruction takes two arguments: (i) a location x in shared memory to write to, and (ii) a function $sfn \in (registers \to num)$ that specifies what value to write. If the state of the registers is rg, then executing `Store` x *sfun* writes $sfun\ rg$ to location x; the registers are unchanged.

Example 4:
The instruction `ST` $x\ a$ that writes location x in shared memory with value a is defined by:

$$\texttt{ST}\ x\ a \stackrel{\text{def}}{=} \texttt{Store}\ x\ (\lambda regs.\ a)$$

End of Example 4.

Example 5:
Suppose that registers are modelled as in Example 1. The instruction `STR` $x\ r$ that writes location x in shared memory with the value of register r is defined by:

$$\texttt{STR}\ x\ a \stackrel{\text{def}}{=} \texttt{Store}\ x\ (\lambda regs.\ regs\ r)$$

End of Example 5.

3.3 Barrier instruction

The `Barrier` instruction forces all read/write accesses issued before it to take place before any read/write accesses issued after it.

Example 6:
Let `x` and `y` be different locations. Consider the sequence:

```
LD x R
Barrier
ST y a
```

The `Barrier` instruction forces the read from shared memory generated by `LD x R` to be before the write generated by `ST y a`. Without the `Barrier` these two memory accesses could be performed in arbitrary order. **End of Example 6.**

3.4 Branch instructions

The `Branch` instruction takes two arguments: (i) a predicate $pred \in (registers \rightarrow bool)$ to determine whether to jump, and (ii) a function $sfn \in (registers \rightarrow num)$ to determine where to jump to if $pred$ is true. If the register state is rg then executing `Branch` $pred\ sfun$ has no effect if $rg = \mathsf{F}$; however if $pred\ rg = \mathsf{T}$ then the program counter is set to $sfun\ rg$.

Example 7:
The instruction `JMP` n that causes an unconditional jump to the nth instruction is defined by:

$$\texttt{JMP}\ n \stackrel{\rm def}{=} \texttt{Branch}\ (\lambda regs.\ \mathsf{T})\ (\lambda regs.\ n)$$

End of Example 7.

3.5 Local instructions

The `Local` instruction takes one argument: a relation $rel \in (registers \rightarrow (registers \rightarrow bool))$ that specifies a change to the contents of the registers. If the state of the registers is rg, then executing `Local` rel results in a new register state rg' such that $rel\ rg\ rg' = \mathsf{T}$.

Example 8:
Suppose that registers are modelled as in Example 1. The instruction `ADD` $r_1\ r_2\ r$ that adds the contents of registers r_1 and r_2 and puts the result in register r is defined by:

$$\texttt{ADD}\ r_1\ r_2\ r \stackrel{\rm def}{=} \texttt{Local}\ (\lambda regs\ regs'.\ regs' = Update\ r\ (regs\ r_1 + regs\ r_2)\ regs)$$

End of Example 8.

Example 9:
The instruction `HALT` that halts the processor is defined by:

$$\texttt{HALT} \stackrel{\rm def}{=} \texttt{Local}\ (\lambda regs\ regs'.\ \mathsf{F})$$

End of Example 9.

Example 10:
The following two-processor example is based on a suggestion by Jim Saxe:

Program for Processor 1 (P1)	**Program for Processor 2 (P2)**
`ST x` v	`ST flag` 0
`Barrier`	`LD flag R1`
`ST flag` 1	`Barrier`
	`LD x R2`

The program running on P1 stores v in location `x` of the shared memory; it then executes an `Barrier` and then it stores 1 into location `flag`. The memory barrier instruction `Barrier` ensures that the write generated by `ST x` v is performed before the write generated by `ST flag` 1.

The program running on P2 first stores 0 into location `flag`, then loads the value of `flag` into its local register `R1`. (Note that the value read is not necessarily the 0 just written, because the write generated by `ST flag` 1 on P1 might be executed after the write generated by `ST flag` 0 on P2.) Next a memory barrier is executed on P2 and finally the value in location `x` of shared memory is loaded into P2's register `R2`. Because of the preceding `Barrier`, the read generated by this load is constrained by the architecture to happen after the references generated by the two instructions preceding the `Barrier`.

If P1 and P2 start up running the two programs in parallel, then the results are not completely predictable. However, the following must hold (assuming no other processors are accessing the shared memory):

> If the value in P2's register `R1` in the final state is 1, then the value in P2's register `R2` in the final state must be v.

End of Example 10.

4 General semantics of the instruction set

As each processor executes a program it issues a sequence of memory references consisting of:

(i) Reads: $\mathsf{R}\ p\ n\ x\ a$;

(ii) Writes: $\mathsf{W}\ p\ n\ x\ a$;

(iii) Memory barriers: $\mathsf{MB}\ p\ n$

where R, W and MB are functions that construct references. They take as arguments a natural number p representing a processor, and a natural number n specifying the order in which the reference is issued. Reads and writes also take a location x in shared memory (a string) and a value read or written a (a natural number).

The execution of an individual instruction is specified by a relation $\mathsf{InstructionStep}$, such that $\mathsf{InstructionStep}\ p\ i\ \mathit{refs}\ s\ s' = \mathsf{T}$ iff executing instruction i on processor p with local state s results in a new local state s' and generates the set refs of memory references. The set refs will be empty if i is a barrier, branch or local instruction; it will be of the form $\{\mathsf{R}\ p\ n\ x\ a\}$ if i is a load and of the form $\{\mathsf{W}\ p\ n\ x\ a\}$ if i is a store.

In order to specify the numbers defining the order in which references are generated (i.e. the *issue sequence*), the local state of each processor will be augmented with a 'reference count' component. Thus the local state of a processor will be a triple (pc, ic, rg) where pc is the program counter (a natural number), ic is the reference count (a natural number) and rg is the register state (a member of *registers*).

$\mathsf{InstructionStep}$ is defined by cases on the five kinds of instructions so as to satisfy:

$$\mathsf{InstructionStep}\ p\ (\mathtt{Load}\ x\ \mathit{lrel})\ \mathit{refs}\ (pc, ic, rg)\ (pc', ic', rg')\ =$$
$$\exists a.\ \mathit{refs} = \{\mathsf{R}\ p\ ic\ x\ a\} \wedge (pc' = pc + 1) \wedge (ic' = ic + 1) \wedge \mathit{lrel}\ a\ rg\ rg'$$

$$\mathsf{InstructionStep}\ p\ (\mathtt{Store}\ x\ \mathit{sfn})\ \mathit{refs}\ (pc, ic, rg)\ (pc', ic', rg')\ =$$
$$\mathit{refs} = \{\mathsf{W}\ p\ ic\ x\ (\mathit{sfn}\ rg)\} \wedge (pc' = pc + 1) \wedge (ic' = ic + 1) \wedge (rg' = rg)$$

$$\mathsf{InstructionStep}\ p\ \mathtt{Barrier}\ \mathit{refs}\ (pc, ic, rg)\ (pc', ic', rg')\ =$$
$$\mathit{refs} = \{\mathsf{MB}\ p\ ic\} \wedge (pc' = pc + 1) \wedge (ic' = ic + 1) \wedge (rg' = rg)$$

$$\mathsf{InstructionStep}\ p\ (\mathtt{Branch}\ \mathit{pred}\ \mathit{sfn})\ \mathit{refs}\ (pc, ic, rg)\ (pc', ic', rg')\ =$$
$$\mathit{refs} = \{\ \} \wedge (pc' = (\mathit{pred}\ rg \longrightarrow \mathit{sfn}\ rg \mid pc + 1)) \wedge (ic' = ic) \wedge (rg' = rg)$$

$$\mathsf{InstructionStep}\ p\ (\mathtt{Local}\ \mathit{rel})\ \mathit{refs}\ (pc, ic, rg)\ (pc', ic', rg')\ =$$
$$\mathit{refs} = \{\ \} \wedge (pc' = pc + 1) \wedge (ic' = ic) \wedge \mathit{rel}\ rg\ rg'$$

5 General semantics of programs

A program will be represented by a list of instructions. The list of length $n+1$ containing the elements $x_0, \ldots, x_n$ will be written as $[x_0; x_1; \ldots; x_n]$. $\mathsf{El}\ i\ l$ is the $(i+1)$th element of l, thus $\mathsf{El}\ i\ [x_0; x_1; \ldots; x_n] = x_i$. The length of a list l is denoted by $\mathsf{Length}\ l$.

If *prog* is a list of instructions and *pc* a number, then $\mathsf{FetchInstruction}\ \mathit{prog}\ pc$ denotes the instruction pointed to by *pc*, or `HALT` (see Example 9) if *pc* doesn't point to an instruction in *prog*.

$$\mathsf{FetchInstruction}\ \mathit{prog}\ pc \stackrel{\text{def}}{=} (pc > (\mathsf{Length}\ \mathit{prog}) \longrightarrow \mathtt{HALT} \mid \mathsf{El}\ pc\ \mathit{prog})$$

Define $\mathsf{ProgSteps}$ so that $\mathsf{ProgSteps}\ p\ \mathit{prog}\ \mathit{refs}\ (pc, ic, rg)\ (pc', ic', rg')$ is true iff whenever program *prog* is executed on processor p with initial local state (pc, ic, rg), then there is a sequence of steps to state (pc', ic', rg'), each of which satisfies $\mathsf{InstructionStep}$, and that generates the set *refs* of references.

Intuitively, $\mathsf{ProgSteps}$ is a kind of transitive closure of $\mathsf{InstructionStep}$ that 'accumulates' memory references. Formally $\mathsf{ProgSteps}$ is the least relation satisfying the following two conditions:

$$\mathsf{InstructionStep}\ p\ (\mathsf{FetchInstruction}\ \mathit{prog}\ pc)\ \mathit{refs}\ (pc, ic, rg)\ (pc', ic', rg')$$
$$\Rightarrow$$
$$\mathsf{ProgSteps}\ p\ \mathit{prog}\ \mathit{refs}\ (pc, ic, rg)\ (pc', ic', rg')$$

$$\mathsf{ProgSteps}\ p\ \mathit{prog}\ \mathit{refs}_1\ (pc, ic, rg)\ (pc', ic', rg') \wedge$$
$$\mathsf{InstructionStep}\ p\ (\mathsf{FetchInstruction}\ \mathit{prog}\ pc')\ \mathit{refs}_2\ (pc', ic', rg')\ (pc'', ic'', rg'')$$
$$\Rightarrow$$
$$\mathsf{ProgSteps}\ p\ \mathit{prog}\ (\mathit{refs}_1 \cup \mathit{refs}_2)\ (pc, ic, rg)\ (pc'', ic'', rg'')$$

Example 11:
Consider the programs discussed in Example 10:

$$\mathsf{Prog}_0(v) \stackrel{\text{def}}{=} [\mathtt{ST\ x}\ v;\ \mathtt{Barrier};\ \mathtt{ST\ flag}\ 1]$$

$$\mathsf{Prog}_1 \stackrel{\text{def}}{=} [\mathtt{ST\ flag}\ 0;\ \mathtt{LD\ flag\ R1};\ \mathtt{Barrier};\ \mathtt{LD\ x\ R2}]$$

It can be shown that:

$$\vdash \mathsf{ProgSteps}\ 0\ (\mathsf{Prog}_0\ v)\ \mathit{refs}_0\ (0,0,\mathit{rg}_0)\ (3,3,\mathit{rg}_0') = \\ (\mathit{refs}_0 = \{\mathsf{W}\ 0\ 0\ \mathtt{x}\ v,\ \mathsf{MB}\ 0\ 1,\ \mathsf{W}\ 0\ 2\ \mathtt{flag}\ 1\}) \wedge \\ (\mathit{rg}_0' = \mathit{rg}_0)$$

$$\vdash \mathsf{ProgSteps}\ 1\ \mathsf{Prog}_1\ \mathit{refs}_1\ (0,0,\mathit{rg}_1)\ (4,4,\mathit{rg}_1') = \\ \exists a1\ a2.\ (\mathit{refs}_1 = \{\mathsf{W}\ 1\ 0\ \mathtt{flag}\ 0,\ \mathsf{R}\ 1\ 1\ \mathtt{flag}\ a1,\ \mathsf{MB}\ 1\ 2,\ \mathsf{R}\ 1\ 3\ \mathtt{x}\ a2\}) \wedge \\ (\mathit{rg}_1' = \mathsf{Update}\ \mathtt{R2}\ a2\ (\mathsf{Update}\ \mathtt{R1}\ a1\ \mathit{rg}_1))$$

End of Example 11.

6 Uniprocessor semantics of programs

In the general semantics of the previous section, the effect of loads and stores was to issue memory accesses. The fact that writes may effect later reads was only specified indirectly via the definition of the before-ordering on the set of all memory references (see Section 7). There is no explicit updating of the shared memory in the semantics.

Although this semantics can be used to reason about programs, it is rather cumbersome (see Example 12 on page 10). An important special case is when there is no communication between processors. When this holds, each processor can be treated as though it were a uniprocessor and much simpler models can be used. In particular, the highly developed technology for reasoning about object code developed by Boyer and Yu [5] is applicable.

In this section a uniprocessor semantics for the instruction set of Section 3 is presented. Its relationship to the general semantics is then considered.

The uniprocessor semantics is a straightforward state-machine. Each step corresponds to the exection of a single instruction. The state consists of the program counter, registers and the shared memory (note that in the uniprocessor case the memory is not really shared as there is only one processor). The effect of a store is to explicity update the shared memory and the value read by a load is obtained by 'looking up' the value of the location in shared memory.

The execution of individual instructions on a uniprocecssor is specified by $\mathsf{UniInstructionStep}$, which is the relation such that $\mathsf{UniInstructionStep}\ i\ (pc, rg, mem)\ (pc', rg', mem') = \mathsf{T}$ iff executing instruction i when the program counter is pc, the register state is rg and the shared memory is mem results in a new state with program counter pc', register state rg' and shared memory state mem'.

The definition of $\mathsf{UniInstructionStep}$ is by cases on the five kinds of instructions. Note that since the shared memory has the same representation as the register model of Example 1, the function Update defined in Example 2 can be used.

$$\mathsf{UniInstructionStep}\ (\mathtt{Load}\ x\ \mathit{lrel})\ (pc, rg, mem)\ (pc', rg', mem') = \\ (pc = pc + 1) \wedge \mathit{lrel}\ (mem\ x)\ rg\ rg' \wedge (mem' = mem)$$

$$\mathsf{UniInstructionStep}\ (\mathtt{Store}\ x\ \mathit{sfn})\ (pc, rg, mem)\ (pc', rg', mem') = \\ (pc' = pc + 1) \wedge (rg' = rg) \wedge (mem' = \mathsf{Update}\ x\ (\mathit{sfn}\ rg)\ mem)$$

$$\mathsf{UniInstructionStep}\ \mathtt{Barrier}\ (pc, rg, mem)\ (pc', rg', mem') = \\ (pc' = pc + 1) \wedge (rg' = rg) \wedge (mem' = mem)$$

$$\mathsf{UniInstructionStep}\ (\mathtt{Branch}\ \mathit{pred}\ \mathit{sfn})\ (pc, rg, mem)\ (pc', rg', mem')\ =$$
$$(pc' = (\mathit{pred}\ rg \longrightarrow \mathit{sfn}\ rg \mid pc + 1)) \wedge (rg' = rg) \wedge (mem' = mem)$$

$$\mathsf{UniInstructionStep}\ (\mathtt{Local}\ \mathit{rel})\ (pc, rg, mem)\ (pc', rg', mem')\ =$$
$$(pc' = pc + 1) \wedge \mathit{rel}\ rg\ rg' \wedge (mem' = mem)$$

Define UniProgSteps so that UniProgSteps $prog\ (pc, rg, mem)\ (pc', rg', mem')$ is true iff whenever program $prog$ is executed with initial state (pc, rg, mem), then there is a sequence of steps to state (pc', rg', mem'), each of which satisfies UniInstructionStep.

$$\begin{array}{l}\mathsf{UniInstructionStep}\ (\mathsf{FetchInstruction}\ prog\ pc)\ (pc, rg, mem)\ (pc', rg', mem') \\ \Rightarrow \\ \mathsf{UniProgSteps}\ prog\ (pc, rg, mem)\ (pc', rg', mem')\end{array}$$

$$\begin{array}{l}\mathsf{UniProgSteps}\ prog\ (pc, rg, mem)\ (pc', rg', mem') \wedge \\ \mathsf{UniInstructionStep}\ (\mathsf{FetchInstruction}\ prog\ pc')\ (pc', rg', mem')\ (pc'', rg'', mem'') \\ \Rightarrow \\ \mathsf{UniProgSteps}\ prog\ (pc, rg, mem)\ (pc'', rg'', mem'')\end{array}$$

7 Shared memory model

The memory model used here is motivated and described in detail in a companion paper [2]. To make this paper self-contained it is summarized below.

A number of auxiliary functions for manipulating references are used.

The components of references can be extracted with the functions Proc, Seq, Var and Val, which are defined so as to satisfy the following equations:

$$\begin{array}{lcllcllcl}
\mathsf{Proc}(\mathsf{R}\ p\ n\ x\ a) & = & p, & \mathsf{Proc}(\mathsf{W}\ p\ n\ x\ a) & = & p, & \mathsf{Proc}(\mathsf{MB}\ p\ n) & = & p, \\
\mathsf{Seq}(\mathsf{R}\ p\ n\ x\ a) & = & n, & \mathsf{Seq}(\mathsf{W}\ p\ n\ x\ a) & = & n, & \mathsf{Seq}(\mathsf{MB}\ p\ n) & = & n, \\
\mathsf{Var}(\mathsf{R}\ p\ n\ x\ a) & = & x, & \mathsf{Var}(\mathsf{W}\ p\ n\ x\ a) & = & x, & & & \\
\mathsf{Val}(\mathsf{R}\ p\ n\ x\ a) & = & a, & \mathsf{Val}(\mathsf{W}\ p\ n\ x\ a) & = & a. & & &
\end{array}$$

The predicates IsR, IsW and IsMB are defined so as to satisfy:

$$\begin{array}{lcllcllcl}
\mathsf{IsR}(\mathsf{R}\ p\ n\ x\ a) & = & \mathsf{T}, & \mathsf{IsR}(\mathsf{W}\ p\ n\ x\ a) & = & \mathsf{F}, & \mathsf{IsR}(\mathsf{MB}\ p\ n) & = & \mathsf{F}, \\
\mathsf{IsW}(\mathsf{R}\ p\ n\ x\ a) & = & \mathsf{F}, & \mathsf{IsW}(\mathsf{W}\ p\ n\ x\ a) & = & \mathsf{T}, & \mathsf{IsW}(\mathsf{MB}\ p\ n) & = & \mathsf{F}, \\
\mathsf{IsMB}(\mathsf{R}\ p\ n\ x\ a) & = & \mathsf{F}, & \mathsf{IsMB}(\mathsf{W}\ p\ n\ x\ a) & = & \mathsf{F}, & \mathsf{IsMB}(\mathsf{MB}\ p\ n) & = & \mathsf{T}.
\end{array}$$

Reference u precedes v in the *processor issue order* if they are issued by the same processor, the sequence number of u is less than the sequence number of v and either u or v is a memory barrier or they are both read/write accesses to the same variable:

$$\begin{array}{l}\mathsf{IssueOrder}\ u\ v\ \stackrel{\mathrm{def}}{=} \\ (\mathsf{Proc}\ u\ =\ \mathsf{Proc}\ v) \wedge (\mathsf{Seq}\ u < \mathsf{Seq}\ v) \wedge (\mathsf{IsMB}\ u \vee \mathsf{IsMB}\ v \vee (\mathsf{Var}\ u = \mathsf{Var}\ v))\end{array}$$

The restriction of IssueOrder to a set *refs* will be needed. Define:

$$\mathsf{Restrict}\ \mathit{refs}\ \mathit{ord}\ u\ v\ \stackrel{\mathrm{def}}{=}\ u \in \mathit{refs} \wedge v \in \mathit{refs} \wedge \mathit{ord}\ u\ v$$

then the restricted issue order is Restrict IssueOrder *refs*.

The processor issue order only orders references from the same processor. A second ordering, the *location access order*, orders references to a given location in shared memory from any processor.

Location access order is only partially specified. It is required to satisfy three properties:

1. Any two accesses to the same location, one of which is a write, must be ordered.
2. Storage:
   (a) If a read u to a location x follows a write v to x, and there is no other intervening write to x with a different value, then the value read in v must be the value written in u.
   (b) If a read u to a location x follows a write v to x and the value read in v is different from the value written in u, then there must exist an intervening write w to x with the value read by v.
3. If u and v are two accesses to the same location, one of which is a write, and u precedes v in the issue order, then u precedes v in the location access order.

To formalize these, first define the auxiliary predicate $\mathsf{AtLeastOneWrite}$ by:

$$\begin{array}{l}\mathsf{AtLeastOneWrite}\ u\ v\ \stackrel{\text{def}}{=}\\ \neg(u = v) \wedge (\mathsf{Var}\ u = \mathsf{Var}\ v) \wedge ((\mathsf{IsR}\ u \wedge \mathsf{IsW}\ v) \vee (\mathsf{IsW}\ u \wedge \mathsf{IsR}\ v) \vee (\mathsf{IsW}\ u \wedge \mathsf{IsW}\ v))\end{array}$$

The first condition, 1, is then formalized by $\mathsf{Linearity}\ \mathit{refs}\ \mathit{ord}$, where:

$$\mathsf{Linearity}\ \mathit{refs}\ \mathit{ord}\ \stackrel{\text{def}}{=}\ \forall u\ v \in \mathit{refs}.\ \mathsf{AtLeastOneWrite}\ u\ v \Rightarrow (\mathit{ord}\ u\ v \vee \mathit{ord}\ v\ u)$$

Part (a) of the second condition, 2, is formalized by $\mathsf{ReadGetsLastWrite}\ \mathit{refs}\ \mathit{ord}$, where:

$$\begin{array}{l}\mathsf{ReadGetsLastWrite}\ \mathit{refs}\ \mathit{ord}\ \stackrel{\text{def}}{=}\\ \forall u\ v \in \mathit{refs}.\\ \quad \mathsf{IsW}\ u \wedge \mathsf{IsR}\ v \wedge (\mathsf{Var}\ u = \mathsf{Var}\ v) \wedge \mathit{ord}\ u\ v\ \wedge\\ \quad \neg(\exists w \in \mathit{refs}.\ \mathsf{IsW}\ w \wedge (\mathsf{Var}\ w = \mathsf{Var}\ u) \wedge \neg(\mathsf{Val}\ u = \mathsf{Val}\ w) \wedge \mathit{ord}\ u\ w \wedge \mathit{ord}\ w\ v)\\ \quad \Rightarrow\\ \quad (\mathsf{Val}\ u = \mathsf{Val}\ v)\end{array}$$

Part (b) is formalized by $\mathsf{ReadRequiresWrite}\ \mathit{refs}\ \mathit{ord}$, where:

$$\begin{array}{l}\mathsf{ReadRequiresWrite}\ \mathit{refs}\ \mathit{ord}\ \stackrel{\text{def}}{=}\\ \forall u\ v \in \mathit{refs}.\\ \quad \mathsf{IsW}\ u \wedge \mathsf{IsR}\ v \wedge (\mathsf{Var}\ u = \mathsf{Var}\ v) \wedge \mathit{ord}\ u\ v \wedge \neg(\mathsf{Val}\ u = \mathsf{Val}\ v)\\ \quad \Rightarrow\\ \quad \exists w \in \mathit{refs}.\ \mathsf{IsW}\ w \wedge (\mathsf{Var}\ w = \mathsf{Var}\ u) \wedge (\mathsf{Val}\ v = \mathsf{Val}\ w) \wedge \mathit{ord}\ u\ w \wedge \mathit{ord}\ w\ v\end{array}$$

The second condition, 2, is the combination of (a) and (b) and is formalized by $\mathsf{Storage}\ \mathit{refs}\ \mathit{ord}$, where:

$$\mathsf{Storage}\ \mathit{refs}\ \mathit{ord}\ \stackrel{\text{def}}{=}\ \mathsf{ReadGetsLastWrite}\ \mathit{refs}\ \mathit{ord} \wedge \mathsf{ReadRequiresWrite}\ \mathit{refs}\ \mathit{ord}$$

The third condition, 3, is formalized by $\mathsf{IssueAccess}\ \mathit{refs}\ \mathit{ord}$, where:

$$\mathsf{IssueAccess}\ \mathit{refs}\ \mathit{ord}\ \stackrel{\text{def}}{=}\ \forall u\ v \in \mathit{refs}.\ \mathsf{AtLeastOneWrite}\ u\ v \wedge \mathsf{IssueOrder}\ u\ v \Rightarrow \mathit{ord}\ u\ v$$

The conditions for an order *ord* to be an access order on a set of references *refs* are thus:

$$\mathsf{IsAccessOrder}\ \mathit{refs}\ \mathit{ord}\ \stackrel{\text{def}}{=}\ \mathsf{Linearity}\ \mathit{refs}\ \mathit{ord} \wedge \mathsf{Storage}\ \mathit{refs}\ \mathit{ord} \wedge \mathsf{IssueAccess}\ \mathit{refs}\ \mathit{ord}$$

An ordering *ord* on a set *refs* of references is a possible *before-ordering* iff there is an access order, *access_order* say, such that:

(i) *ord* is the transitive closure of the union of the issue order restricted to *refs* and *access_order*;

(ii) if u and v are references, then at most one of $ord\ u\ v$ and $ord\ v\ u$ is true.

If R_1 and R_2 are binary relations, then the transitive closure of the union of R_1 and R_2 is the least transitive relation including R_1 and R_2.

This is the relation $\mathsf{Closure}\ R_1\ R_2$ characterized by the four properties:

$$\forall u\ v.\ R_1\ u\ v \Rightarrow \mathsf{Closure}\ R_1\ R_2\ u\ v$$

$$\forall u\ v.\ R_2\ u\ v \Rightarrow \mathsf{Closure}\ R_1\ R_2\ u\ v$$

$$\forall u\ v.\ (\exists w.\ \mathsf{Closure}\ R_1\ R_2\ u\ w \wedge \mathsf{Closure}\ R_1\ R_2\ w\ v) \Rightarrow \mathsf{Closure}\ R_1\ R_2\ u\ v$$

$$\begin{aligned}\forall P.\ & (\forall u\ v.\ R_1\ u\ v \Rightarrow P\ u\ v) \wedge \\ & (\forall u\ v.\ R_2\ u\ v \Rightarrow P\ u\ v) \wedge \\ & (\forall u\ v.\ (\exists w.\ P\ u\ w \wedge P\ w\ v) \Rightarrow P\ u\ v) \\ & \Rightarrow \\ & (\forall u\ v.\ \mathsf{Closure}\ R_1\ R_2\ u\ v \Rightarrow P\ u\ v)\end{aligned}$$

A relation is formally defined to be a 'before relation' on *refs* by:

$$\begin{aligned}&\mathsf{IsBefore}\ refs\ ord\ \stackrel{\text{def}}{=} \\ &\exists access\_order.\ \mathsf{IsAccessOrder}\ refs\ access\_order\ \wedge \\ &\qquad (ord = \mathsf{Closure}\ (\mathsf{Restrict}\ IssueOrder\ refs)\ access\_order) \wedge \\ &\qquad \neg(\exists u\ v \in refs.\ ord\ u\ v \wedge ord\ v\ u)\end{aligned}$$

Theorem 1: If $\mathsf{Trans}\ ord$ means that *ord* is transitive, then:

$$\begin{aligned}\vdash \forall refs\ ord.\ \mathsf{Finite}\ refs \Rightarrow (\mathsf{IsBefore}\ refs\ ord\ =\ & \\ & \mathsf{IsAccessOrder}\ refs\ ord\ \wedge \\ & \mathsf{Trans}\ (\mathsf{Restrict}\ ord\ refs) \wedge \\ & \neg(\exists u\ v \in refs.\ ord\ u\ v \wedge ord\ v\ u))\end{aligned}$$

Because it does not use Closure, this characterization of IsBefore is sometimes easier to work with than the definition given above.

Example 12:
Recall Example 11 concerning the programs:

$$\mathsf{Prog}_0(v) \stackrel{\text{def}}{=} [\texttt{ST x}\ v;\ \texttt{Barrier};\ \texttt{ST flag}\ 1]$$

$$\mathsf{Prog}_1 \stackrel{\text{def}}{=} [\texttt{ST flag}\ 0;\ \texttt{LD flag R1};\ \texttt{Barrier};\ \texttt{LD x R2}]$$

It follows that:

$$\begin{aligned}&\vdash \forall v\ rg_0\ rg_1\ rg_0'\ rg_1'\ refs_0\ refs_1\ ord. \\ &\quad \mathsf{ProgSteps}\ 0\ (\mathsf{Prog}_0\ v)\ refs_0\ (0,0,rg_0)\ (3,3,rg_0') \wedge \\ &\quad \mathsf{ProgSteps}\ 1\ \mathsf{Prog}_1\ \quad refs_1\ (0,0,rg_1)\ (4,4,rg_1') \\ &\quad \Rightarrow \\ &\quad refs_0 \cup refs_1 = \{\begin{array}[t]{ll} \mathsf{W}\ 0\ 0\ \texttt{x}\ v, & \mathsf{W}\ 1\ 0\ \texttt{flag}\ 0, \\ \mathsf{MB}\ 0\ 1, & \mathsf{R}\ 1\ 1\ \texttt{flag}\ a1, \\ \mathsf{W}\ 0\ 2\ \texttt{flag}\ 1, & \mathsf{MB}\ 1\ 2, \\ & \mathsf{R}\ 1\ 3\ \texttt{x}\ a2\} \end{array}\end{aligned}$$

This can be used to prove that if $\mathsf{Prog}_0\ v$ and Prog_1 are executed with a shared memory, and if there is a before-ordering on the resultant set of references, i.e. a before-ordering on the set:

$$
\begin{array}{llll}
\{\mathsf{W} & 0\ 0\ \mathtt{x}\ v, & \mathsf{W} & 1\ 0\ \mathtt{flag}\ 0, \\
\ \mathsf{MB} & 0\ 1, & \mathsf{R} & 1\ 1\ \mathtt{flag}\ a1, \\
\ \mathsf{W} & 0\ 2\ \mathtt{flag}\ 1, & \mathsf{MB} & 1\ 2, \\
 & & \mathsf{R} & 1\ 3\ \mathtt{x}\ a2\}
\end{array}
$$

then if the final value of P2's register `R1` is 1 then the final value of its register `R2` must be v.

$$
\begin{array}{l}
\vdash \forall v\ \mathit{rg}_0\ \mathit{rg}_1\ \mathit{rg}'_0\ \mathit{rg}'_1\ \mathit{refs}_0\ \mathit{refs}\cdot\ \mathit{ord}. \\
\quad \mathsf{ProgSteps}\ 0\ (\mathsf{Prog}_0\ v)\ \mathit{refs}_0\ (0,0,\mathit{rg}_0)\ (3,3,\mathit{rg}'_0) \wedge \\
\quad \mathsf{ProgSteps}\ 1\ \mathsf{Prog}_1\qquad \mathit{refs}_1\ (0,0,\mathit{rg}_1)\ (4,4,\mathit{rg}'_1) \wedge \\
\quad \mathsf{IsBefore}\ (\mathit{refs}_0 \cup \mathit{refs}_1)\ \mathit{ord} \\
\quad \Rightarrow \\
\quad (\mathit{rg}'_1\ \mathtt{R1} = 1) \Rightarrow (\mathit{rg}'_1\ \mathtt{R2} = v)
\end{array}
$$

This is a safety property. The corresponding liveness property is that there always does exist a before-ordering in which the final value of P2's register `R1` is 1. Formally:

$$
\begin{array}{l}
\vdash \forall v\ \mathit{rg}_0\ \mathit{rg}_1. \\
\quad \exists \mathit{rg}'_0\ \mathit{rg}'_1\ \mathit{refs}_0\ \mathit{refs}_1\ \mathit{ord}. \\
\quad\ \mathsf{ProgSteps}\ 0\ (\mathsf{Prog}_0\ v)\ \mathit{refs}_0\ (0,0,\mathit{rg}_0)\ (3,3,\mathit{rg}'_0) \wedge \\
\quad\ \mathsf{ProgSteps}\ 1\ \mathsf{Prog}_1\qquad \mathit{refs}_1\ (0,0,\mathit{rg}_1)\ (4,4,\mathit{rg}'_1) \wedge \\
\quad\ \mathsf{IsBefore}\ (\mathit{refs}_0 \cup \mathit{refs}_1)\ \mathit{ord} \wedge \\
\quad\ (\mathit{rg}'_1\ \mathtt{R1} = 1)
\end{array}
$$

End of Example 12.

A set *refs* of references is *initialized* with respect to a memory state *init* and an ordering *ord* iff every read in *refs* can be 'explained' by *init* or by a preceding write in *refs*.

More precisely: for every read u in *refs*, if u reads from location x, then the value a that is read is either the value of x in *init* or the value written by some write to x in *refs* that precedes the read in the ordering *ord*.

$$
\begin{array}{l}
\mathsf{Init}\ \mathit{init}\ \mathit{refs}\ \mathit{ord}\ \stackrel{\mathrm{def}}{=} \\
\ \forall u \in \mathit{refs}.\ \mathsf{IsR}\ u \Rightarrow (\mathsf{Val}\ u = \mathit{init}(\mathsf{Var}\ u)) \\
\qquad\qquad\qquad\qquad\quad \vee \\
\qquad\qquad\qquad\qquad\quad \exists v \in \mathit{refs}.\ \mathsf{IsW}\ v \wedge (\mathsf{Var}\ v = \mathsf{Var}\ u) \wedge (\mathsf{Val}\ v = \mathsf{Val}\ u) \wedge \mathit{ord}\ v\ u
\end{array}
$$

8 Relationships between the two semantics

In the general semantics of section 4 the state of the shared memory is not explicit, but is implicit in the history of reads and writes that have occurred. If *refs* is a set of references, *ord* a before-ordering on *refs* and *init* an initial state, then the final state determined by *refs*, *ord* and *init* is denoted by $\mathsf{Final}\ \mathit{refs}\ \mathit{ord}\ \mathit{init}$.

The function Final is defined using four auxiliary predicates: IsLast, IsLastWrite, IsFinalLocVal and IsFinal.

$\mathsf{IsLast}\ \mathit{refs}\ p\ \mathit{ord}\ u$ is true iff u is the last (with respect to *ord*) member of the set *refs* satisfying the predicate p.

$$
\mathsf{IsLast}\ \mathit{refs}\ p\ \mathit{ord}\ u\ \stackrel{\mathrm{def}}{=}\ u \in \mathit{refs} \wedge p\ u \wedge \forall v \in \mathit{refs}.\ p\ v \Rightarrow ((v = u) \vee \mathit{ord}\ v\ u)
$$

$\mathsf{IsLastWrite}\ refs\ ord\ x\ u$ is true iff u is the last (with respect to ord) write access to x in $refs$.

$$\mathsf{IsLastWrite}\ refs\ ord\ x\ u \stackrel{\text{def}}{=} \mathsf{IsLast}\ refs\ (\lambda u.\ \mathsf{IsW}\ u \wedge (\mathsf{Var}\ u = x))\ ord\ u$$

$\mathsf{IsFinalLocVal}\ refs\ ord\ s\ x\ a$ is true iff a is the last value (with respect to the ordering ord) written to x by a write in $refs$. If there is no write to x in $refs$, then a is the initial value of x, i.e. $init\ x$

$$\begin{array}{l} \mathsf{IsFinalLocVal}\ refs\ ord\ init\ x\ a \stackrel{\text{def}}{=} \\ \quad ((\exists w \in refs.\ \mathsf{IsW}\ w \wedge (\mathsf{Var}\ w = x)) \longrightarrow \exists w \in refs.\ \mathsf{IsLastWrite}\ refs\ ord\ x\ w \wedge (a = \mathsf{Val}\ w) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \mid\ \ (a = init\ x)) \end{array}$$

$\mathsf{IsFinal}\ refs\ ord\ s\ s'$ is true iff s' is the memory state resulting from performing the memory accesses in $refs$ on the state s. More precisely, the value of every location in s' is the last value written to the location by a reference in $refs$.

$$\mathsf{IsFinal}\ refs\ ord\ s\ s' \stackrel{\text{def}}{=} \forall x.\ \mathsf{IsFinalLocVal}\ refs\ ord\ s\ x\ (s'\ x)$$

Theorem 2: If ord is a before-ordering on $refs$ then if a final state exists, it is unique:

$$\begin{array}{l} \vdash \forall refs\ ord. \\ \quad \mathsf{IsBefore}\ refs\ ord \\ \quad \Rightarrow \\ \quad \forall init\ s_1\ s_2.\ \mathsf{IsFinal}\ refs\ ord\ init\ s_1 \wedge \mathsf{IsFinal}\ refs\ ord\ init\ s_1 \Rightarrow (s_1 = s_2) \end{array}$$

If, in addition, $refs$ is finite and is initialized with respect to ord and $init$, then a final state always exists.

Theorem 3:

$$\begin{array}{l} \vdash \forall init\ refs\ ord. \\ \quad \mathsf{Init}\ init\ refs\ ord \wedge \mathsf{IsBefore}\ refs\ ord \wedge \mathsf{Finite}\ refs \Rightarrow \exists s.\ \mathsf{IsFinal}\ refs\ ord\ init\ s \end{array}$$

The function Final can now be defined by:

$$\mathsf{Final}\ refs\ ord\ init \stackrel{\text{def}}{=} \varepsilon s.\ \mathsf{IsFinal}\ refs\ ord\ init\ s$$

Theorem 4: If there is only one processor, then the general semantics is equivalent to the uniprocessor semantics in the following sense:

$$\begin{array}{l} \vdash \forall p\ prog\ pc\ ic\ rg\ pc'\ rg'\ mem\ mem'. \\ \quad (\exists refs.\ (\exists ic'.\ \mathsf{ProgSteps}\ p\ prog\ refs\ (pc, ic, rg)\ (pc', ic', rg')) \wedge \\ \qquad\quad (\exists ord.\ \mathsf{Init}\ mem\ refs\ ord \wedge \\ \qquad\qquad\quad \mathsf{IsBefore}\ refs\ ord \wedge \\ \qquad\qquad\quad (mem' = \mathsf{Final}\ refs\ ord\ mem)) \\ \quad = \\ \quad \mathsf{UniProgSteps}\ prog\ (pc, rg, mem)\ (pc', rg', mem') \end{array}$$

This result is, perhaps, of limited utility as it assumes that there is only one processor. Its main significance is that it reassures us that the apparently rather ad hoc collection of properties constituting the definition of $\mathsf{IsBefore}$ are sound and complete.

A second result, that promises to be more useful, establishes that a set of programs running on a true multiprocessor can be 'factored' into a conjunction of uniprocessors if the programs access disjoint sets of references. For simplicity, this will only be stated for the case of two processors.

Theorem 5: Let $\mathsf{DisjointLocs}\ prog_0\ prog_1$ be true if and only if $prog_0$ and $prog_1$ access disjoint sets of locations. Then:

$$
\begin{array}{l}
\vdash \forall mem\ p_0\ prog_0\ pc_0\ ic_0\ rg_0\ pc_0'\ rg_0'\ mem_0'\ p_1\ prog_1\ pc_1\ ic_1\ rg_1\ pc_1'\ rg_1'\ mem_1'.\\
\quad \mathsf{DisjointLocs}\ proc_0\ proc_1 \wedge \neg(p_0 = p_1)\\
\quad \Rightarrow\\
\quad ((\exists refs_0\ refs_1.\ (\exists ic_0'.\ \mathsf{ProgSteps}\ p_0\ prog_0\ refs_0\ (pc_0, ic_0, rg_0)\ (pc_0', ic_0', rg_0')) \wedge\\
\qquad\qquad\qquad (\exists ic_1'.\ \mathsf{ProgSteps}\ p_1\ prog_1\ refs_1\ (pc_1, ic_1, rg_1)\ (pc_1', ic_1', rg_1')) \wedge\\
\qquad\qquad\qquad (\exists ord.\ \mathsf{Init}\ mem\ (refs_0 \cup refs_1)\ ord \wedge\\
\qquad\qquad\qquad\qquad \mathsf{IsBefore}\ (refs_0 \cup refs_1)\ ord \wedge\\
\qquad\qquad\qquad\qquad (mem_0' = \mathsf{Final}\ refs_0\ (\mathsf{Restrict}\ ord\ refs_0)\ mem) \wedge\\
\qquad\qquad\qquad\qquad (mem_1' = \mathsf{Final}\ refs_1\ (\mathsf{Restrict}\ ord\ refs_1)\ mem)))\\
\quad =\\
\quad \mathsf{UniProgSteps}\ prog_0\ (pc_0, rg_0, mem_0)\ (pc_0', rg_0', mem_0') \wedge\\
\quad \mathsf{UniProgSteps}\ prog_1\ (pc_1, rg_1, mem_1)\ (pc_1', rg_1', mem_1'))
\end{array}
$$

9 Acknowledgements

The work reported here started during a four week visit I made to DEC's Systems Research Center in March/April, 1993. The main ideas were worked out then through a combination of conversations with Martin Abadi, Jim Horning, Jim Saxe and Yuan Yu.

I am particularly grateful to Yuan Yu for many hours of help and discussions, to Martin Abadi for hosting my visit to SRC and to Digital for supporting it.

References

[1] *The SPARC Architecture Manual.* Prentice Hall, 1992.

[2] M. J. C. Gordon and ??? A formalization of a simplified subset of the Alpha shared memory model. 1993.

[3] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem-proving environment for higher-order logic.* Cambridge University Press, 1993.

[4] Richard L. Sites, editor. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[5] Yuan Yu. *Automated Proofs of Object Code for a Widely Used Microprocessor.* PhD thesis, The University of Texas at Austin, yuanyu@com.dec.src, 1992.